

VMS DECwindows Guide to Application Programming

Order Number: AA-MG21B-TE

June 1990

This manual is a guide to creating applications using the XUI Toolkit, including the User Interface Language (UIL) and the XUI Resource Manager (DRM).

Revision/Update Information: This manual supersedes the *VMS DECwindows Guide to Application Programming, Version 5.3*.

Software Version: VMS Version 5.4

**digital equipment corporation
maynard, massachusetts**

June 1990

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.


Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1990.

All Rights Reserved.
Printed in U.S.A.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	DEQNA	MicroVAX	VAX RMS
DDIF	Desktop-VMS	PrintServer 40	VAXserver
DEC	DIGITAL	Q-bus	VAXstation
DECdtm	GIGI	ReGIS	VMS
DECnet	HSC	ULTRIX	VT
DECUS	LiveLink	UNIBUS	XUI
DECwindows	LN03	VAX	
DECwriter	MASSBUS	VAXcluster	

The following are third-party trademarks:

PostScript is a registered trademark of Adobe Systems Incorporated.

X Window System, Version 11 and its derivations (X, X11, X Version 11, X Window System) are trademarks of the Massachusetts Institute of Technology.

ZK4734

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

PREFACE

xxxi

CHAPTER 1 OVERVIEW OF THE XUI TOOLKIT

1-1

1.1	OVERVIEW OF XUI TOOLKIT COMPONENTS	1-1
1.1.1	User Interface Objects _____	1-2
1.1.2	X Toolkit Routines _____	1-3
1.1.3	Cut and Paste Routines _____	1-4
1.1.4	Application Development Tools _____	1-4
<hr/>		
1.2	PROGRAMMING CONCEPTS	1-4
1.2.1	Creating the Form of Your Application _____	1-6
1.2.2	Associating Function with Form _____	1-7
<hr/>		
1.3	WIDGETS IN THE XUI TOOLKIT	1-8
<hr/>		
1.4	WIDGET ATTRIBUTES	1-14
1.4.1	Size and Position Attributes _____	1-14
1.4.2	Appearance Attributes _____	1-15
1.4.3	Callback Attributes _____	1-15
1.4.4	Assigning Values to Widget Attributes _____	1-15

CHAPTER 2 CREATING A VMS DECWINDOWS APPLICATION

2-1

2.1	OVERVIEW OF A VMS DECWINDOWS APPLICATION	2-1
<hr/>		
2.2	SYMBOL DEFINITION FILES	2-2
<hr/>		
2.3	INITIALIZING THE XUI TOOLKIT	2-4
2.3.1	Application Shell Widget _____	2-6
2.3.2	Using Multiple Shell Widgets _____	2-7

Contents

2.4	CREATING THE WIDGETS IN THE INTERFACE	2-8
2.4.1	Using Low-Level Widget Creation Routines _____	2-9
2.4.1.1	Using Low-Level Routines to Define the Parent/Child Relationship of a Widget • 2-10	
2.4.1.2	Using Low-Level Routines to Define the Initial Appearance of a Widget • 2-10	
2.4.1.3	Using Low-Level Routines to Associate Callback Routines with a Widget • 2-12	
2.4.2	Using High-Level Widget Creation Routines _____	2-14
2.4.2.1	Using High-Level Routines to Define the Parent/Child Relationship of a Widget • 2-15	
2.4.2.2	Using High-Level Routines to Define the Initial Appearance of a Widget • 2-15	
2.4.2.3	Using High-Level Routines to Associate Callback Routines with a Widget • 2-16	
2.4.3	Using UIL and DRM to Create Widgets _____	2-17
2.4.3.1	Using UIL to Define the Parent/Child Relationship of a Widget • 2-18	
2.4.3.2	Using UIL to Define the Initial Appearance of a Widget • 2-18	
2.4.3.3	Using UIL to Associate Callbacks with a Widget • 2-19	

2.5	MANAGING THE WIDGETS IN THE INTERFACE	2-21
2.5.1	Managing a Single Child Widget _____	2-22
2.5.2	Managing Multiple Child Widgets _____	2-23

2.6	REALIZING THE WIDGETS IN THE INTERFACE	2-24
------------	---	-------------

2.7	MAIN INPUT LOOP	2-25
------------	------------------------	-------------

2.8	CREATING A CALLBACK ROUTINE	2-27
2.8.1	Identifying the Widget Performing the Callback _____	2-27
2.8.2	Associating Application-Specific Data with a Widget _____	2-27
2.8.3	Widget-Specific Callback Data _____	2-27
2.8.4	Guidelines for Creating Callback Routines _____	2-29

2.9	MANIPULATING THE INTERFACE AT RUN TIME	2-30
2.9.1	Standard Widget Manipulation Routines _____	2-31
2.9.2	Widget-Specific Manipulation Routines _____	2-32

2.10	COMPLETE LISTING OF THE HELLO WORLD! SAMPLE APPLICATION	2-32
2.10.1	Using Low-Level Routines to Create the Hello World! User Interface	2-32
2.10.2	Using High-Level Routines to Create the Hello World! User Interface	2-33
2.10.3	Using UIL and DRM to Create the Hello World! User Interface	2-35
2.10.4	The Hello World! Sample Application Main Input Loop and Callback Routine	2-38

CHAPTER 3 CREATING A USER INTERFACE USING UIL AND DRM **3-1**

3.1	OVERVIEW OF UIL AND DRM	3-1
3.2	SPECIFYING A USER INTERFACE USING UIL—A SAMPLE PROGRAM	3-4
3.2.1	Recommended UIL Coding Techniques	3-5
3.2.1.1	Naming Values and Objects • 3-6	
3.2.1.2	Declaring Values, Identifiers, and Procedures • 3-6	
3.2.1.3	Declaring Objects • 3-7	
3.2.1.4	Using Local Definitions for Objects • 3-10	
3.2.2	Creating a UIL Specification File	3-10
3.2.3	Structure of a UIL Module	3-11
3.2.4	Declaring the UIL Module	3-12
3.2.5	Using the UIL Constants Include File	3-13
3.2.6	Declaring Procedures in UIL	3-15
3.2.7	Declaring Values in UIL	3-16
3.2.7.1	Defining Arguments for Attached Dialog Box Widgets • 3-17	
3.2.7.2	Defining Integer Values • 3-18	
3.2.7.3	Defining String Values • 3-18	
3.2.7.4	Specifying Multiline Compound Strings • 3-20	
3.2.7.5	Defining String Table Values • 3-20	
3.2.7.6	Defining Font Values • 3-21	
3.2.7.7	Defining Color Values • 3-22	
3.2.7.8	Defining Pixmap Values • 3-23	
3.2.8	Declaring Interface Objects in a UIL Module	3-24
3.2.8.1	Specifying Arguments in an Object Declaration • 3-25	
3.2.8.2	Specifying Children in an Object Declaration • 3-26	
3.2.8.3	Specifying Callbacks in an Object Declaration • 3-27	
3.2.9	Specifying an Icon as a Widget Label	3-29

Contents

3.3	CREATING A USER INTERFACE AT RUN TIME USING DRM	3-31
3.3.1	Accessing the UID File at Run Time	3-35
3.3.2	Deferring Fetching	3-37
3.3.3	Retrieving Literal Values from UID Files	3-38
3.3.4	Setting Values at Run Time Using UID Resources	3-40
3.3.5	Using an Object Definition as a Template	3-45
3.4	CUSTOMIZING A VMS DECWINDOWS INTERFACE USING UIL AND DRM	3-49
3.4.1	Designing an International Application Using UIL and DRM	3-50
3.4.2	Specifying the User Interface for an International Application	3-52
3.4.3	Creating the User Interface for an International Application	3-53
3.5	USING IDENTIFIERS IN UIL	3-57
3.6	USING SYMBOLIC REFERENCES TO WIDGET IDENTIFIERS IN UIL	3-58
3.7	DEVELOPING AND TESTING PROTOTYPES USING UIL	3-59
3.7.1	Setting Up the UIL Module for Prototype Testing	3-60
3.7.2	Setting Up the Application Program for Prototype Testing	3-62
3.8	USING UIL ON LARGE PROJECTS	3-63
3.9	WORKING WITH USER-DEFINED WIDGETS IN UIL	3-65
3.9.1	Defining Arguments and Reasons for a User-Defined Widget	3-66
3.9.2	Using a User-Defined Widget in an Interface Specification	3-68
3.9.3	Accessing a User-Defined Widget at Run Time	3-71
CHAPTER 4 CREATING A MAIN WINDOW WIDGET		4-1
4.1	OVERVIEW OF WINDOW WIDGETS	4-1
4.2	CHILDREN OF A MAIN WINDOW WIDGET	4-1
4.2.1	Menu Bar Widget	4-2
4.2.2	Command Window Widget	4-2
4.2.3	Scroll Bar Widgets	4-2
4.2.4	Work Area Widget	4-3

4.3	CREATING A MAIN WINDOW WIDGET	4-4
4.3.1	Adding Children to a Main Window Widget	4-5
4.3.1.1	Using SET VALUES to Add Children to a Main Window Widget • 4-6	
4.3.1.2	Using the MAIN WINDOW SET AREAS Routine • 4-6	
4.3.1.3	Accepting Main Window Widget Defaults • 4-6	
4.3.2	Customizing the Main Window Widget	4-7
4.3.3	Associating Callback Routines with a Main Window Widget	4-8
<hr/>		
4.4	CREATING A SCROLL WINDOW WIDGET	4-8
4.4.1	Adding Children to a Scroll Window Widget	4-10
4.4.1.1	Using SET VALUES to Add Children to a Scroll Window Widget • 4-10	
4.4.1.2	Using the SCROLL WINDOW SET AREAS Support Routine • 4-10	
4.4.1.3	Accepting Scroll Window Widget Defaults • 4-11	
<hr/>		
4.5	CREATING A WINDOW WIDGET	4-11
4.5.1	Drawing Graphics in a Window Widget	4-12
4.5.2	Associating Callback Routines with a Window Widget	4-14
<hr/>		
4.6	CREATING A COMMAND WINDOW WIDGET	4-15
4.6.1	Command Window Widget Support Routines	4-16
4.6.2	Specifying the Contents of the Command Line	4-16
4.6.3	Displaying Error Messages in the Command Window Widget	4-17
4.6.4	Defining Accelerators for the Command Window Widget	4-17
4.6.5	Customizing the Appearance of the Command Window Widget	4-17
4.6.5.1	Specifying the Command Line Prompt • 4-17	
4.6.5.2	Specifying the Size and Content of the Command History Window • 4-17	
4.6.6	Associating Callback Routines with the Command Window Widget	4-18
<hr/>		
CHAPTER 5	USING THE LABEL, SEPARATOR, AND BUTTON WIDGETS	5-1
<hr/>		
5.1	OVERVIEW OF LABEL, SEPARATOR, AND BUTTON WIDGETS AND GADGETS	5-1

Contents

5.2	CREATING A LABEL WIDGET OR GADGET	5-2
5.2.1	Customizing a Label Widget	5-3
5.2.1.1	Specifying the Size and Position of a Label Widget • 5-3	
5.2.1.2	Specifying the Alignment in a Label Widget • 5-4	
5.2.1.3	Specifying Margins in a Label Widget • 5-4	
5.2.1.4	Specifying the Content of a Label Widget • 5-5	
5.2.2	Customizing a Label Gadget	5-5
<hr/>		
5.3	CREATING A SEPARATOR WIDGET OR GADGET	5-6
5.3.1	Customizing a Separator Widget or Gadget	5-7
<hr/>		
5.4	CREATING A PUSH BUTTON WIDGET OR GADGET	5-7
5.4.1	Customizing a Push Button Widget	5-10
5.4.1.1	Specifying Highlighting Behavior • 5-10	
5.4.1.2	Specifying Shadowing • 5-10	
5.4.1.3	Specifying the Insensitive Pixmap • 5-10	
5.4.2	Customizing a Push Button Gadget	5-10
5.4.3	Associating Callback Routines with a Push Button Widget or Gadget	5-11
<hr/>		
5.5	CREATING A TOGGLE BUTTON WIDGET OR GADGET	5-12
5.5.1	Specifying the State of a Toggle Button Widget or Gadget	5-16
5.5.2	Customizing a Toggle Button Widget	5-17
5.5.2.1	Specifying the Appearance of the Indicator • 5-17	
5.5.2.2	Specifying On and Off Pixmap • 5-17	
5.5.3	Customizing a Toggle Button Gadget	5-18
5.5.4	Associating Callback Routines with a Toggle Button Widget or Gadget	5-18
<hr/>		
5.6	WORKING WITH COMPOUND STRINGS	5-19
5.6.1	Creating a Compound String	5-21
5.6.2	Creating Compound Strings with Multiple Segments	5-22
5.6.3	Manipulating a Compound String	5-23
5.6.4	Retrieving Information About a Compound String	5-23
5.6.5	Specifying Fonts	5-25
<hr/>		
5.7	DEFINING ACCELERATORS FOR BUTTON WIDGETS AND GADGETS	5-27
5.7.1	Defining the Accelerator Key or Key Combination	5-27
5.7.2	Adding an Accelerator to a Widget or Gadget	5-28
5.7.3	Installing an Accelerator in an Application	5-28
5.7.4	Specifying an Accelerator Label	5-29

5.7.5	Adding an Accelerator to the Hello World! Sample Application	5-30
-------	--	------

CHAPTER 6	CREATING MENU WIDGETS	6-1
------------------	------------------------------	------------

6.1	OVERVIEW OF MENU WIDGETS	6-1
------------	---------------------------------	------------

6.2	MENU WIDGETS IN THE XUI TOOLKIT	6-1
------------	--	------------

6.2.1	Creating Menu Items	6-2
--------------	----------------------------	------------

6.2.2	Nesting Menu Widgets	6-4
--------------	-----------------------------	------------

6.3	CREATING A WORK AREA MENU WIDGET	6-5
------------	---	------------

6.3.1	Customizing a Work Area Menu Widget	6-9
--------------	--	------------

6.3.1.1	Specifying the Size of a Work Area Menu Widget • 6-9
---------	--

6.3.1.2	Specifying the Arrangement of Menu Items • 6-10
---------	---

6.3.1.3	Specifying Margins and Spacing • 6-10
---------	---------------------------------------

6.3.1.4	Determining Menu Item Alignment • 6-11
---------	--

6.3.1.5	Specifying Radio Button Exclusivity • 6-11
---------	--

6.3.1.6	Restricting Menu Items to Classes of Widgets • 6-11
---------	---

6.3.2	Associating Callback Routines with a Work Area Menu Widget	6-12
--------------	---	-------------

6.4	CREATING A PULL-DOWN MENU WIDGET	6-12
------------	---	-------------

6.4.1	Customizing the Appearance of a Pull-Down Menu Widget	6-14
--------------	--	-------------

6.4.2	Associating Callback Routines with a Pull-Down Menu Widget	6-14
--------------	---	-------------

6.5	CREATING A MENU BAR WIDGET	6-15
------------	-----------------------------------	-------------

6.5.1	Customizing a Menu Bar Widget	6-19
--------------	--------------------------------------	-------------

6.6	CREATING AN OPTION MENU WIDGET	6-19
------------	---------------------------------------	-------------

6.6.1	Customizing an Option Menu Widget	6-23
--------------	--	-------------

6.6.1.1	Specifying the Initial Value of an Option Menu Widget • 6-23
---------	--

6.6.1.2	Specifying the Label in an Option Menu Widget • 6-24
---------	--

6.7	CREATING A POP-UP MENU WIDGET	6-24
------------	--------------------------------------	-------------

6.7.1	Creating an Action Procedure	6-26
--------------	-------------------------------------	-------------

6.7.2	Adding an Action Procedure to a Widget	6-27
--------------	---	-------------

6.7.3	Customizing a Pop-Up Menu Widget	6-31
--------------	---	-------------

6.7.4	Associating Callback Routines with a Pop-Up Menu Widget	6-31
--------------	--	-------------

Contents

CHAPTER 7	CREATING DIALOG BOX WIDGETS	7-1
<hr/>		
7.1	OVERVIEW OF THE DIALOG BOX WIDGET	7-1
<hr/>		
7.2	DIALOG BOX WIDGETS IN THE XUI TOOLKIT	7-1
7.2.1	Generic Dialog Box Widgets	7-1
7.2.1.1	Dialog Box Widget • 7-2	
7.2.1.2	Attached Dialog Box Widget • 7-2	
7.2.2	Standard Dialog Box Widgets	7-4
7.2.2.1	Message Box Widget • 7-4	
7.2.2.2	Selection Box Widget • 7-4	
<hr/>		
7.3	STYLES OF DIALOG BOX WIDGETS	7-4
<hr/>		
7.4	CREATING A DIALOG BOX WIDGET	7-5
7.4.1	Specifying the Layout of Children in a Dialog Box Widget	7-6
7.4.2	Customizing the Dialog Box Widget	7-10
7.4.2.1	Sizing and Resizing a Dialog Box Widget • 7-10	
7.4.2.2	Positioning a Dialog Box Widget • 7-11	
7.4.2.3	Selecting the Unit of Measure Used in a Dialog Box Widget • 7-11	
7.4.2.4	Defining Translations for Simple Text Widgets • 7-11	
7.4.2.5	Assigning Accelerators to Child Widgets • 7-12	
7.4.2.6	Grabbing the Input Focus • 7-12	
7.4.3	Associating Callback Routines with a Dialog Box Widget	7-12
<hr/>		
7.5	CREATING AN ATTACHED DIALOG BOX WIDGET	7-13
7.5.1	Defining Attachments in an Attached Dialog Box Widget	7-14
7.5.1.1	Attaching an Edge to the Attached Dialog Box • 7-16	
7.5.1.2	Attaching an Edge to Another Child Widget • 7-17	
7.5.1.3	Attaching an Edge to a Position in the Attached Dialog Box Widget • 7-18	
7.5.1.4	Accepting Default Attachments • 7-19	
7.5.2	Using Attachment Attributes	7-19
7.5.3	Customizing an Attached Dialog Box Widget	7-21
7.5.3.1	Specifying the Default Spacing Between Child Widgets • 7-22	
7.5.3.2	Defining the Default Denominator Used in Fraction Positioning • 7-22	
7.5.3.3	Controlling Resizing Behavior of Child Widgets • 7-22	
7.5.4	Associating Callback Routines with an Attached Dialog Box Widget	7-22

CHAPTER 8 CREATING A LIST BOX WIDGET		8-1
8.1	OVERVIEW OF THE LIST BOX WIDGET	8-1
8.2	CREATING A LIST BOX WIDGET	8-2
8.2.1	Creating an Item List	8-3
8.2.1.1	Creating an Item List as an Array of Compound Strings • 8-3	
8.2.1.2	Creating an Item List Using the UIL STRING TABLE Function • 8-5	
8.2.2	Selecting and Canceling Selections of List Items	8-6
8.3	LIST BOX WIDGET SUPPORT ROUTINES	8-8
8.3.1	Adding and Deleting List Items at Run Time	8-9
8.3.1.1	Using SET VALUES to Add or Delete List Items • 8-9	
8.3.1.2	Using a Support Routine to Add an Item to an Item List • 8-10	
8.3.1.3	Using a Support Routine to Delete an Item from an Item List • 8-10	
8.3.2	Selecting and Canceling the Selection of List Items at Run Time	8-11
8.3.2.1	Using the SET VALUES Intrinsic Routine to Select List Items • 8-11	
8.3.2.2	Using a Support Routine to Select a List Item • 8-11	
8.3.2.3	Canceling the Selection of Items in an Item List • 8-12	
8.3.3	Customizing the Appearance of a List Box Widget	8-12
8.3.3.1	Specifying the Size of a List Box Widget • 8-12	
8.3.3.2	Specifying List Items to Be Visible • 8-14	
8.3.3.3	Specifying Margins and Spacing in a List Box Widget • 8-14	
8.3.4	Associating Callbacks with a List Box Widget	8-15
CHAPTER 9 HANDLING TEXT		9-1
9.1	OVERVIEW OF TEXT WIDGETS	9-1
9.2	CREATING TEXT WIDGETS	9-4
9.2.1	Manipulating the Text Contents of the Text Widgets	9-6
9.2.1.1	Placing Text in a Text Widget • 9-6	
9.2.1.2	Retrieving Text from a Text Widget • 9-7	
9.2.1.3	Disabling Text Editing • 9-7	
9.2.1.4	Limiting the Length of the Text • 9-8	
9.2.2	Customizing the Appearance of the Text Widgets	9-8
9.2.2.1	Specifying Size • 9-8	

Contents

9.2.2.2	Specifying Margins • 9–9	
9.2.2.3	Controlling Resizing Behavior • 9–10	
9.2.2.4	Controlling Text Cursor Appearance • 9–10	
9.2.2.5	Positioning the Insertion Point • 9–11	
9.2.2.6	Specifying Border Visibility and Color • 9–11	
9.2.2.7	Identifying the Current Writing and Editing Directions • 9–11	
9.2.3	Handling Text Selections _____	9–12
9.2.3.1	Selecting Text • 9–12	
9.2.3.2	Retrieving Selected Text • 9–12	
9.2.3.3	Canceling the Selection of Text • 9–13	
9.2.4	Associating Callbacks with Text Widgets _____	9–13

CHAPTER 10 USING THE SCALE AND THE SCROLL BAR WIDGETS 10–1

10.1	OVERVIEW OF THE SCALE WIDGET	10–1
10.2	CREATING A SCALE WIDGET	10–2
10.2.1	Determining the Range of Values _____	10–3
10.2.2	Customizing the Appearance of a Scale Widget _____	10–4
10.2.2.1	Specifying the Size of a Scale Widget • 10–4	
10.2.2.2	Specifying the Orientation of the Scale Widget • 10–5	
10.2.2.3	Specifying the Title of the Scale Widget • 10–5	
10.2.2.4	Specifying the Color of the Slider • 10–6	
10.2.2.5	Representing the Value of the Scale • 10–6	
10.2.2.6	Adding Labeled Tick Marks to a Scale Widget • 10–7	
10.2.3	Associating Callbacks with a Scale Widget _____	10–8
10.3	OVERVIEW OF THE SCROLL BAR WIDGET	10–10
10.4	CREATING A SCROLL BAR WIDGET	10–11
10.4.1	Determining the Range of a Scroll Bar Widget _____	10–12
10.4.2	Specifying the Size of the Slider in a Scroll Bar Widget _____	10–13
10.4.3	Defining the Size of Increment and Decrement _____	10–13
10.4.4	Modifying the Action of the Stepping Arrows _____	10–14
10.4.5	Customizing the Appearance of the Scroll Bar Widget _____	10–14
10.4.6	Associating Callbacks with a Scroll Bar Widget _____	10–15

CHAPTER 11 USING THE COLOR MIXING WIDGET		11-1
11.1	OVERVIEW OF THE COLOR MIXING WIDGET	11-1
11.1.1	Color Models	11-1
11.1.2	Components of the Color Mixing Widget	11-2
11.1.2.1	Color Display Subwidget • 11-4	
11.1.2.2	Color Model Option Menu Subwidget • 11-5	
11.1.2.3	Color Mixer Subwidget • 11-5	
11.1.2.4	Push Button Subwidgets • 11-6	
11.1.2.5	Label Subwidgets • 11-7	
11.1.2.6	Work Area Subwidget • 11-7	
11.2	CREATING A COLOR MIXING WIDGET	11-7
11.2.1	Setting and Retrieving New Color Values	11-8
11.2.2	Customizing the Color Mixing Widget	11-9
11.2.2.1	Specifying the Size • 11-9	
11.2.2.2	Specifying Margins • 11-9	
11.2.2.3	Labeling the Color Mixing Widget • 11-10	
11.2.2.4	Defining the Background Color of the Color Display Subwidget • 11-13	
11.2.2.5	Adding a Work Area to the Color Mixing Widget • 11-13	
11.3	SUPPORTING OTHER COLOR MODELS	11-14
11.3.1	Replacing the Color Display Subwidget	11-14
11.3.2	Replacing the Color Mixer Subwidget	11-14
11.4	ASSOCIATING CALLBACKS WITH A COLOR MIXING WIDGET	11-14
CHAPTER 12 USING HELP		12-1
12.1	OVERVIEW OF THE HELP WIDGET	12-1
12.1.1	Help Widget Terminology	12-3
12.2	HELP LIBRARY INFORMATION	12-3
12.2.1	VMS Help Library Enhancements	12-4

Contents

12.3	MODIFYING HELP WIDGET APPEARANCE	12-7
12.3.1	Help Widget Topic Information	12-7
12.4	USING THE HELP WIDGET	12-8
12.5	CONTEXT-SENSITIVE HELP	12-13
<hr/>		
CHAPTER 13	USING THE CUT AND PASTE ROUTINES	13-1
13.1	OVERVIEW OF THE CUT AND PASTE ROUTINES	13-1
13.1.1	Communicating with Other Applications	13-3
13.1.2	Implementing the Copy, Cut, and Paste Functions	13-3
13.2	COPYING DATA TO THE CLIPBOARD	13-5
13.2.1	Copying Data to the Clipboard by Name	13-9
13.2.2	Creating a Clipboard Callback Routine	13-10
13.2.3	Deleting Data from the Clipboard	13-11
13.2.4	Specifying Clipboard Data Formats	13-11
13.3	COPYING DATA FROM THE CLIPBOARD	13-11
13.4	INQUIRING ABOUT CLIPBOARD CONTENTS	13-15
13.5	QUICKCOPY IMPLEMENTATION	13-16
13.5.1	QuickCopy Message Types	13-16
13.5.2	Selection Threshold Resource	13-17
13.5.3	Implementing the QuickCopy Function	13-17
13.5.3.1	CopyFrom and MoveFrom Operations • 13-17	
13.5.3.2	CopyTo and MoveTo Operations • 13-22	
<hr/>		
CHAPTER 14	COMMUNICATING WITH THE WINDOW MANAGER	14-1
14.1	OVERVIEW	14-1

14.2	MAKING REQUESTS OF THE WINDOW MANAGER	14-1
14.2.1	Using Window Properties	14-2
14.2.1.1	Predefined Window Properties • 14-2	
14.2.1.2	Vendor-Specific Window Properties • 14-4	
14.2.2	Using Shell Widget Attributes	14-8
<hr/>		
14.3	SETTING AND RETRIEVING PREDEFINED WINDOW MANAGER PROPERTIES	14-8
<hr/>		
14.4	SETTING AND RETRIEVING VENDOR-SPECIFIC WINDOW MANAGER PROPERTIES	14-10
<hr/>		
14.5	SETTING AND RETRIEVING SHELL WIDGET ATTRIBUTES	14-11
14.5.1	Setting Shell Widget Attributes at Widget Creation Time	14-11
14.5.2	Setting Shell Widget Attributes After Creation Time	14-13
<hr/>		
14.6	RECEIVING MESSAGES FROM THE WINDOW MANAGER	14-14
<hr/>		
14.7	CUSTOMIZING YOUR APPLICATION USING WINDOW MANAGER HINTS	14-14
14.7.1	Customizing Your Main Application Window	14-17
14.7.1.1	Associating a Name with Your Main Application Window • 14-18	
14.7.1.2	Specifying the Initial Size and Position of Your Application • 14-19	
14.7.1.3	Customizing the Title Bar • 14-19	
14.7.1.4	Including Shrink-to-Icon, Push-to-Back, and Resize Buttons in the Title Bar • 14-22	
14.7.2	Getting Information About Your Main Application Window	14-22
14.7.3	Customizing Your Application Icon	14-23
14.7.3.1	Specifying the Text in the Icon • 14-24	
14.7.3.2	Specifying the Pixmap Used in Your Application Icon • 14-24	
14.7.3.3	Using a Window in Your Icon • 14-26	
14.7.3.4	Positioning Your Icon on the Display • 14-27	
14.7.4	Specifying the Initial State of Your Application	14-27
14.7.5	Creating Transient and Sticky Windows	14-27
14.7.6	Bypassing the Window Manager	14-27

APPENDIX A	USING THE DECTERM PORT ROUTINE	A-1
	DECTERM PORT	A-3

Contents

APPENDIX B USING THE VAX BINDINGS B-1

B.1	USING THE DECWINDOWS ADA PROGRAMMING INTERFACES	B-1
B.1.1	Using the Ada Packages	B-2
B.1.1.1	Package CDA • B-3	
B.1.1.2	Package DDIF • B-3	
B.1.1.3	Package DTIF • B-3	
B.1.1.4	Package DWT • B-4	
B.1.1.5	Package X • B-4	
B.1.2	Callbacks	B-6
B.1.3	Tasking Considerations	B-6
B.1.4	Ada Examples	B-7

B.2 USING THE FORTRAN BINDINGS B-11

B.3 USING THE VAX PASCAL BINDINGS B-14

APPENDIX C INTERNATIONAL VERSION OF THE DECBURGER APPLICATION C-1

APPENDIX D BUILDING YOUR OWN WIDGETS D-1

D.1	OVERVIEW OF WIDGETS	D-1
D.1.1	Building a Widget	D-1
D.1.2	Building a Sample Widget	D-2

D.2	WIDGET CLASS DEFINITIONS	D-10
D.2.1	Core Widgets	D-10
D.2.1.1	CoreClassPart Structure • D-11	
D.2.1.2	CorePart Structure • D-12	
D.2.1.3	CorePart Default Values • D-12	
D.2.2	Composite Widgets	D-14
D.2.2.1	CompositeClassPart Structure • D-14	
D.2.2.2	CompositePart Structure • D-14	
D.2.2.3	CompositePart Default Values • D-15	
D.2.3	Constraint Widgets	D-15
D.2.3.1	ConstraintClassPart Structure • D-15	
D.2.3.2	ConstraintPart Structure • D-16	

D.3	WIDGET CLASSING	D-16
D.3.1	Widget Naming Conventions _____	D-17
D.3.2	Widget Subclassing in Public .h Files _____	D-18
D.3.3	Widget Subclassing in Private .h Files _____	D-19
D.3.4	Widget Subclassing in .c Files _____	D-20
D.3.5	Superclass Chaining _____	D-23
D.3.6	Class Initialization _____	D-24
D.3.7	Inheritance of Superclass Operations _____	D-25
D.3.8	Invocation of Superclass Operations _____	D-27
<hr/>		
D.4	CREATING INSTANCES OF WIDGETS TO BUILD A USER INTERFACE	D-27
D.4.1	Widget Instance Initialization _____	D-28
D.4.2	Constraint Widget Instance Initialization _____	D-29
D.4.3	Nonwidget Data Initialization _____	D-30
D.4.4	Widget Instance Window Creation _____	D-30
D.4.5	Dynamic Data Deallocation _____	D-31
D.4.6	Dynamic Constraint Data Deallocation _____	D-32
<hr/>		
D.5	COMPOSITE WIDGETS AND THEIR CHILDREN	D-32
D.5.1	Addition of Children to a Composite Widget _____	D-34
D.5.2	Insertion Order of Children _____	D-34
D.5.3	Deleting Children _____	D-35
D.5.4	Constrained Composite Widgets _____	D-35
<hr/>		
D.6	GEOMETRY MANAGEMENT	D-37
D.6.1	Initiating Geometry Changes _____	D-37
D.6.2	General Geometry Manager Requests _____	D-38
D.6.3	Resize Requests _____	D-39
D.6.4	Potential Geometry Changes _____	D-39
D.6.5	Child Geometry Management _____	D-40
D.6.6	Widget Placement and Sizing _____	D-41
D.6.7	Obtaining the Preferred Geometry _____	D-42
D.6.8	Managing Size Changes _____	D-43
<hr/>		
D.7	EVENT MANAGEMENT	D-44
D.7.1	X Event Filters _____	D-44
D.7.1.1	Pointer Motion Compression • D-45	
D.7.1.2	Enter/Leave Compression • D-45	
D.7.1.3	Exposure Compression • D-45	
D.7.2	Widget Exposure and Visibility _____	D-45
D.7.2.1	Redisplay of a Widget • D-45	
D.7.2.2	Widget Visibility • D-47	

Contents

D.7.3	X Event Handlers _____	D-47
<hr/>		
D.8	RESOURCE MANAGEMENT	D-48
D.8.1	Resource Lists _____	D-48
D.8.2	Superclass to Subclass Chaining of Resource Lists _____	D-52
D.8.3	Retrieving Subresources _____	D-52
D.8.4	Obtaining Application Resources _____	D-52
D.8.5	Resource Conversions _____	D-52
D.8.5.1	Predefined Resource Converters • D-53	
D.8.5.2	New Resource Converters • D-54	
D.8.6	Reading and Writing Widget Resource Fields _____	D-56
D.8.6.1	Widget Subpart Resource Data • D-57	
D.8.7	Setting Widget Resource Fields _____	D-57
D.8.7.1	Specifying Widget State • D-57	
D.8.7.2	Specifying Widget Geometry Values • D-58	
D.8.7.3	Specifying Widget Constraint Information • D-59	
D.8.7.4	Specifying the Widget Subpart Resources • D-59	
<hr/>		
D.9	TRANSLATION MANAGEMENT	D-60
D.9.1	Action Tables _____	D-60
D.9.2	Translating Action Names to Procedures _____	D-61
D.9.3	Translation Tables _____	D-62
D.9.3.1	Event Sequences • D-62	
D.9.3.2	Action Sequences • D-63	
D.9.4	Translation Table Syntax _____	D-63
D.9.4.1	Modifier Names in a Translation Table • D-63	
D.9.4.2	Event Types • D-66	
D.9.4.3	Canonical Representation • D-68	
D.9.5	Translation Table Management _____	D-71
D.9.6	Using Accelerators _____	D-71
D.9.7	Key Code to Key Symbol Conversions _____	D-72

GLOSSARY

Glossary-1

INDEX

EXAMPLES

2-1	Including the XUI Toolkit Symbol Definition File in an Application _____	2-3
2-2	Initializing the XUI Toolkit _____	2-4
2-3	Creating Your Own Application Context _____	2-5
2-4	Creating a User Interface Using Low-Level Routines _____	2-13
2-5	Creating a User Interface Using High-Level Routines _____	2-16
2-6	Using UIL to Define a Widget _____	2-19
2-7	Creating the Interface at Run Time Using DRM _____	2-21
2-8	Managing a Single Widget _____	2-22
2-9	Managing a Group of Child Widgets _____	2-23
2-10	Realizing a Widget Hierarchy _____	2-25
2-11	Entering the Main Input Loop _____	2-26
2-12	Hello World! Application Callback Routine _____	2-29
2-13	Adding a Work Procedure _____	2-30
2-14	Setup Section of the Hello World! Application Using Low-Level Routines _____	2-32
2-15	Setup Section of the Hello World! Application Using High-Level Routines _____	2-34
2-16	Hello World! Application UIL Specification File _____	2-35
2-17	Hello World! Application Using UIL _____	2-36
2-18	Main Input Loop and Callback Routine of the Hello World! Application _____	2-38
3-1	Widget Hierarchy in the DECburger UIL Module _____	3-10
3-2	UIL Module Structure _____	3-12
3-3	Module Declaration in the DECburger UIL Module _____	3-13
3-4	Constants from Include File in the DECburger UIL Module _____	3-14
3-5	Procedure Declaration in the DECburger UIL Module _____	3-16
3-6	Defining Integer Values in the DECburger UIL Module _____	3-18
3-7	Defining String Values in the DECburger UIL Module _____	3-19
3-8	Defining a String Table Value in the DECburger UIL Module _____	3-21
3-9	Declaring a Font Value in the DECburger UIL Module _____	3-21
3-10	Defining Colors in the DECburger UIL Module _____	3-22
3-11	Defining a Color Table in the DECburger UIL Module _____	3-23
3-12	Defining an Icon in the DECburger UIL Module _____	3-24
3-13	Declaring an Object in the DECburger UIL Module _____	3-25
3-14	Specifying Children in the DECburger UIL Module _____	3-27
3-15	Specifying Multiple Procedures per Callback Reason _____	3-29
3-16	Using an Icon as a Label in the DECburger UIL Module _____	3-31

Contents

3-17	Initializing DRM and the XUI Toolkit in the DECburger Application _____	3-35
3-18	Declaring the UID Hierarchy for the DECburger Application _____	3-36
3-19	Opening the UID Hierarchy for the DECburger Application _____	3-36
3-20	Declaring a Vector of Names to Register for DRM in the DECburger Application _____	3-37
3-21	Registering Names for DRM in the DECburger Application _____	3-37
3-22	DECburger UIL Module Setup for Deferred Fetching _____	3-38
3-23	Title Bar String for DECburger Application _____	3-39
3-24	Getting a Value from the UID File for the DECburger Application _____	3-39
3-25	UIL Module for the FETCH SET VALUES Application _____	3-42
3-26	C Program for the FETCH SET VALUES Application _____	3-44
3-27	UIL Module Setup for the FETCH WIDGET OVERRIDE Routine _____	3-47
3-28	Using the FETCH WIDGET OVERRIDE Routine in a C Program _____	3-48
3-29	French UIL Module for the International DECburger Application _____	3-52
3-30	C Program for the International DECburger Application _____	3-54
3-31	Using Identifiers in a UIL Module _____	3-57
3-32	Using Symbolic References in a UIL Module _____	3-59
3-33	Declarations in the DECburger UIL Module for Prototype Testing _____	3-61
3-34	Declaring an Unimplemented Object in the DECburger UIL Module _____	3-62
3-35	Definition of the Activate Routine in the DECburger Application _____	3-62
3-36	Sample Main UIL File _____	3-64
3-37	User-Defined XYZ Widget _____	3-67
3-38	Declaring the User-Defined XYZ Widget in a UIL Module _____	3-68
3-39	C Program for Displaying the XYZ User-Defined Widget _____	3-71
4-1	Main Window Created in the DECburger UIL Module _____	4-8
4-2	Performing Graphics Operations in a Window Widget _____	4-12
5-1	Push Button Gadgets in the DECburger Option Menu _____	5-9
5-2	Push Button Callback Procedure in the DECburger Application _____	5-12
5-3	Creating the Radio Box Widget in the DECburger Application _____	5-15
5-4	Setting the Initial State of a Toggle Button _____	5-16
5-5	Toggle Button Callback Procedure in the DECburger Application _____	5-19
5-6	Creating a Compound String _____	5-22

4-8	Widget Attributes Accessible Using the High-Level Routine WINDOW _____	4-12
4-9	Command Window Widget Creation Mechanisms _____	4-15
4-10	Widget Attributes Accessible Using the High-Level Routine COMMAND WINDOW _____	4-15
4-11	Command Window Widget Support Routines _____	4-16
4-12	Command Window Widget Callbacks _____	4-18
5-1	Label Widget and Gadget Creation Mechanisms _____	5-2
5-2	Attributes Accessible Using the High-Level Routine LABEL _____	5-3
5-3	Separator Widget and Gadget Creation Mechanisms _____	5-6
5-4	Attributes Accessible Using the High-Level Routine SEPARATOR _____	5-7
5-5	Push Button Widget and Gadget Creation Mechanisms _____	5-8
5-6	Attributes Accessible Using the High-Level Routine PUSH BUTTON _____	5-8
5-7	Push Button Widget and Gadget Callbacks _____	5-11
5-8	Toggle Button Widget and Gadget Creation Mechanisms _____	5-12
5-9	Attributes Accessible Using the High-Level Routine TOGGLE BUTTON _____	5-13
5-10	Toggle Button Widget and Gadget Callbacks _____	5-19
5-11	Compound String Routines _____	5-20
6-1	Work Area Menu Widget Creation Mechanisms _____	6-6
6-2	Attributes Accessible Using the High-Level Routine MENU _____	6-6
6-3	XUI Toolkit Widget and Gadget Class Names _____	6-12
6-4	Pull-Down Menu Widget Creation Mechanisms _____	6-13
6-5	Pull-Down Menu Entry Widget and Gadget Creation Mechanisms _____	6-13
6-6	Attributes Accessible Using the High-Level Routine MENU _____	6-14
6-7	Menu Bar Widget Creation Mechanisms _____	6-16
6-8	Attributes Accessible Using the High-Level Routine MENU BAR _____	6-17
6-9	Option Menu Widget Creation Mechanisms _____	6-20
6-10	Attributes Accessible Using the High-Level Routine OPTION MENU _____	6-21
6-11	Pop-Up Menu Widget Creation Mechanisms _____	6-25
6-12	Attributes Accessible Using the High-Level Routine MENU _____	6-25
7-1	Dialog Box Widget Creation Mechanisms _____	7-5
7-2	Attributes Accessible Using the High-Level Routine DIALOG BOX _____	7-6
7-3	Attached Dialog Box Widget Creation Mechanisms _____	7-13
7-4	Attributes Accessible Using the High-Level Routine ATTACHED DIALOG BOX _____	7-14

Contents

7-5	Attachment Attributes	7-15
7-6	Attachment Constants for the Attached Dialog Box Widget	7-15
8-1	List Box Widget Creation Mechanisms	8-2
8-2	Attributes Accessible Using the High-Level Routine LIST BOX	8-3
8-3	List Box Widget Support Routines	8-8
8-4	List Box Widget Callbacks	8-15
9-1	Text Widget Support Routines	9-3
9-2	Mechanisms for Creating Text Widgets	9-4
9-3	Attributes Accessible Using the High-Level Routines S TEXT and CS TEXT	9-5
9-4	Text Widget Callbacks	9-14
10-1	Scale Widget Creation Mechanisms	10-2
10-2	Attributes Accessible Using the High-Level Routine SCALE	10-3
10-3	Horizontal and Vertical Orientation Constants	10-5
10-4	Scale Widget Callbacks	10-9
10-5	Scroll Bar Widget Creation Mechanisms	10-11
10-6	Attributes Accessible Using the High-Level Routine SCROLL BAR	10-12
10-7	Scroll Widget Callbacks	10-15
11-1	Color Model Constants	11-5
11-2	Mechanisms for Creating the Color Mixing Widget	11-7
11-3	Support Routines for the Color Mixing Widget	11-9
11-4	Color Mixing Widget Label Attributes	11-10
11-5	Color Mixing Widget Callbacks	11-15
12-1	Help Widget Terminology	12-3
12-2	VMS Librarian Utility Extensions	12-5
12-3	Help Widget Appearance Attributes	12-7
12-4	Help Widget Topic Attributes	12-7
12-5	Help Widget Creation Routines	12-9
13-1	Cut and Paste Routines	13-1
13-2	Edit Menu Functions	13-5
13-3	QuickCopy Operations	13-16
14-1	Predefined Window Manager Properties	14-3
14-2	Members of the WM Hints Data Structure	14-4
14-3	Properties Defined by the DECwindows Window Manager	14-5
14-4	Members of the DEC WM Hints Data Structure	14-6
14-5	Members of the WM Decoration Geometry Data Structure	14-8
14-6	Xlib Routines for Setting and Retrieving Predefined Window Manager Properties	14-9
14-7	Common Tasks Performed with the Window Manager	14-16

14-8	Information Provided by the Window Manager _____	14-17
B-1	Subtype Definitions—Package DWT _____	B-4
B-2	Subtype Definitions—Package X _____	B-4
D-1	Default Values for the CorePart Structure _____	D-13
D-2	Default Values for the CompositePart Structure _____	D-15
D-3	Resource Types _____	D-49
D-4	Translation Table Modifiers _____	D-64
D-5	Event Types _____	D-66
D-6	Event Type Abbreviations for Translation Tables _____	D-67

Preface

This manual describes how to create an application using the XUI Toolkit, including the User Interface Language (UIL) and the XUI Resource Manager (DRM).

Intended Audience

This manual is intended for experienced programmers who want to learn how to use the components of the VMS DECwindows programming environment to create applications. Readers should be familiar with a high-level programming language.

Document Structure

This manual is organized as follows:

- Chapter 1 provides an overview of the XUI Toolkit, introduces the basic programming concepts of using the XUI Toolkit, and introduces the widgets in the XUI Toolkit.
- Chapter 2 describes the basic structure of a typical application program by examining a sample program, the Hello World! application.
- Chapter 3 describes how to create a user interface using the User Interface Language (UIL) and the XUI Resource Manager (DRM).
- Chapters 4 through 12 provide tutorials that show how to use the widgets in the XUI Toolkit and include code examples to illustrate the concepts described.
- Chapter 13 describes how to use the cut and paste routines.
- Chapter 14 describes how your application can communicate with the window manager.

The manual includes the following appendixes:

- Appendix A, Using the DECTERM PORT Routine, describes how to create a terminal window on a local or remote node.
- Appendix B, Using the VAX Bindings, presents three versions of the Hello World! sample application created in Chapter 2. The appendix includes versions of the program written in VAX Ada, VAX FORTRAN, and VAX Pascal. The appendix also includes specific information about using the Ada bindings.
- Appendix C, International Version of the DECburger Application, is the complete source listing for a version of DECburger that illustrates how to internationalize an application using UIL and DRM. Chapter 3 describes this example.

Preface

- Appendix D, *Building Your Own Widgets*, describes how to build your own widgets.
- The Glossary defines key terms used in this manual.

Associated Documents

For more information about topics covered in this manual, see the following manuals in the VMS DECwindows document set.

- *XUI Style Guide*
- *VMS DECwindows Toolkit Routines Reference Manual*
- *VMS DECwindows Xlib Routines Reference Manual*
- *VMS DECwindows Xlib Programming Volume*
- *VMS DECwindows User Interface Language Reference Manual*

Conventions

The following conventions are used in this manual:

mouse	The term <i>mouse</i> is used to refer to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (The buttons can be redefined by the user.)
Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
Return	In examples, a key name is shown enclosed in a box to indicate that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
.	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[]	In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.

5-7	Creating a Compound String with Multiple Segments _____	5-22
5-8	Extracting the Text Content from a Compound String _____	5-25
5-9	Specifying a Font _____	5-26
5-10	Adding an Accelerator to a Push Button Widget or Gadget _____	5-28
5-11	Adding an Accelerator to the Hello World! Application _____	5-30
6-1	Building a Work Area Menu _____	6-7
6-2	Creating the Menu Bar Widget in the DECburger Application _____	6-17
6-3	Creating the Option Menu Widget in the DECburger Application _____	6-22
6-4	Creating an Option Menu Widget with an Item Selected _____	6-23
6-5	Action Procedure to Pop Up a Pop-Up Menu Widget _____	6-27
6-6	Creating a Pop-Up Menu Widget _____	6-28
7-1	Creating the Dialog Box Widget in the DECburger Application _____	7-7
7-2	Creating the Children of the Dialog Box Widget in the DECburger Application _____	7-8
7-3	Positioning Children in an Attached Dialog Box Widget _____	7-19
8-1	Creating an Item List as an Array of Compound Strings _____	8-4
8-2	Creating an Item List Using the UIL STRING TABLE Function _____	8-5
8-3	Selecting an Item in an Item List _____	8-7
8-4	Adding an Item to a List Box Widget _____	8-10
8-5	Specifying the Size of the DECburger List Box Widget _____	8-13
8-6	Associating a Callback Routine with a List Box Widget _____	8-16
8-7	Callback Routine DECburger Associates with the List Box Widget _____	8-17
9-1	Defining the Simple Text Widget in the DECburger Sample Application _____	9-5
9-2	Using the S TEXT GET STRING Support Routine in the DECburger Sample Application _____	9-7
10-1	Determining the Range of Values _____	10-4
10-2	Setting Appearance Attributes of the Scale Widget in the DECburger Sample Application _____	10-6
10-3	Labeling Points Along a Scale in a Scale Widget _____	10-7
10-4	Associating a Callback Routine with a Scale Widget _____	10-9
10-5	Scale Widget Callback Routine in the DECburger Application _____	10-10
10-6	Specifying the Range of Values in a Scroll Bar Widget _____	10-13
11-1	Creating a Color Mixing Widget _____	11-8
12-1	Sample Help File _____	12-6
12-2	Creating a Help Widget _____	12-9

Contents

12-3	UIL Help Widget Implementation _____	12-11
13-1	Copying Data to the Clipboard _____	13-6
13-2	Copying Data from the Clipboard _____	13-12
13-3	Calling the OWN SELECTION Routine _____	13-18
13-4	Notifying the Receiving Application that Data Is Available ____	13-18
13-5	Getting the Selection Value _____	13-20
13-6	Getting the Secondary Selection Data _____	13-21
13-7	QuickCopy Callback Routine _____	13-22
13-8	Sending a KILL_SELECTION Message _____	13-23
14-1	Assigning Values to Predefined Window Manager Properties _____	14-9
14-2	Setting Vendor-Specific Window Manager Properties _____	14-10
14-3	Setting Shell Widget Attributes at Widget Creation Time ____	14-12
14-4	Using the SET VALUES Intrinsic Routine to Set Shell Widget Attributes _____	14-13
14-5	Specifying the Shrink-to-Icon Pixmap Using the CHANGE PROPERTY Xlib Routine _____	14-20
14-6	Using Shell Widget Attributes to Specify Your Application Icon _____	14-25
A-1	Creating a DECterm on a Remote Node _____	A-1
A-2	Command Procedure to Compile, Link, and Run a DECterm on a Remote Node _____	A-2
B-1	Hello World! Application in VAX Ada _____	B-7
B-2	Hello World! Application in VAX FORTRAN _____	B-11
B-3	Hello World! Application in VAX Pascal _____	B-15
C-1	C Program for the International Version of DECburger ____	C-2
D-1	Sample Widget _____	D-3
D-2	Modifying the Hello World! Application to Use the Sample Widget _____	D-9
D-3	Compiling and Linking the Sample Widget _____	D-10
D-4	The .c File for a Label Widget _____	D-21

FIGURES

1-1	XUI Layered Architecture _____	1-2
1-2	Hello World! Application User Interface _____	1-5
1-3	Application Widget Hierarchy of the Hello World! Application _____	1-7
1-4	DECburger User Interface _____	1-12
2-1	Structure of an XUI Application _____	2-2
2-2	Relationship of Shell Widget to Application _____	2-7

2-3	Argument Data Structure (VAX Binding) _____	2-11
2-4	Callback Routine Data Structure (VAX Binding) _____	2-12
2-5	Widget Callback Data Structure (VAX Binding) _____	2-28
3-1	Setting Up a User Interface Specified with UIL _____	3-2
3-2	Radio Box with Toggle Buttons in the DECburger Application _____	3-8
3-3	Widget Hierarchy for the DECburger Radio Box Widget _____	3-9
3-4	Using an Icon in the DECburger Application Interface _____	3-30
3-5	Widget Creation in a DRM Fetch Operation _____	3-33
3-6	Sample Application Using the FETCH SET VALUES Routine _____	3-41
3-7	Using UID Hierarchies to Provide Alternatives or Refinements to an Interface _____	3-50
4-1	Main Window Widget _____	4-4
5-1	Attributes for Setting Margins _____	5-5
5-2	Radio Box with Toggle Button Gadgets in the DECburger Application _____	5-14
5-3	Hello World! Application with an Accelerator _____	5-29
6-1	Menu Widget _____	6-2
6-2	Relationship of Pull-Down Menu Widget and Pull-Down Menu Entry Widget or Gadget _____	6-4
6-3	Widget Hierarchy of Nested Pull-Down Menu Widgets _____	6-5
6-4	Widget Hierarchy of a Work Area Menu _____	6-9
6-5	Laying Out Menu Items _____	6-10
6-6	DECburger Menu Bar with a Pull-Down Menu Selected _____	6-15
6-7	Widget Hierarchy of the DECburger Menu Bar Widget _____	6-19
6-8	Option Menu Widget _____	6-20
6-9	Pop-Up Menu Widget _____	6-31
7-1	Resizing a Dialog Box Widget _____	7-3
7-2	Layout of the DECburger Dialog Box Widget _____	7-10
7-3	Attaching an Edge of a Child Widget to the Attached Dialog Box Widget _____	7-16
7-4	Attaching an Edge of a Child Widget to Another Child Widget in an Attached Dialog Box Widget _____	7-17
7-5	Attaching an Edge to a Position in an Attached Dialog Box _____	7-18
8-1	List Box Widget _____	8-1
8-2	List Box Widget Used in the DECburger User Interface _____	8-6
8-3	Margins and Spacing in a List Box Widget _____	8-15
9-1	Text Widgets _____	9-2
9-2	Default Configuration of the Text Widgets _____	9-9
10-1	Scale Widget _____	10-2
10-2	Scale Widget Sizing Attributes _____	10-5

Contents

10-3	Scroll Bar Widget _____	10-10
11-1	Components of the Color Mixing Widget (HLS Color Model) .	11-3
11-2	Components of the Color Mixing Widget (RGB Color Model)	11-4
11-3	Labels in the Color Mixing Widget (HLS Color Model) _____	11-12
11-4	Labels in the Color Mixing Widget (RGB Color Model) _____	11-13
12-1	Sample XUI Toolkit Help Widget _____	12-2
13-1	Edit Menu _____	13-4
14-1	DEC WM Hints Data Structure (VAX Binding) _____	14-6
14-2	WM Decoration Geometry Data Structure (VAX Binding) _____	14-7
14-3	Appearance of an Application Running Under the DECwindows Window Manager _____	14-15
14-4	Customizable Aspects of the Main Application Window _____	14-18
14-5	Informational Attributes Provided by the Window Manager _____	14-22
14-6	Customizable Aspects of Your Application Icon _____	14-24

TABLES

1-1	Summary of XUI Toolkit Widgets _____	1-9
1-2	Widget Size and Position Attributes _____	1-14
1-3	Callback Attributes Supported by the Push Button Widget _____	1-15
2-1	Symbol Definition Files _____	2-3
2-2	Widget Creation Mechanisms _____	2-8
2-3	Standard Arguments Used with Low-Level Routines _____	2-10
2-4	Arguments Used with the High-Level Routine PUSH BUTTON _____	2-14
2-5	Standard Widget Manipulation Routines _____	2-31
3-1	Optional UIL Module Header Clauses _____	3-12
3-2	UIL Compiler Rules for Checking Argument Type and Count _____	3-15
3-3	UIL Value Types _____	3-16
3-4	DRM Routines and Functions _____	3-34
4-1	Main Window Widget Creation Mechanisms _____	4-4
4-2	Widget Attributes Accessible Using the High-Level Routine MAIN WINDOW _____	4-5
4-3	Child Widget Attributes of the Main Window Widget _____	4-6
4-4	Scroll Window Widget Creation Mechanisms _____	4-9
4-5	Widget Attributes Accessible Using the High-Level Routine SCROLL WINDOW _____	4-9
4-6	Child Widget Attributes of the Scroll Window Widget _____	4-10
4-7	Window Widget Creation Mechanisms _____	4-11

red ink

Red ink indicates information that you must enter from the keyboard or a screen object that you must choose or click on.

For online versions of the book, user input is shown in **bold**.

boldface text

Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.

Boldface text is also used to show user input in online versions of the book.

UPPERCASE TEXT

Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.

numbers

Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radices—binary, octal, or hexadecimal—are explicitly indicated.

1

Overview of the XUI Toolkit

This chapter provides the following:

- An overview of the XUI Toolkit components
- An overview of basic XUI Toolkit programming concepts
- A list of the widgets in the XUI Toolkit

1.1

Overview of XUI Toolkit Components

The XUI Toolkit is a set of run-time routines and application development tools you can use to create application programs that implement the user interface techniques and appearance guidelines specified in the *XUI Style Guide*.

Using the XUI Toolkit, you can:

- Open a connection to a display device (workstation)
- Create windows on the display
- Perform output operations to windows
- Receive notification of pointer or keyboard input through windows

The XUI Toolkit consists of the following components:

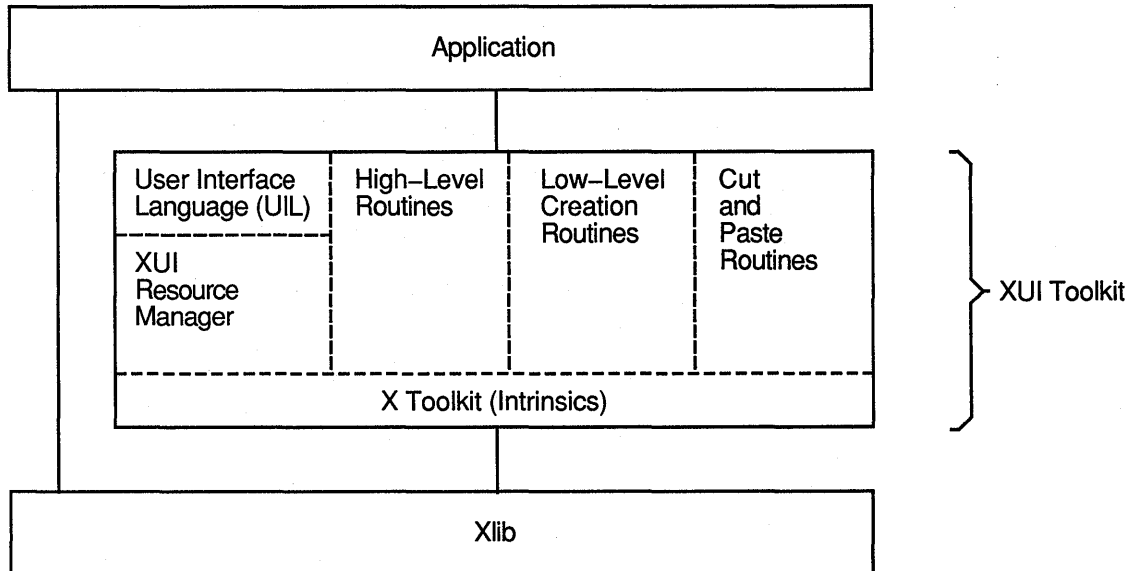
- A set of user interface objects, called **widgets**, with run-time routines to create them
- A set of run-time routines to manipulate the widgets, called X Toolkit intrinsics
- A set of cut and paste routines to copy data between applications
- A pair of application development tools, called the User Interface Language (UIL) and the XUI Resource Manager (DRM)

Figure 1-1 illustrates the components of the XUI Toolkit and its relationship to other layers of the XUI architecture. The following sections describe each component of the XUI Toolkit.

Overview of the XUI Toolkit

1.1 Overview of XUI Toolkit Components

Figure 1-1 XUI Layered Architecture



ZK-0084A-GE

1.1.1 User Interface Objects

The XUI Toolkit provides a set of user interface objects including menus, push buttons, and scroll bars. These objects, called widgets, are the building blocks of the user interface of an XUI application.

An XUI Toolkit widget is made up of a window packaged with input and output capabilities. Some widgets display information, such as text or graphics. Others are merely containers for other widgets. Some widgets are output-only and do not react to pointer or keyboard input. Others change their display in response to input and can invoke functions that an application has attached to them. Table 1-1, in Section 1.3, lists all the widgets provided by the XUI Toolkit.

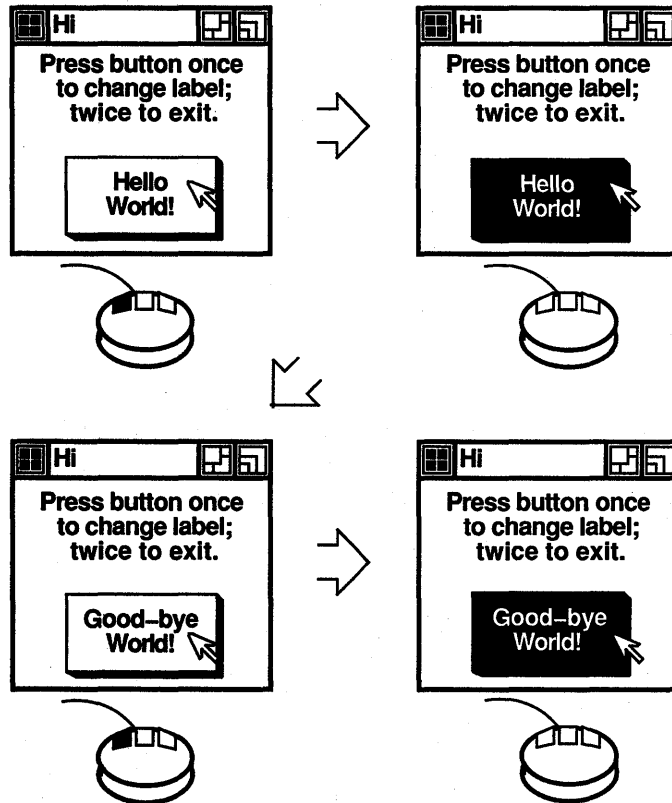
Each widget supports a set of attributes, such as width, height, font, color, and border width, that you can use to customize widget appearance and function. The XUI Toolkit assigns default values to widget attributes to create widgets that conform to the recommendations of the *XUI Style Guide*. Section 1.4 describes these attributes.

The XUI Toolkit provides two versions of some widgets. These widgets have variants, called **gadgets**, that have the same general appearance as their widget counterparts but have restricted capabilities. Gadgets use fewer system resources and can offer improved application performance. For example, gadgets do not have an associated window, thus eliminating the processing involved with creating a window. However, gadgets do not

Overview of the XUI Toolkit

1.2 Programming Concepts

Figure 1-2 Hello World! Application User Interface



ZK-0201A-GE

The source files for the Hello World! sample application are included in the examples directory (DECW\$EXAMPLES:). To become familiar with a basic VMS DECwindows application, run the sample application on your workstation. To do this, copy the source files into your own directory, as follows:

```
$ COPY DECW$EXAMPLES:HELLOWORLD.* *
```

Use the UIL compiler to compile the UIL module that defines the user interface of the Hello World! application. (You must define the UIL include file logical before invoking the compiler.) Then compile the Hello World! C language program and link it with the XUI Toolkit shareable image. The following summarizes this procedure:

```
$ DEFINE UIL$INCLUDE DECW$INCLUDE
```

```
$ UIL HELLOWORLD.UIL
```

```
$ CC HELLOWORLD.C
```

```
$ LINK/NODEB HELLOWORLD,SYSS$INPUT/OPT  
SYSS$LIBRARY:DECW$DWTTLIBSHR/SHARE
```

```
Ctrl/Z
```

```
$ RUN HELLOWORLD
```

Overview of the XUI Toolkit

1.2 Programming Concepts

The XUI Toolkit includes a second example program called DECburger. The DECburger sample application implements an order-entry system for a fictitious fast food restaurant. In DECburger, the user interface is made up of dozens of widgets (and gadgets). (Figure 1-4, in Section 1.3, illustrates the user interface of the DECburger sample application.)

DECburger is designed only to illustrate examples of using the widgets and gadgets in the XUI Toolkit. It is not meant as an example of interface design.

The source files for the DECburger sample application are included in the examples directory (DECW\$EXAMPLES:). To run the sample program, copy all the component source files and execute the command procedure using the following commands:

```
$ COPY DECW$EXAMPLES:DECBURGER.* *
$ @DECBURGER.COM
```

1.2.1 Creating the Form of Your Application

You create a user interface for your application by arranging widgets in parent/child relationships. Parent widgets control the behavior and appearance of their children. In turn, their children can have children. This layering of parent/child relationships creates the **application widget hierarchy**. The application widget hierarchy mirrors the window hierarchy maintained by the X Window System. Child widgets are clipped by their parents just as subwindows are clipped by their superiors; that is, the edge of a child widget cannot extend outside the boundaries of its parent.

The XUI Toolkit includes one type of widget, called a **pop-up** widget, that breaks the window hierarchy. Pop-up widgets can extend beyond the boundaries of their parents. The XUI Toolkit includes several menu and dialog box widgets that are pop-up widgets. For more information, see Section 6.1 and Section 7.3.

Not every XUI Toolkit widget can be a parent. Widgets are either **composite widgets** or **primitive widgets**. Composite widgets can be parents or children of other composite widgets; primitive widgets can only be children.

The user interface of the Hello World! application is an example of a simple application widget hierarchy made up of four widgets: an application shell widget, a dialog box widget, a label widget, and a push button widget. (Note that a real application could contain hundreds of widgets.)

At the top of the application widget hierarchy of the Hello World! program is the application shell widget. The application shell widget acts as the mediator between the application program and the workstation environment in which the application runs. Every XUI application must have a shell widget at the top of its application widget hierarchy. Section 2.3.1 provides more information about the application shell widget.

Overview of the XUI Toolkit

1.1 Overview of XUI Toolkit Components

provide access to all the attributes supported by their widget counterparts. For more information about gadgets, see Section 5.1.

To build a user interface using widgets (or gadgets), you must create instances of the widgets in your application program. When you create a widget, you specify its parent/child relationship, its initial appearance, and other characteristics of the widget by assigning values to widget attributes. To create widgets and determine widget attributes, the XUI Toolkit provides two sets of run-time routines, called **low-level** and **high-level routines**.

Low-level widget creation routines provide access to the complete set of attributes supported by a widget. Using these routines, you assign values to widget attributes in a data structure called an **argument list**. You then pass this argument list to the low-level routine. Section 2.4.1 describes how to build a user interface using low-level routines.

High-level widget creation routines provide a more convenient way to create widgets. Instead of assigning values to widget attributes in an argument list, you pass the values of widget attributes as arguments to the high-level routines. However, high-level routines provide access to only a subset of a widget's attributes at widget creation time. High-level routines specify only the most commonly used widget attributes as arguments. Section 2.4.2 describes how to build a user interface using high-level routines.

Note that you always can use the widget manipulation routines (described in Section 2.9) to access the complete set of widget attributes after a widget has been created. However, it is more efficient to assign values to widget attributes when you create the widget. For this reason, choose the creation routine that provides access to the widget attributes you need to set. The *VMS DECwindows Toolkit Routines Reference Manual* provides complete information about XUI Toolkit high- and low-level widget creation routines.

The application development tools UIL and DRM provide another way to create the widgets in a user interface. For more information, see Section 1.1.4.

1.1.2 X Toolkit Routines

X Toolkit routines, called **intrinsic**s, let you manipulate widgets at run time. The X Toolkit is a standard public domain routine library layered on the X Window System, Version 11.

Intrinsics are the basis of every XUI application. You use intrinsics to do the following:

- Initialize the XUI Toolkit
- Map and unmap widgets to the screen
- Process input from an application end user

Section 2.9.1 provides more information about intrinsics.

Overview of the XUI Toolkit

1.1 Overview of XUI Toolkit Components

You can also use intrinsics to build your own widgets. Appendix D provides more information about this topic.

1.1.3 Cut and Paste Routines

The cut and paste routines provided by the XUI Toolkit are a set of run-time routines you can use to copy data to or from applications. Chapter 13 describes the cut and paste routines.

1.1.4 Application Development Tools

The XUI Toolkit includes two closely related application development tools: the User Interface Language (UIL) and the XUI Resource Manager (DRM).

UIL is a user interface definition language. Using UIL, you can specify a user interface in a text file called a **UIL specification file**. You then compile this file using the UIL compiler. At run time, your application retrieves the compiled interface specification, called a **UID file**, using DRM routines. DRM routines enable you to open the UID specification file, retrieve the widget definitions from the file, create the widgets, and build the user interface at run time. Use of DRM run-time routines optimizes initialization and startup (that is, widget creation) for an XUI application. Chapter 3 describes how to define a user interface in a UIL file and how to use the DRM routines to create the user interface at run time.

Using UIL and DRM, you can change the user interface specification without having to recompile or relink your main application program. This feature of UIL and DRM is particularly important for applications developed for international markets. For example, you can create user interfaces in several languages for a single application.

When you define widgets in a UIL specification file, you can access the complete set of widget attributes. The UIL compiler checks that the values you assign to attributes are of the data type expected by the widget. High-level and low-level widget creation routines do not perform any type-checking on attribute values.

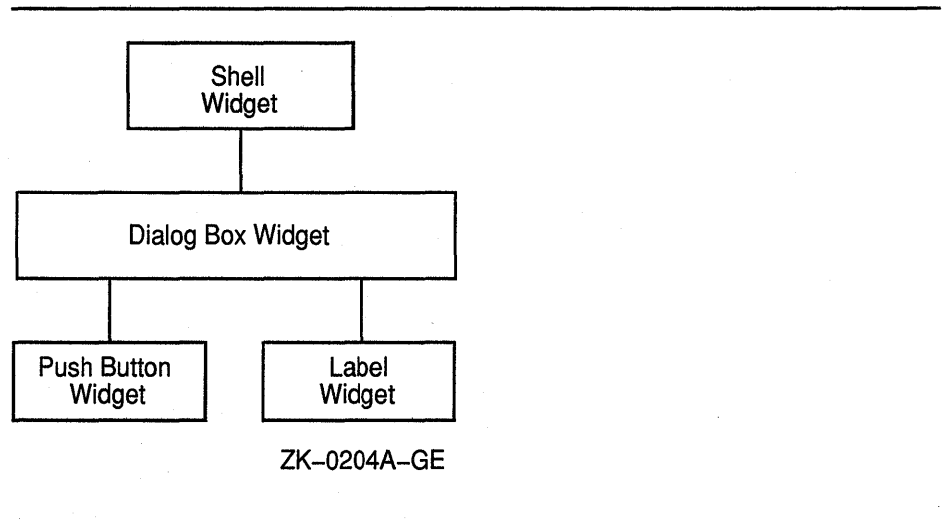
1.2 Programming Concepts

The fundamental concept of programming with the XUI Toolkit is the separation of form and function. Using the XUI Toolkit, you can consider the form your application takes, its user interface, separately from the routines that implement the functions of your application.

For example, the form of the sample Hello World! application is a push button widget containing the text string "Hello World!". The function of the Hello World! application is to change the wording of the text string in the push button widget to "Goodbye World!". Figure 1–2 shows the user interface of the Hello World! application as it initially appears and as it changes when a user interacts with the interface.

The main widget of the Hello World! application is a dialog box widget. This widget is the child of the application shell widget (an application shell widget can only have one child). The dialog box widget, a composite widget, is the parent of a push button widget and a label widget. The label and push button widgets are children of the dialog box widget. The label and push button widgets are examples of primitive widgets; they do not support children. Figure 1-3 illustrates the widget hierarchy formed by the user interface of the Hello World! application.

Figure 1-3 Application Widget Hierarchy of the Hello World! Application



Note that the application widget hierarchy should not be confused with the widget class hierarchy. The application widget hierarchy defines the parent/child relationship of widgets in a user interface. The widget class hierarchy defines the subclass/superclass relationship of the widgets in the XUI Toolkit. The widget class hierarchy determines which attributes a widget inherits from its superclass and which attributes are unique to a particular widget class. For more information about widget classes and the widget class hierarchy, see Appendix D and the *VMS DECwindows Toolkit Routines Reference Manual*.

1.2.2 Associating Function with Form

When a user invokes a VMS DECwindows application program, the initial user interface of the application appears on the display. The application then waits in an infinite loop for the user to interact with its interface. Applications running in the VMS DECwindows environment perform their functions only in response to user interaction with the interface.

When a user of your application interacts with a widget in its interface using a pointing device, such as a mouse or the keyboard, the user action causes a change in the state of the widget. Each widget supports a specific set of such changes in its state that cause it to notify an application. This flow of data from the interface to the application at run time is accomplished through the **callback mechanism**. The callback mechanism

Overview of the XUI Toolkit

1.2 Programming Concepts

provides a one-way path of communication from the interface to the application. This is the primary means an application has of getting input from its interface.

A widget can define one or more callbacks depending on how many changes in its state it is willing to communicate. Each particular set of user actions that triggers a callback is called a **reason**. When a change in state in the widget triggers a callback, your application executes the routine you have associated with the widget. This routine is called a **callback routine**. In this way, you associate the routines that implement the functions of your application with the widgets that make up the user interface of your application. You can associate more than one callback routine with a single callback reason. When there is more than one callback routine, the routines are executed in the order in which you specify them.

For example, one callback reason supported by the push button widget is the **activate** reason. This callback occurs when a user clicks MB1 on the push button widget. The Hello World! application associates its function with the **activate** callback reason. (The *VMS DECwindows Toolkit Routines Reference Manual* lists the callback reasons supported by each widget.)

Note that reasons are not actions such as MB1 up; they are more abstract concepts such as “activate.” The X Window System, on which the XUI Toolkit is based, defines an action such as MB1 up that occurs in a window as an **event**. The server is responsible for noting when an event occurs in a window. An application that uses XUI Toolkit widgets need not be concerned with events. XUI Toolkit widgets automatically notify applications when the event or sequence of events the widget defines as a reason occurs. For example, the push button widget defines the MB1 down/MB1 up sequence of events as the **activate** callback reason.

1.3 Widgets in the XUI Toolkit

The XUI Toolkit contains three types of widgets:

- Input/output widgets

These widgets provide the basic input and output capabilities of a user interface, such as displaying text or graphics, allowing text editing, and enabling a user to input values to your application. The widgets that provide these functions are the label, push button, toggle button, scale, scroll bar, and simple text widgets.

- Container widgets

These widgets act as containers for other widgets. You use these widgets to gather together the widgets that provide access to the functions of your application. The widgets that provide these functions include the dialog box, attached dialog box, and main window widgets. The XUI Toolkit includes some container widgets that are preconfigured to perform commonly needed functions such as presenting caution messages.

Overview of the XUI Toolkit

1.3 Widgets in the XUI Toolkit

- Choice widgets

These widgets present choices to the user of your application. The widgets that provide these functions include the menu and list box widgets.

Table 1–1 lists all the widgets in the XUI Toolkit.

Table 1–1 Summary of XUI Toolkit Widgets

Widget	Function
Input/Output Widgets	
Compound string text	Allows text to be entered and edited in multiple characters sets and writing directions.
Label	A rectangle containing read-only text or graphics.
Separator	A dotted line used to graphically set off areas of a user interface.
Push button	A label widget with input capabilities. Used to invoke an immediate action when selected.
Toggle button	A label widget with input capabilities. Maintains state information such as “on” or “off.” Usually contains a graphical indicator that indicates its current state.
Scale	An elongated rectangle that graphically represents a range of values and is sensitive to user input. Users can select a value within the range by moving a slider or by clicking MB1 within the scale.
Scroll bar	A widget designed to allow users to input information relating to scrolling a work area. A scroll bar widget contains an elongated rectangle that graphically represents a range of values and is sensitive to user input. Users can select a value within the range by moving the slider that overlays the scroll region or by clicking a mouse button within the scroll region. The scroll bar widget also contains two arrow-shaped buttons that implement the stepping functions.
Simple text	Allows text to be entered and edited.

(continued on next page)

Overview of the XUI Toolkit

1.3 Widgets in the XUI Toolkit

Table 1–1 (Cont.) Summary of XUI Toolkit Widgets

Widget	Function
Container Widgets	
Dialog box	A box into which you can place other widgets. You can use dialog boxes to solicit information from or present information to a user.
Attached dialog box	A box into which you can place other widgets. Note that, in an attached dialog box, you specify the relative position of the child widgets instead of specifying fixed positions. When a user resizes an attached dialog box, the child widgets it contains move and resize to maintain the original layout of the box.
Pop-up dialog box	A variant of the dialog box that does not get clipped by its parent.
Pop-up attached dialog box	A variant of the attached dialog box that does not get clipped by its parent.
Message box	A type of dialog box that contains predefined child widgets that allow you to display a message to the user.
Caution box	A version of the message box widget configured to present a warning message to the user.
Work-in-progress box	A version of the message box widget configured to present a “Work in Progress” message.
Selection box	A type of dialog box widget that contains predefined child widgets that allow you to present a choice to the user.
File selection	A special type of selection box widget that queries the user for a file specification.
Main window	A tiling window that can contain a menu bar, scroll bars, a command window, and a work area.
Command window	A window that contains a text entry field that allows users to enter commands on a command line. This widget includes a visible display of command history.
Scroll window	A convenience widget that automatically sizes the slider on the scroll bars used with the window.
Window	An empty rectangle in which you can perform graphics operations. The window widget is the only XUI Toolkit widget that supports graphics operations.
Help	A widget that presents the user of an application with information about a chosen topic.

(continued on next page)

Overview of the XUI Toolkit

1.3 Widgets in the XUI Toolkit

Table 1–1 (Cont.) Summary of XUI Toolkit Widgets

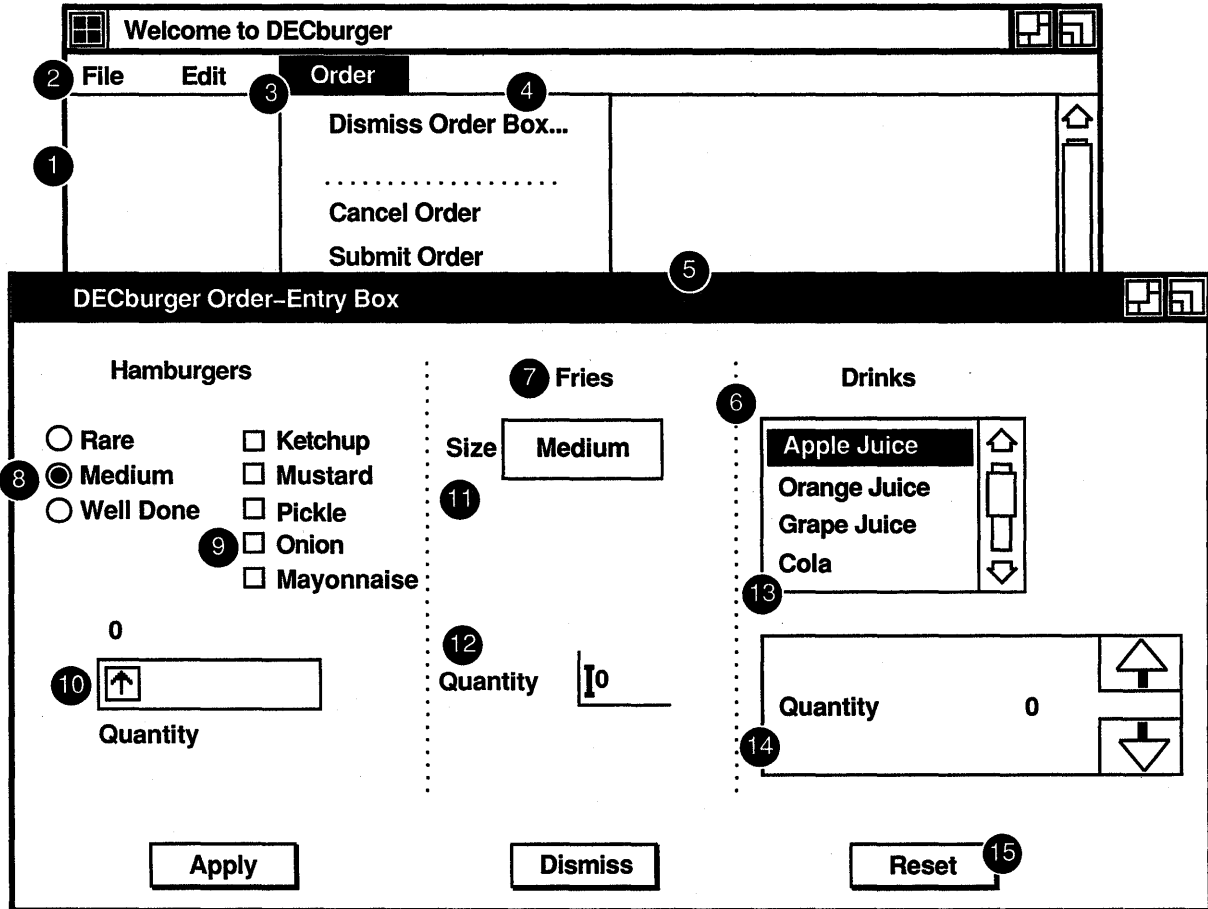
Widget	Function
Choice Widgets	
Color mixing	A pop-up dialog box widget that enables users to define colors and provides users immediate feedback by displaying the colors they define.
Work area menu	A rectangle containing menu items. This is the generic menu widget.
Menu bar	A type of menu widget in which the menu items cause a pull-down menu to appear on the display when selected.
Option menu	A type of menu widget that contains a descriptive text label and a display of the current selection. The actual menu containing the menu items, which is a pull-down menu, appears on the display only when the option menu is activated by the user.
Pop-up menu	A menu that appears on the display when the user presses MB2; a pop-up menu can extend beyond the borders of its parent.
Pull-down menu entry	A button-like widget that causes a pull-down menu to appear.
Pull-down menu	A menu that appears on the display when a user presses MB1 or MB2; a pull-down menu can extend beyond the borders of its parent.
Radio box	A type of work area menu in which a list of choices is presented, only one of which can be selected at any one time.
List box	A rectangle containing a list of choices. List boxes are typically used to present long lists of items. Only a portion of the list is visible in the list box at any time. The list box widget contains a scroll bar that enables users to view the complete item list.

To illustrate these widgets, Figure 1–4 shows the DECBurger user interface.

Overview of the XUI Toolkit

1.3 Widgets in the XUI Toolkit

Figure 1-4 DECburger User Interface



ZK-0136A-GE

- 1 DECburger uses a **main window widget** as the base of the application. The main window widget enables the DECburger application to present some of its basic functions, such as placing an order, as items in a menu bar widget.
- 2 The DECburger **menu bar widget** contains three menus: File, Edit, and Order.
- 3 Each item in the DECburger menu bar widget is a **pull-down menu entry widget**. When the user selects one of the menus in the menu bar widget, a pull-down menu widget appears on the screen. The pull-down menu widget disappears when the user releases MB1. In the figure, the pull-down menu widget associated with the Order menu in the menu bar widget is illustrated as if a user had selected that menu. The pull-down menu widget itself is described in 4.

Overview of the XUI Toolkit

1.3 Widgets in the XUI Toolkit

- ④ The **pull-down menu widget** displayed is the Order pull-down menu widget DECburger uses when the order box is already displayed. The contents of this menu vary depending on whether the order-entry box is visible.
- ⑤ The DECburger order-entry box is a **pop-up dialog box widget**. Pop-up widgets may extend beyond the boundaries of their parent widgets.
- ⑥ DECburger uses a **separator gadget** to draw the vertical dotted lines that mark the boundary of each section of the order-entry box.
- ⑦ To distinguish each section of the order-entry box, DECburger includes a descriptive text label at the top of each section. Each of these text labels is a **label gadget**.
- ⑧ DECburger uses a **radio box widget** to present a list of choices from which the user can choose only one item at a time. Each item in the radio box widget is implemented by a **toggle button gadget**.
- ⑨ To present a list of choices from which the user can select any number of items, DECburger uses a **work area menu widget**. Each item in the menu is a **toggle button gadget**.
- ⑩ To solicit quantity information, DECburger uses the **scale widget**. Because scale widgets graphically present a range of values, they prevent users from entering an incorrect value.
- ⑪ DECburger uses an **option menu widget** to present a list of choices from which only one item can be selected at a time. Each item in the option menu widget is a **push button gadget**. As with the pull-down menu widget, the option menu only appears on the display when the user presses MB1. In this way, the list of items does not take up any display space until it is invoked. The option menu widget always displays its current selection.
- ⑫ DECburger uses a **simple text widget** to handle another quantity choice. The simple text widget enables the user to enter text from the keyboard.
- ⑬ To present a long list of choices, DECburger uses the **list box widget**. Only a portion of the entire list of items is visible in the list box as it appears on the display. Users must use the **scroll bar widget** included in the list box widget to view the complete list of items. List box widgets can be configured to allow users to select more than one item at a time.
- ⑭ DECburger uses an **attached dialog box widget** to implement drink quantity selection. The attached dialog box widget includes two **push button widgets** with pixmap labels. The “up arrow” push button increases the drink quantity; the “down arrow” push button decreases the drink quantity. Note the use of push button widgets instead of gadgets. You cannot use pixmap labels with push button gadgets. The attached dialog box widget also includes two **label gadgets** to display descriptive text and to present the current value selected by the user.

Overview of the XUI Toolkit

1.3 Widgets in the XUI Toolkit

- ⑬ DECburger uses a horizontally oriented **work area menu widget** containing three **push button widgets** to implement the Apply, Dismiss, and Reset functions. Note the use of push button widgets instead of gadgets to allow DECburger to specify a larger font size to emphasize these important functions. You cannot specify the font in a gadget; gadgets use the font specified in their parent. (The figure does not represent the actual font used in these buttons. To see this attribute, run the DECburger application.)

1.4 Widget Attributes

Every XUI Toolkit widget supports a set of attributes you can use to customize aspects of its appearance and function. A subset of these widget attributes is supported by every XUI Toolkit widget. These are called **common widget attributes**. In addition, most widgets support their own unique attributes. The *VMS DECwindows Toolkit Routines Reference Manual* describes the complete set of attributes that each widget supports.

All widgets support the following basic types of attributes:

- Size and position attributes (geometry management)
- Appearance attributes
- Callback attributes

1.4.1 Size and Position Attributes

All widgets support size and position attributes. Table 1–2 lists these attributes.

Table 1–2 Widget Size and Position Attributes

Attribute	Description
width	Specifies the width of the widget in pixels
height	Specifies the height of the widget in pixels
x	Specifies the x-coordinate of the upper left corner of the widget
y	Specifies the y-coordinate of the upper left corner of the widget

Note that, while you can specify the size and position of a widget using these attributes, for many widgets it is preferable to let the widget define its own size and position in the context in which it is used. The size and position of a widget is controlled by its parent. A child can request to be a certain size, but its parent makes the final decision. Parent widgets must weigh the sizing and positioning needs of their other children. In addition, parent widgets are children themselves and must negotiate their space requirements with their parent. This negotiation between parent and child for display space is called **geometry management**.

1.4.2 Appearance Attributes

All XUI Toolkit widgets support attributes that specify aspects of their appearance. Many of these attributes are unique to each widget. For example, the push button widget can appear on the display with a shadow to give a three-dimensional impression. However, you can create push buttons without shadows by setting the push button widget **shadow** attribute to false.

If you do not set an appearance attribute of a widget, the XUI Toolkit uses a default value. The default values for widget attributes create widgets that conform to the recommendations of the *XUI Style Guide*.

1.4.3 Callback Attributes

All XUI Toolkit widgets support attributes that let you associate callback routines with their callback reasons. For example, Table 1–3 lists the four callback attributes supported by the push button widget.

Table 1–3 Callback Attributes Supported by the Push Button Widget

activate_callback	Callback performed when a user clicks MB1 inside the push button widget
arm_callback	Callback performed when a user holds down MB1 inside the push button widget
disarm_callback	Callback performed when a user moves the pointer cursor off the push button widget without releasing MB1
help_callback	Callback performed when a user presses the Help key and clicks MB1 in the push button widget

1.4.4 Assigning Values to Widget Attributes

When you create a widget, the XUI Toolkit determines the initial settings of widget attributes from the following sources, checked in order:

- 1 The argument list supplied with the creation routine
- 2 The widget attribute database
- 3 The default values contained in the widget

The XUI Toolkit first checks the argument list for attribute values. You assign values to widget attributes when you create the widget using high-level routines, low-level routines, or UIL/DRM. (See Section 2.4 for more information about using these widget creation mechanisms.) If you have specified any attribute values in an argument list, the XUI Toolkit assigns this value to the widget when it creates it.

For any attributes to which you do not assign values, the XUI Toolkit retrieves a default value from a database of attribute values.

Overview of the XUI Toolkit

1.4 Widget Attributes

If the XUI Toolkit cannot find a value for an attribute in an argument list or an attribute database, the default value contained in the widget itself is used. Each widget contains a default value for every attribute it supports.

2

Creating a VMS DECwindows Application

This chapter describes how to create an application using the XUI Toolkit. The chapter includes information about the following:

- XUI Toolkit symbol definition files
- Initializing the XUI Toolkit
- Creating the widgets in the user interface
- Managing the widgets in the user interface
- Realizing the widgets in the user interface
- Entering the main processing loop
- Creating a callback routine
- Manipulating the interface at run time

This chapter also includes complete listings for three versions of the Hello World! sample application. Each version illustrates a different method for creating the widgets in the interface.

2.1

Overview of a VMS DECwindows Application

A typical VMS DECwindows application consists of three sections:

- Initial setup of the user interface
- Main input loop
- Callback routines

In the first section, you create the widgets that make up the user interface and make them appear on the display. In this section, you must perform the following steps:

- Initialize the XUI Toolkit
- Create the widgets used in the interface
- Manage the widgets
- Realize the widgets to make them appear on the display

In the second section, your application enters an infinite loop in which it waits for input from a user. When the event or sequence of events the widget has defined as a reason occurs, the widget notifies the application using the callback mechanism. Your application responds to this user interaction by executing a callback routine.

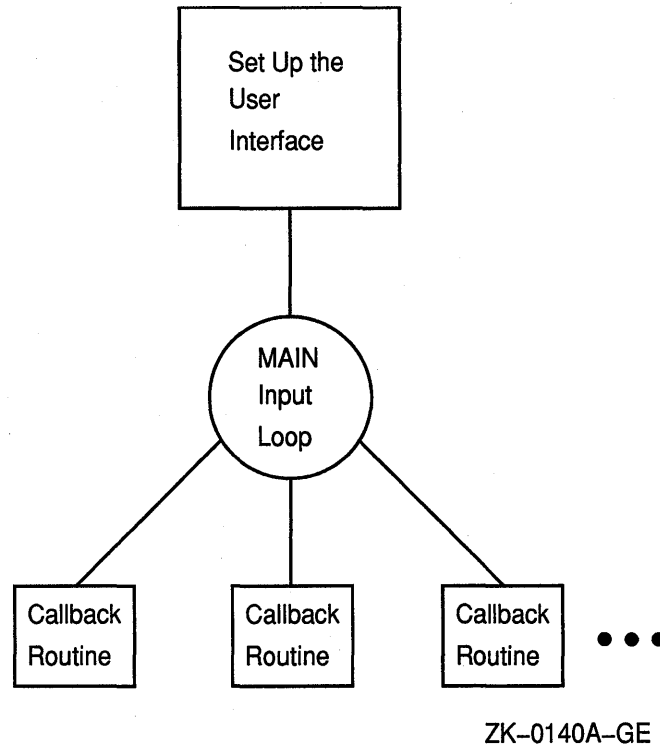
The last section of your application contains the callback routines that implement the functions of your application.

Creating a VMS DECwindows Application

2.1 Overview of a VMS DECwindows Application

Figure 2-1 illustrates the structure of a typical VMS DECwindows application.

Figure 2-1 Structure of an XUI Application



The following sections describe the components of a VMS DECwindows application and illustrate this structure by creating the Hello World! application, introduced in Chapter 1.

2.2 Symbol Definition Files

Before you start setting up the user interface, you must include the XUI Toolkit symbol definition file in your application. The XUI Toolkit routines are available in the VAX binding and the MIT C binding. Use the symbol definition file associated with the language and binding you are using to write your application. Table 2-1 shows the symbol definition files available for the VAX and MIT C bindings. The symbol definition files for the VAX binding reside in `SYS$LIBRARY:`. The symbol definition files for the MIT C binding reside in the `DECW$INCLUDE:` directory.

Creating a VMS DECwindows Application

2.2 Symbol Definition Files

Table 2-1 Symbol Definition Files

File Specification	Description
MIT C Binding	
DwtAppl.h	Contains symbol definitions (constants for commonly used arguments, for example) of interest to application developers
DwtWidget.h	Contains symbol definitions of interest to programmers who will be building their own widgets
VAX Binding	
DECW\$DWTDEF ¹	Contains symbol definitions (constants for commonly used arguments, for example) of interest to application developers
DECW\$DWTWIDGETDEF ¹	Contains symbol definitions of interest to programmers who will be building their own widgets

¹The file type for these files depends on the language. There is a symbol definition file available for several languages (including VAX BASIC, VAX Pascal, VAX BLISS, VAX Ada, VAX PL/1, VAX MACRO, VAX C, and VAX FORTRAN).

The examples used in this chapter build the Hello World! application using the C language with the MIT C binding. In Example 2-1, the Hello World! application includes the symbol definition file. For more information about the symbol definition files used with other languages and examples of Hello World! written using the VAX binding, see Appendix B.

Example 2-1 Including the XUI Toolkit Symbol Definition File in an Application

```
①#include <decw$include/DwtAppl.h>
②static void helloworld_button_activate();
static DwtCallback callback_arg[2];
/***** Main Program *****/
③int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    .
    .
    .
}
```

- ① In this statement, the Hello World! application includes the XUI Toolkit symbol definition file.
- ② These declarations are used by the callback mechanism. Later sections describe their use.
- ③ This statement is the required starting point for a C program.

Creating a VMS DECwindows Application

2.3 Initializing the XUI Toolkit

2.3 Initializing the XUI Toolkit

To initialize the XUI Toolkit, use the INITIALIZE intrinsic routine. This routine performs three essential startup functions:

- Establishes the connection between the application program and the server
- Initializes internal XUI Toolkit data structures
- Creates the application shell widget

The INITIALIZE routine takes the following arguments:

- A name you assign to the application, passed as a text string
- A class name you assign to the application, passed as a text string
- An array of options that instruct the application how to parse the command line
- The number of command line option instructions
- The number of command line arguments passed at application startup
- An array of command line arguments passed as text strings

The name you assign to your application appears in the title bar of your main window. Example 2–2 shows the initialization of the XUI Toolkit in the Hello World! application.

Example 2–2 Initializing the XUI Toolkit

```

.  

.  

①Widget toplevel, helloworldmain, button, label;  

   Arg arglist[5];  

②toplevel = XtInitialize( "Hi","helloworldclass",NULL, 0, &argc, argv);  

   XtSetArg( arglist[0], XtNallowShellResize, TRUE);  

③XtSetValues( toplevel, arglist, 1);  

.  

.  

.
```

- ① This statement creates variables to hold the identifiers of the widgets used in the Hello World! application. The variable named *toplevel* will hold the widget identifier returned by the INITIALIZE intrinsic routine.
- ② The Hello World! application calls the intrinsic routine INITIALIZE to initialize the XUI Toolkit. The Hello World! application names the application with the text string "Hi". This text will appear in the title bar of the application. The class name of the application is the text string "helloworldclass". The Hello World! application does not pass any command line option instructions or command line

Creating a VMS DECwindows Application

2.3 Initializing the XUI Toolkit

arguments. The INITIALIZE intrinsic routine returns the identifier of the application shell widget in the variable *toplevel*.

- ③ After creating the shell widget, the Hello World! application sets one of the attributes of the shell widget using the intrinsic routine SET VALUES. The attribute, named XtNallowShellResize, is set to true. This enables the application shell widget to change its size if the child of the shell widget requests a size change. The attribute is assigned a value in an argument list. See Section 2.4.1.2 for information about creating argument lists.

When you initialize the XUI Toolkit, you obtain an **application context** for your application. An application context is an internal data structure in which the XUI Toolkit maintains information about the state of your application. For example, the XUI Toolkit stores the list of displays to which your application has open connections in an application context. This structure also contains the list of work procedures you register. (For information about work procedures, see Section 2.8.4.) Every application using the XUI Toolkit has an application context.

The INITIALIZE intrinsic routine creates a default application context for your application. However, you can also explicitly create one for your application by calling the CREATE APPLICATION CONTEXT intrinsic routine. If you wish to create your own application context, you must use the TOOLKIT INITIALIZE intrinsic routine to initialize the XUI Toolkit, instead of the INITIALIZE intrinsic routine, and you must explicitly open a connection to a display by calling the OPEN DISPLAY intrinsic routine. In addition, you must create the application shell widget at the top of your application widget hierarchy by calling the APPLICATION CREATE SHELL intrinsic routine. The INITIALIZE intrinsic routine performs all these tasks for you.

Example 2-3 shows the initialization of the XUI Toolkit in a version of the the Hello World! application that creates its own application context.

Example 2-3 Creating Your Own Application Context

```
Widget toplevel, helloworldmain, button, label;
Arg arglist[5];
①XtAppContext context;
②Display *display;
③XtToolkitInitialize();
④context = XtCreateApplicationContext();
⑤display = XtOpenDisplay( context, "mynode::0", "Hi", "testclass",
                        NULL, 0, &argc, &argv );
XtSetArg( arglist[ac], XtNallowShellResize, TRUE ); ac++;
⑥toplevel = XtAppCreateShell( "Hi", "helloworldclass",
                             applicationShellWidgetClass, display, arglist, ac);
.
.
.
```

- ① Declaration of an application context.

Creating a VMS DECwindows Application

2.3 Initializing the XUI Toolkit

- ② Declaration of a variable to hold a pointer to a display.
- ③ The TOOLKIT INITIALIZE intrinsic routine is called to initialize the toolkit. This routine takes no arguments and does not return anything.
- ④ The CREATE APPLICATION CONTEXT intrinsic routine returns an application context. This application context will be used throughout the application as an argument to other intrinsic routines.
- ⑤ The OPEN DISPLAY intrinsic routine is called to open a connection to a display. You pass the application context as the first argument to the routine. The XUI Toolkit maintains a list of open connections to displays in the application context.
- ⑥ The APPLICATION CREATE SHELL intrinsic routine is called to create the application shell widget. This routine returns the identifier of the application shell widget. Note that, when you use the APPLICATION CREATE SHELL intrinsic routine, you can assign values to shell widget attributes when you create the widget. When you create the shell widget with the INITIALIZE intrinsic routine, you must use the SET VALUES intrinsic routine to assign values to shell widget attributes after it has been created.

Note that, if you create your own application context, you must use the version of the intrinsic routines that accepts an application context as an argument. Many intrinsic routines have two interfaces: one that takes an application context as its first argument and one that does not. For example, you would use the ADD TIMEOUT intrinsic routine if you accept the default application context and you would use the APPLICATION ADD TIMEOUT intrinsic routine if you create your own application context. (For another example, see Section 2.7.) The routines without the application context argument use the default application context. You can use either set of routines to create a VMS DECwindows application.

2.3.1 Application Shell Widget

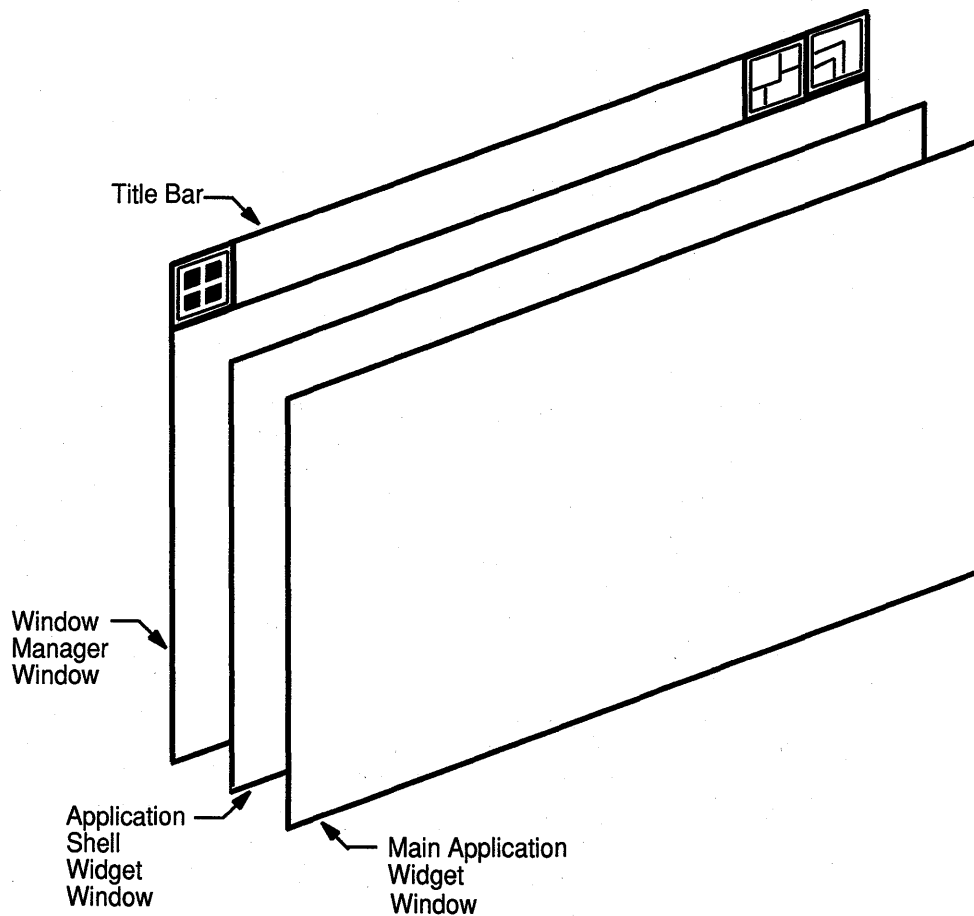
The application shell widget handles the interaction between the application and the outside world; that is, the VMS DECwindows environment in which it runs. When a user moves or resizes an application running in the VMS DECwindows environment, the moving and resizing of the application is controlled by the window manager. Because more than one application can run in the VMS DECwindows environment simultaneously, the window manager controls the sizing and positioning of all applications that appear on a display. (For more information about the window manager, see Chapter 14.)

An application shell widget is a rectangular window that sizes itself to exactly fit its child. The child obscures the application shell widget on the display. A shell widget can have only one child, which is typically the widget at the top of your application widget hierarchy. Figure 2-2 is a graphic representation of the relationship between the application shell widget, the window manager, and your application.

Creating a VMS DECwindows Application

2.3 Initializing the XUI Toolkit

Figure 2-2 Relationship of Shell Widget to Application



ZK-0397A-GE

2.3.2 Using Multiple Shell Widgets

An application should only call the `INITIALIZE` intrinsic routine once. To have multiple windows for your application, you can do one of two things:

- Use a pop-up dialog box
- Create another shell widget

To create another shell widget, use the `APPLICATION CREATE SHELL` or the `CREATE POPUP SHELL` intrinsic routine. The XUI Toolkit defines several types of shell widgets. The application shell widget is typically the top of an application widget hierarchy.

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

2.4 Creating the Widgets in the Interface

After initializing the XUI Toolkit and creating the application shell widget, you must create the widgets that make up the user interface of your application. When you create a widget, you specify three aspects of the widget:

- The parent/child relationship of the widget
- The initial appearance of the widget
- The callback routines associated with the widget

When you create a widget, the XUI Toolkit allocates memory for the internal data structures that define the widget. In addition, the parent of the widget is notified that it is responsible for the widget being created. Every widget in an application has a parent except for shell widgets created by the APPLICATION CREATE SHELL intrinsic routine (or the INITIALIZE intrinsic routine).

To release the memory allocated for a widget, use the intrinsic routine DESTROY WIDGET. Because creating widgets consumes system resources, do not destroy widgets that you may want to reuse in your application. Instead, make widgets appear and disappear from the display by manipulating their parent's list of managed children. For more information about this topic, see Section 2.5.

You can create the widgets that comprise a user interface by calling high-level or low-level widget creation routines in your application program, or you can define the interface in a UIL module. Table 2-2 lists the UIL object type, the high-level creation routine, and the low-level creation routine for each widget in the XUI Toolkit. The following sections describe how to use these mechanisms. Note that, for some widgets, the name of the creation routine is different for the high-level routine and the low-level routine. In addition, the UIL object type for some widgets is different than the high- or low-level creation routine name.

Table 2-2 Widget Creation Mechanisms

Widget	UIL Object Type	High-Level Routine	Low-Level Routine
Attached dialog box	attached_dialog_box	ATTACHED DIALOG BOX	ATTACHED DIALOG BOX CREATE
Caution box	caution_box	CAUTION BOX	CAUTION BOX CREATE
Color mixing	color_mix	No high-level routine	COLOR MIX CREATE
Command window	command_window	COMMAND WINDOW	COMMAND WINDOW CREATE
Compound string text	compound_str_text	CSTEXT	CSTEXT CREATE
Dialog box	dialog_box	DIALOG BOX	DIALOG BOX CREATE
File selection	file_selection	FILE SELECTION	FILE SELECTION CREATE

(continued on next page)

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

Table 2–2 (Cont.) Widget Creation Mechanisms

Widget	UIL Object Type	High-Level Routine	Low-Level Routine
Help	help_box	HELP	HELP CREATE
Label	label	LABEL	LABEL CREATE
List box	list_box	LIST BOX	LIST BOX CREATE
Main window	main_window	MAIN WINDOW	MAIN WINDOW CREATE
Menu bar	menu_bar	MENU BAR	MENU BAR CREATE
Message box	message_box	MESSAGE BOX	MESSAGE BOX CREATE
Option menu	option_menu	OPTION MENU	OPTION MENU CREATE
Pop-up attached dialog box	popup_attached_db	ATTACHED DIALOG BOX ¹	ATTACHED DIALOG BOX POPUP CREATE
Pop-up dialog box	popup_dialog_box	DIALOG BOX ¹	DIALOG BOX POPUP CREATE
Pop-up menu	popup_menu	MENU ²	MENU POPUP CREATE
Pull-down menu entry	pulldown_entry	PULL DOWN MENU ENTRY	PULL DOWN MENU ENTRY CREATE
Pull-down menu	pulldown_menu	MENU ²	MENU PULLDOWN CREATE
Push button	push_button	PUSH BUTTON	PUSH BUTTON CREATE
Radio box	radio_box	RADIO BOX	RADIO BOX CREATE
Scale	scale	SCALE	SCALE CREATE
Scroll bar	scroll_bar	SCROLL BAR	SCROLL BAR CREATE
Scroll window	scroll_window	SCROLL WINDOW	SCROLL WINDOW CREATE
Selection box	selection	SELECTION	SELECTION CREATE
Separator	separator	SEPARATOR	SEPARATOR CREATE
Simple text	simple_text	S TEXT	S TEXT CREATE
Toggle button	toggle_button	TOGGLE BUTTON	TOGGLE BUTTON CREATE
Window	window	WINDOW	WINDOW CREATE
Work area menu	work_area_menu	MENU ²	MENU CREATE
Work-in-progress box	work_in_progress_box	WORK BOX	WORK BOX CREATE

¹The high-level routines DIALOG BOX and ATTACHED DIALOG BOX allow you to specify the pop-up variant in their **style** argument.

²The high-level routine MENU allows you to specify whether the menu is a pop-up, pull-down, or work area menu in its **format** argument.

2.4.1 Using Low-Level Widget Creation Routines

Every XUI Toolkit widget has a corresponding low-level widget creation routine (listed in Table 2–2). By convention, the name of the routine is the name of the widget followed by the word *create*. For example, the low-level routine for the push button widget is called PUSH BUTTON CREATE.

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

All low-level widget creation routines take the same four arguments. Table 2-3 describes these arguments.

Table 2-3 Standard Arguments Used with Low-Level Routines

parent_widget	The widget identifier of the parent widget.
name	A name you assign to the widget.
override_arglist	The address of an argument list containing values for attributes of the widget.
override_argcount	The number of arguments in the argument list. If you do not specify an argument list, this argument must be specified as 0.

2.4.1.1 Using Low-Level Routines to Define the Parent/Child Relationship of a Widget

You use the **parent_widget** argument to define the parent/child relationship of the widget you are creating. Pass the widget identifier of the parent as the value of this low-level routine argument. Note that parent widgets must be created before their children.

In the following example, taken from the Hello World! application, the push button widget is created as the child of the dialog box widget by using the low-level widget creation routine PUSH BUTTON CREATE. The widget identifier of the dialog box widget, *helloworldmain*, is passed as the first argument to the routine.

```
button = DwtPushButtonCreate( helloworldmain, "button", arglist, 4 );
```

2.4.1.2 Using Low-Level Routines to Define the Initial Appearance of a Widget

You define the initial appearance of a widget by assigning values to widget attributes. Each widget supports a set of attributes that controls aspects of its appearance such as width and height. Using low-level routines, you assign values to widget attributes in an argument list. If you assign a value to an attribute that the widget does not support, the widget ignores the value.

An argument list is an array of argument data structures. In each argument data structure, which is defined by the XUI Toolkit, you associate the name of the widget attribute with the value you want assigned to that attribute. The following is the definition of the argument data structure.

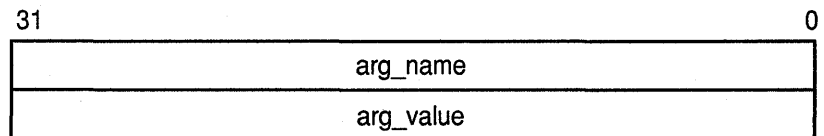
```
typedef struct {  
    char      *name;  
    XtArgVal  value;  
} Arg, *ArgList;
```

Figure 2-3 details the VAX binding definition of this structure.

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

Figure 2-3 Argument Data Structure (VAX Binding)



ZK-0541A-GE

The XUI Toolkit defines the name of each widget attribute as a constant. The *VMS DECwindows Toolkit Routines Reference Manual* lists the complete set of attributes supported by each widget, with their associated constants.

Widget attributes can take a variety of values, such as integers or character strings. The XUI Toolkit defines the value field of this data structure as a longword, named *XtArgVal*. If the attribute value fits into a longword, the *value* field of the structure contains the actual value. If the size of the value exceeds a longword, the *value* field of the structure contains a pointer to the value.

As a convenience, the XUI Toolkit provides a routine you can use to fill in the argument data structures in an argument list. This intrinsic routine SET ARG takes the following three arguments:

- The address of the argument list element
- The name of the widget attribute
- The value being assigned to the attribute

In the following example, taken from the Hello World! application, values for push button widget attributes are specified in an argument list:

```
Arg arglist[5];
.
.
.
XtSetArg(arglist[0], DwtNx, 15);
XtSetArg(arglist[1], DwtNy, 40);
XtSetArg(arglist[2], DwtNactivateCallback, callback_arg);
XtSetArg(arglist[3], DwtNlabel, DwtLatin1String("Hello\nWorld!"));
button = DwtPushButtonCreate(helloworldmain, "button", arglist, 4);
```

The attributes include the x- and y-coordinates that determine the position of the push button widget and the text label the push button widget contains. The argument list, named *arglist*, is declared as an array of argument data structures. The address of the argument list is passed as the third argument to the PUSH BUTTON CREATE routine. (The fourth argument to the PUSH BUTTON CREATE routine is the number of attributes specified in the argument list.)

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

2.4.1.3 Using Low-Level Routines to Associate Callback Routines with a Widget

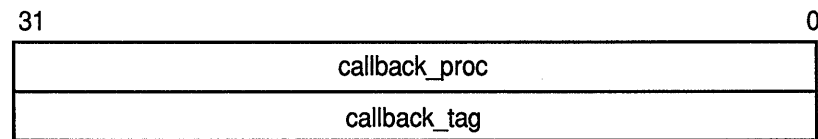
To use a low-level routine to associate a callback routine with a widget, you must pass a callback routine list as the value of a callback attribute. As with other widget attributes, using the low-level routine, you assign the value to the attribute in an argument list (described in Section 2.4.1.2).

A callback routine list is a null-terminated array of callback routine data structures. A callback routine data structure is an XUI Toolkit-defined data structure that pairs the address of the callback routine with any application-specific data you specify. This application-specific data is called a **tag**. The following is the definition of the callback routine data structure:

```
typedef struct {
    VoidProc  proc;
    int       tag;
} DwtCallback, *DwtCallbackPtr;
```

Figure 2-4 is the VAX binding definition of this data structure.

Figure 2-4 Callback Routine Data Structure (VAX Binding)



ZK-0268A-GE

The first field of the callback routine data structure contains the address of the callback routine. The second field of the data structure contains the actual tag value, if it can fit into a longword. If the tag cannot fit into a longword, the tag field contains the address of the tag. A tag can be any data you want to associate with the widget, such as an integer, text string, or data structure. When the widget performs a callback, it passes this data to your application. The XUI Toolkit performs no processing on this data.

The Hello World! application associates the callback routine, named *helloworld_button_activate*, with the push button widget attribute **activate_callback**. Example 2-4 shows how the Hello World! application creates a callback routine list by declaring an array, named *callback_arg*, consisting of two callback routine data structures. The example assigns the address of the callback routine and a tag value to members of the callback routine data structure. Note that you must terminate a callback routine list by assigning a null value to the last callback routine structure.

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

Example 2-4 Creating a User Interface Using Low-Level Routines

```
1 static void helloworld_button_activate();
2 static DwtCallback callback_arg[2];
3
4 Widget toplevel, helloworldmain, button, label;
   Arg arglist[5];
5
6 helloworldmain = DwtDialogBoxCreate (toplevel, "MAINWIN", NULL, 0);
7
8
9 callback_arg[0].proc = helloworld_button_activate;
   callback_arg[0].tag = 0;
   callback_arg[1].proc = NULL;
10
11 XtSetArg (arglist[0], DwtNx, 15);
   XtSetArg (arglist[1], DwtNy, 40);
   XtSetArg (arglist[2], DwtNactivateCallback, callback_arg);
   XtSetArg (arglist[3], DwtNlabel, DwtLatin1String("Hello\nWorld!"));
12
13 button = DwtPushButtonCreate( helloworldmain, "button", arglist, 4);
14
15
16
```

-
- 1 The Hello World! application makes a forward declaration of the callback routine named *helloworld_button_activate* to be able to refer to the routine in a callback routine list.
 - 2 The Hello World! application declares the callback routine list as an array of callback routine data structures. Note that the array contains two elements. All callback routine lists must contain at least two elements because a callback routine list is a null-terminated list. Assign the value null to the last element of the array to signify the end of the list.
 - 3 The Hello World! application creates variables to hold the identifiers of the widgets used in the application.
 - 4 The Hello World! application creates the main widget of its user interface, a dialog box widget, using the low-level routine DIALOG BOX CREATE. Note that the application does not set any widget attributes of the dialog box. The **override_arglist** argument is passed as null.
 - 5 In these three statements, the example assigns values to elements of the callback routine list. Each callback routine data structure contains the address of a callback routine and a tag. The first statement assigns the address of the callback routine used in the Hello World! application as the value of the first member of the data structure. The second statement assigns the tag value to the second member of

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

this data structure. The third statement assigns a null to a callback routine data structure signifying the end of the callback routine list.

- ⑥ These four calls to the intrinsic routine SET ARG create the argument list used to set attributes of the push button widget. In this argument list, the Hello World! application positions the push button widget within its parent by assigning values to its x- and y-coordinates. In addition, the text string "Hello World!" is passed as the label the push button widget will contain. A callback routine is associated with the activate callback.

All text strings that are to appear on the display must be converted to compound strings. The example shows how the text string used as the label in the push button widget is converted into a compound string using the routine LATIN1 STRING. Section 5.6 provides more information about compound strings. (UIL performs this conversion automatically. See Section 3.2.7.3 for more information.)

- ⑦ This statement creates the push button widget using the low-level routine PUSH BUTTON CREATE. In the four standard arguments to the low-level routine, the Hello World! application names the dialog box widget as the parent of the push button widget, assigns the name *button* to the widget, passes the address of the argument list containing attribute values, and passes the number of attributes set in the argument list.

2.4.2 Using High-Level Widget Creation Routines

Every XUI Toolkit widget has a corresponding high-level creation routine (listed in Table 2-2). By convention, the name of the high-level routine is the name of the widget. For example, the high-level routine for creating a push button widget is called PUSH BUTTON.

All high-level widget creation routines take the same first two arguments. These are the widget identifier of the parent widget and the name you assign to the widget. The other arguments vary for each widget because, instead of using an argument list to assign values to widget attributes, the high-level routines accept attribute values as arguments to the routine. As an example, Table 2-4 lists the arguments accepted by the high-level routine used to create a push button widget.

Table 2-4 Arguments Used with the High-Level Routine PUSH BUTTON

parent_widget	The widget identifier of the parent widget
name	A name you assign to the widget
x	The x-coordinate of the upper left corner of the widget
y	The y-coordinate of the upper left corner of the widget

(continued on next page)

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

Table 2-4 (Cont.) Arguments Used with the High-Level Routine PUSH BUTTON

label	The text string to be displayed in the widget
callback	The address of a callback routine list
help_callback	The address of a callback routine list

The *VMS DECwindows Toolkit Routines Reference Manual* describes the arguments supported by each of the high-level routines.

2.4.2.1 Using High-Level Routines to Define the Parent/Child Relationship of a Widget

As with low-level routines, you specify the parent of the widget in the **parent_widget** argument. Pass the widget identifier of the parent as the value of this high-level routine argument. Note that parent widgets must be created before their children.

In the following example, the push button widget in the Hello World! application is created as the child of the dialog box widget by using the high-level widget creation routine PUSH BUTTON. The widget identifier of the dialog box widget, *helloworldmain*, is passed as the first argument to the routine.

```
button = DwtPushButton( helloworldmain, "button", 15, 40,  
                        DwtLatin1String("Hello\nWorld!"), callback_arg, 0);
```

2.4.2.2 Using High-Level Routines to Define the Initial Appearance of a Widget

As with low-level routines, you specify the initial appearance of a widget by assigning values to widget attributes. However, instead of assigning values to widget attributes in an argument list, with a high-level routine you pass the attribute values as arguments to the high-level routine. The high-level routine arguments provide access to the same widget attributes as the argument list used with a low-level routine. For example, the **label** argument of the PUSH BUTTON routine provides access to the same attribute as the **label** attribute used in an argument list.

Note that high-level widget creation routines only provide access to a subset of the attributes supported by the widget at widget creation time. You can use the widget manipulation routines (described in Section 2.9.2) to access the complete set of widget attributes after the widget has been created. However, it is more efficient to set widget attributes when you create the widget.

In the following example, values for push button widget attributes are specified as arguments to the PUSH BUTTON routine. The attributes include the x- and y-coordinates that determine the position of the push button widget in the dialog box widget and the text label the push button widget contains.

```
button = DwtPushButton( helloworldmain, "button", 15, 40,  
                        DwtLatin1String("Hello\nWorld!"), callback_arg, 0);
```

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

2.4.2.3 Using High-Level Routines to Associate Callback Routines with a Widget

To use a high-level routine to associate a callback routine with a widget, you must pass a callback routine list as the value of an argument to the high-level routine. See Section 2.4.1.3 for information about how to create a callback routine list. Each high-level routine includes arguments associated with the callback supported by the widget.

The high-level routine might not define arguments for every callback supported by a widget. In these cases, the callback routine is associated with the callback reason the widget identifies as its main callback. For example, the main callback supported by the push button widget is its **activate** callback reason.

In Example 2-5, the Hello World! application associates the callback routine with the push button widget by passing the address of a callback routine list as an argument to the routine.

Example 2-5 Creating a User Interface Using High-Level Routines

```
.
.
.
1static void helloworld_button_activate();
2static DwtCallback callback_arg[2];
.
.
.
3Widget toplevel, helloworldmain, button, label;
.
.
.
4helloworldmain = DwtDialogBox( toplevel, "MAINWIN", TRUE,0,0,
    DwtLatin1String("Hi"), DwtWorkarea, 0, 0 );
.
.
.
5callback_arg[0].proc = helloworld_button_activate;
  callback_arg[0].tag = 0;
  callback_arg[1].proc = NULL;
.
.
6button = DwtPushButton( helloworldmain, "button", 15, 40,
    DwtLatin1String("Hello\nWorld!"), callback_arg, 0 );
.
.
.
```

-
- ❶ The example makes a forward declaration of the callback routine named *helloworld_button_activate* to be able to refer to the routine in a callback routine list.
 - ❷ The example declares the callback routine list. The callback routine list is declared as an array of callback routine data structures. Note that the array contains two elements. All callback routine lists must contain at least two elements because a callback routine list is a null-terminated list. The last element of the array is always set to null to signify the end of the list.

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

- ③ The Hello World! application creates variables to hold the identifiers of the widgets used in the application. The variable *button* will hold the push button widget identifier.
- ④ The Hello World! application creates the main widget of its user interface, the dialog box widget, using the high-level routine `DIALOG BOX`.
- ⑤ As with the low-level routine, the Hello World! application creates a callback routine list with these three assignment statements. This list is used to associate a callback routine with a widget. Section 2.4.1.3 describes how to create a callback routine list.
- ⑥ This statement creates the push button widget using the high-level routine `PUSH BUTTON`. In the arguments passed to the high-level widget creation routine, the Hello World! application names the parent of the push button widget and assigns the name "button" to the widget. In addition, the Hello World! application passes values for several widget attributes as arguments to the high-level routine. These attributes include values for the x- and y-coordinates of the push button widget, the text string to be contained in the push button widget, and the address of a callback routine list.

As with the low-level widget creation routine, the example shows how the text string used as the label in the push button widget is converted into a compound string using the routine `LATIN1 STRING`. Section 5.6 provides more information about compound strings.

2.4.3 Using UIL and DRM to Create Widgets

Every XUI Toolkit widget has a corresponding UIL object type. By convention, the UIL object type is the name of the widget made into a single word using the underscore character. For example, the UIL object type for the push button widget is `push_button`. Table 2-2 lists the UIL object types for all XUI Toolkit widgets.

As with high- and low-level routines, you can use UIL to define the same three aspects of the widgets in a user interface:

- The parent/child relationship of the widget
- The initial appearance of the widget
- The callback routines associated with the widget

Using UIL, you define these aspects of a widget in a UIL object declaration. A UIL module can contain the object declarations of all the widgets in an interface. You then compile this interface definition using the UIL compiler and store the output in a file called a User Interface Definition (UID) file. At run time, your application calls DRM routines to open the UID file and fetch the compiled interface definition. The DRM routine `FETCH WIDGET` creates the widgets according to the definitions you specify in the UIL module. You can fetch the entire interface with one call to the `FETCH WIDGET` routine.

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

This section provides a brief overview of how to define an interface in a UIL module. For more information about using UIL and DRM, see Chapter 3.

2.4.3.1 Using UIL to Define the Parent/Child Relationship of a Widget

Using UIL, you specify the children of a widget in the object declaration of the parent, instead of specifying the parent of the widget when you create the child (as with the high- and low-level routines). You specify the children by name in the **controls list** section of the parent object declaration.

In the following example from the Hello World! application UIL module, the push button widget, named *helloworld_button*, is specified as the child of the dialog box widget in the controls list of the dialog box widget object declaration. (The label widget, named *helloworld_label*, which is the only other child of the dialog box widget in the Hello World! application, also appears in the controls list.)

```
object
  helloworld_main : dialog_box {
    controls {
      label      helloworld_label;
      push_button helloworld_button;
    };
  };
```

2.4.3.2 Using UIL to Define the Initial Appearance of a Widget

You define the initial appearance of a widget by assigning values to widget attributes in the **arguments list** section of a UIL object declaration. UIL defines a keyword that identifies every widget attribute. For example, the keyword identifying the **label** attribute of a push button widget is **label_label**. UIL provides access to the complete set of widget attributes.

In the following example from the Hello World! application UIL module, values for push button widget attributes are specified in the arguments list of the push button widget object declaration. The attributes include the x- and y-coordinates that specify the position of the push button widget and the text label the push button widget contains.

```
object
  helloworld_button : push_button {
    arguments {
      x = 15;
      y = 40;
      label_label = compound_string('Hello', separate=true) &
                    compound_string('World!');
    };
    callbacks {
      activate = procedure helloworld_button_activate();
    };
  };
```


Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

2.4.3.3 Using UIL to Associate Callbacks with a Widget

You associate callbacks with a widget in the **callbacks list** section of the object declaration. As with the other widget attributes, UIL defines keywords to identify each of the callback attributes supported by a widget. For example, the callback attribute for the **activate** reason supported by the push button widget is identified by the **ACTIVATE** keyword. However, you do not have to create a callback routine list to pass to the callback attribute. In a UIL module, you declare the callback routine by name in the procedure section of a UIL module. You can use this name to refer to the callback routine in the remainder of the UIL module in callbacks lists.

You associate a tag with the callback routine by inserting the value between the parentheses used in the procedure name. (The Hello World! application does not use the tag feature. For an example of this capability, see Section 3.2.6.)

In Example 2-6, the Hello World! application associates the callback routine, named *helloworld_button_activate*, with the push button widget by listing it in the callbacks list section of the object declaration.

Example 2-6 Using UIL to Define a Widget

```
.
.
.
① procedure
    helloworld_button_activate();
.
.
.
② object
③ helloworld_main : dialog_box {
    controls {
④     label      helloworld_label;
        push_button helloworld_button;
    };
.
.
.
object
    helloworld_button : push_button {
⑤     arguments {
        x = 15;
        y = 40;
        label_label = compound_string('Hello', separate=true) &
            compound_string('World!');
    };
⑥     callbacks {
        activate = procedure helloworld_button_activate();
    };
.
.
.
.
```

- ① In this statement, Hello World! declares the callback routine in the procedure section of a UIL module. The name of the callback routine can be used throughout the UIL module. This procedure must also be

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

declared and registered with DRM in your application program (see Example 2-17).

- ② The UIL keyword OBJECT signifies the start of an object declaration.
- ③ This object declaration defines a dialog box widget using the UIL object type dialog_box. The object declaration names the widget *helloworld_main*.
- ④ In the object declaration of the dialog box widget, the parent/child relationship of the push button widget is defined. The controls list section lists the two widgets that are children of the dialog box widget by their object type and the name assigned to them in the UIL module. In the Hello World! application, the push button widget is named *helloworld_button*.
- ⑤ In the object declaration of the push button widget, the initial appearance of the push button widget is defined in the arguments list section. The Hello World! application positions the push button widget by assigning values to the x- and y-coordinates. In addition, the module creates a compound string containing the text "Hello World!" and assigns it as the value of the label attribute of the push button widget. To make the text "Hello World!" appear on two separate lines, the UIL module uses the UIL keyword SEPARATE. The SEPARATE function ensures that a newline character will appear after the word "Hello". The module then creates the word "World!" as a separate compound string and concatenates the two strings.
- ⑥ In the callbacks list section of the object declaration, the Hello World! application associates a callback routine with the **activate_callback** attribute.

At run time, you create the interface defined in the UIL module by using the DRM routine FETCH WIDGET. Example 2-7 shows how the Hello World! application uses DRM routines to open the interface definition file, match the callback routines specified in the UIL module with their addresses in the application program, and fetch the application widget hierarchy.

Creating a VMS DECwindows Application

2.4 Creating the Widgets in the Interface

Example 2-7 Creating the Interface at Run Time Using DRM

```
.  
. .  
①if (DwtOpenHierarchy (1,vec,NULL, &s_DRMHierarchy) != DRMSuccess)  
{  
    printf ("can't open hierarchy");  
}  
②DwtRegisterDRMNames (regvec, regnum);  
③if (DwtFetchWidget (s_DRMHierarchy, "helloworld_main", toplevel,  
                    &helloworldmain, &class) != DRMSuccess)  
    printf("can't fetch interface");  
. .  
.
```

- ① This statement opens the compiled interface definition file using the DRM routine OPEN HIERARCHY.
- ② The example fills in the actual values of symbols used in the interface definition file. For example, this call to the DRM routine REGISTER DRM NAMES resolves references to addresses of callback routines.
- ③ The example retrieves the main application widget, *helloworld_main*. The DRM routine FETCH WIDGET retrieves the widget definitions for all of its children as well. The FETCH WIDGET routine creates all the widgets in the interface.

Procedures declared in a UID file cannot be bound to addresses prior to running the program. This is a function usually done by the linker (which accepts object modules as input). DRM provides a registration facility that accepts the string name and the procedure address and maps the name to the address.

For more information about use of the DRM routines, see Section 3.3.

2.5 Managing the Widgets in the Interface

Once you create the widgets of the user interface, the next step is to manage them. Managing a widget adds the widget to its parent's list of managed children. The parent widget is responsible for the physical layout of all of its children. When you manage a widget, the parent widget calculates its space requirements to accommodate all of its managed children. Managing a widget makes it displayable.

You can manage a single widget at a time using the intrinsic routine MANAGE CHILD, or you can manage multiple children of the same parent at the same time using the intrinsic routine MANAGE CHILDREN. Section 2.5.1 and Section 2.5.2 describe these routines.

You can remove a widget from its parent's list of managed children using the UNMANAGE CHILD intrinsic routine. If the widget and its parent appear on the display, removal causes the child to disappear from the display. Removing a widget from its parent's list of managed children does not destroy the widget. You can make the widget reappear on the display

Creating a VMS DECwindows Application

2.5 Managing the Widgets in the Interface

by managing it again. To remove a group of children of the same parent in a single call, use the UNMANAGE CHILDREN intrinsic routine.

Note that manipulating a parent's list of managed children is a very effective way to make widgets appear, disappear, and then reappear during the execution of your application. Creating, destroying, and then re-creating widgets at run time consumes many more system resources and is less efficient.

To find out if a widget is currently managed, use the intrinsic routine IS MANAGED. This routine takes one argument: the identifier of the widget you are querying about.

Note that when using UIL to define a user interface, you do not have to explicitly manage the widgets. By default, the DRM routine FETCH WIDGET manages every widget it creates. You can override this default by specifying the keyword UNMANAGED in the controls list section. Only the topmost widget in the hierarchy being fetched needs to be managed. For more information, see Section 3.2.8.2.

2.5.1 Managing a Single Child Widget

Use the MANAGE CHILD intrinsic routine to add a single child widget to the set of managed children of its parent. This routine takes one argument: the identifier of the widget being managed. (You specify the parent of the widget when you create it.) Example 2-8 shows how the Hello World! application manages the push button widget.

Example 2-8 Managing a Single Widget

```
.
.
.
callback_arg[0].proc = helloworld_button_activate;
callback_arg[0].tag = 0;
callback_arg[1].proc = NULL;

XtSetArg (arglist[0], DwtNx, 15) ;
XtSetArg (arglist[1], DwtNy, 40);
XtSetArg (arglist[2], DwtNactivateCallback, callback_arg) ;
XtSetArg (arglist[3], DwtNlabel, DwtLatin1String("Hello\nWorld!") ) ;

❶button = DwtPushButtonCreate( helloworldmain, "button", arglist, 4);
❷XtManageChild( button );

.
.
.
```

- ❶ The example program creates the push button widget using the low-level routine PUSH BUTTON CREATE. The parent of the push button widget is specified in the first argument to the routine. The variable *button* receives the widget identifier returned by the creation routine.

Creating a VMS DECwindows Application

2.5 Managing the Widgets in the Interface

- ② The Hello World! application manages the push button widget in this call to the `MANAGE CHILD` intrinsic routine. The variable `button`, the only argument passed to the routine, contains the widget identifier of the push button widget.

2.5.2 Managing Multiple Child Widgets

To manage a group of widgets in a single call, use the `MANAGE CHILDREN` intrinsic routine. This routine takes two arguments: an array of widget identifiers and the number of widget identifiers in the array. All the widgets managed using the `MANAGE CHILDREN` intrinsic routine must be children of the same parent.

After the parent widget has been realized (see Section 2.6), using `MANAGE CHILDREN` to manage multiple children of the same parent is more efficient than making multiple calls to `MANAGE CHILD`. With `MANAGE CHILDREN`, the parent only has to calculate the layout of its children once.

Example 2-9 is a version of the Hello World! application in which the two children of the dialog box widget are managed using the `MANAGE CHILDREN` intrinsic routine. The example creates the interface of the Hello World! application using high-level widget creation routines. In the Hello World! user interface, the dialog box widget has two children: a label widget and a push button widget.

Example 2-9 Managing a Group of Child Widgets

```
.
.
Widget toplevel, helloworldmain;
①WidgetList main_children[2];
②int count = 0;
   Arg arglist[2];
.
.
helloworldmain = DwtDialogBox( toplevel, "MAINWIN", TRUE,0,0,
                               DwtLatin1String("Hi"), DwtWorkarea, 0, 0 );
③main_children[count++] = DwtLabel( helloworldmain, "label", 0, 0,
DwtLatin1String("Press button once\nto change label;\ntwice to exit."), 0 );
   callback_arg[0].proc = helloworld_button_activate;
   callback_arg[0].tag = 0;
   callback_arg[1].proc = NULL;
④main_children[count++] = DwtPushButton( helloworldmain, "button", 15, 40,
                                          DwtLatin1String("Hello\nWorld!"), callback_arg, 0 );
⑤XtManageChildren( main_children, count );
.
.
```

Creating a VMS DECwindows Application

2.5 Managing the Widgets in the Interface

- ❶ To use the intrinsic routine `MANAGE CHILDREN`, you must create an array of widget identifiers. The XUI Toolkit defines a data type, `WidgetList`, that you use for this purpose. The example declares an array, named *main_children*, composed of pointers to widget identifiers.
- ❷ This statement declares a variable that will contain the number of widget identifiers in the array.
- ❸ `Hello World!` creates the label widget child of the dialog box widget using the high-level routine `LABEL`. The first element of the *main_children* array receives the widget identifier returned by this routine.
- ❹ The `Hello World!` application creates the push button widget using the high-level routine `PUSH BUTTON`. The second element of the *main_children* array receives the widget identifier returned by this routine.
- ❺ This version of the `Hello World!` application manages both children of the dialog box widget at the same time by calling the `MANAGE CHILDREN` intrinsic routine. The array of widget identifiers and the number of widgets in the array are passed as arguments to the routine.

2.6 Realizing the Widgets in the Interface

As the last step in setting up a user interface, you make the widgets that you have created and managed appear on a display by **realizing** them. Realizing a widget creates a window for the widget and maps the window to the display. For composite widgets (widgets with children), realizing the widget also creates windows for all of the managed children of the widget and causes these windows to be mapped as well. Thus, you need only realize the widget at the top of the widget hierarchy in a user interface to cause the entire interface to appear on the display.

To realize a widget, use the intrinsic routine `REALIZE WIDGET`. This routine takes one argument: the identifier of the widget being realized.

To find out if a widget is currently realized, use the intrinsic routine `IS REALIZED`. This routine takes one argument: the identifier of the widget you are querying about.

Example 2–10 shows how the complete widget hierarchy of the `Hello World!` application is realized in a single call to `REALIZE WIDGET`. The `Hello World!` application realizes the shell widget, called *toplevel*, returned by the `INITIALIZE` intrinsic routine. By doing this, all the widgets below the top-level widget in the widget hierarchy are realized in one call.

Creating a VMS DECwindows Application

2.6 Realizing the Widgets in the Interface

Example 2-10 Realizing a Widget Hierarchy

```
.
.
Widget toplevel, helloworldmain, button, label;
Arg arglist[5];
/***** Set Up the User Interface *****/
❶toplevel = XtInitialize("Hi","helloworldclass",NULL, 0, &argc, argv);
XtSetArg (arglist[0], XtNallowShellResize, TRUE);
XtSetValues (toplevel, arglist, 1);
.
.
/**** Create and manage the widgets using either ****
**** low-level routines, high-level routines, or UIL. ****/
.
.
❷XtRealizeWidget (toplevel);
/***** End of Set Up *****/
.
.
```

- ❶ In the example, the Hello World! application creates the shell widget using the intrinsic routine INITIALIZE. This routine returns the identifier of the shell widget.
- ❷ After creating and managing the widgets in the interface, the example realizes the widget at the top of the widget hierarchy, *toplevel*, causing the entire widget hierarchy to appear on the display.

2.7 Main Input Loop

After setting up the interface, your application program must wait for input from the user of the application. The widgets in the interface notify your application when a user interacts with them. For example, when a user moves the pointer cursor onto a push button widget and clicks MB1, the widget notifies the application of this action by executing a callback. You can perform any type of processing in response to these callbacks using a callback routine.

To make your application loop while waiting for input, use the intrinsic routine MAIN LOOP. The MAIN LOOP routine takes no arguments. Example 2-11 shows the call to the MAIN LOOP routine used in the Hello World! application.

Creating a VMS DECwindows Application

2.7 Main Input Loop

Example 2-11 Entering the Main Input Loop

```
#include <decw$include/DwtAppl.h>
static void helloworld_button_activate();
static DwtCallback callback_arg[2];
/***** Main Program *****/
int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Widget toplevel, helloworldmain, button, label;
    Arg arglist[5];
/***** Set Up the User Interface *****/
toplevel = XtInitialize( "Hi", "helloworldclass",
                        NULL, 0, &argc, argv);

    XtSetArg( arglist[0], XtNallowShellResize, TRUE);
    XtSetValues( toplevel, arglist, 1);
    .
    .
/**** Create and manage the widgets using either ****
**** low-level routines, high-level routines, or UIL. ****/
    .
    .
XtRealizeWidget (toplevel);
/***** MAIN INPUT LOOP *****/
    XtMainLoop();
    .
    .
}
```

The MAIN LOOP routine encloses calls to the intrinsic routines NEXT EVENT and DISPATCH EVENT in an infinite loop. The NEXT EVENT intrinsic routine returns the value from the head of the input queue associated with the application. The DISPATCH EVENT intrinsic routine calls the appropriate event handlers and passes them the widget identifier, the event, and the application-specific data registered with each procedure. The MAIN LOOP routine never returns. Your application should terminate from a callback routine as a result of a user action.

If you have created an application context, as described in Section 2.3, you must use the APPLICATION MAIN LOOP intrinsic routine to enter an event-processing loop. The APPLICATION MAIN LOOP intrinsic routine takes one argument: the application context that you created using the CREATE APPLICATION CONTEXT intrinsic routine. As with the MAIN LOOP intrinsic routine, the APPLICATION MAIN LOOP intrinsic routine never returns.

2.8 Creating a Callback Routine

You associate the functions of your application with its user interface using callback routines. All callback routines have three standard arguments:

- The widget identifier of the widget making the callback
- The tag (application-specific data)
- The callback data structure (widget-specific data)

The following sections describe these arguments.

2.8.1 Identifying the Widget Performing the Callback

The first standard argument to a callback routine identifies the widget performing the callback. The callback routine used in the Hello World! application, shown in Example 2–12 in Section 2.8.3, uses this information when it calls the SET VALUES intrinsic routine to change the text contained in the push button widget.

2.8.2 Associating Application-Specific Data with a Widget

In the second standard argument to a callback routine, the tag, you can associate data with a widget. The widget passes this data to your application when it performs a callback. You can use this argument to pass integers, text strings, application-specific data structures, or any other type of data you define. The XUI Toolkit performs no processing on this data; it simply passes it to your application when the widget executes the callback routine. (The Hello World! application passes a 0 as its tag value because it does not use this feature. However, the DECburger sample application uses the tag argument to identify each widget in its user interface.)

2.8.3 Widget-Specific Callback Data

In the third standard argument to a callback routine, the callback data structure, the widget returns other data to your application. The content of this data structure varies among the XUI Toolkit widgets. The *VMS DECwindows Toolkit Routines Reference Manual* describes the callback data returned by each widget. At a minimum, all XUI Toolkit widgets that perform callbacks return the following data in their callback data structure:

- The reason for the callback
- The address of the last event data structure on the X event queue

In the reason field of the callback data structure, the widget returns the callback reason. The reason identifies the event or sequence of events that caused the widget to perform the callback. The XUI Toolkit defines a set of constants to identify each callback reason. Each widget supports its own set of callback reasons. You can find out why the widget performed the callback by reading the value of this field of the callback data structure.

Creating a VMS DECwindows Application

2.8 Creating a Callback Routine

You typically only need to read the reason field when using certain high-level routines or if your application specifies the same callback routine for different callback reasons. Using low-level routines or UIL, you can associate a callback routine with a specific widget attribute associated with a particular callback. For example, in the Hello World! application using low-level routines, the callback routine is associated with the **activate_callback** attribute. However, some high-level routines do not provide access to every callback attribute supported by a widget. These routines associate the callback routine list you pass as the value of the callback argument with many callback attributes supported by the widget.

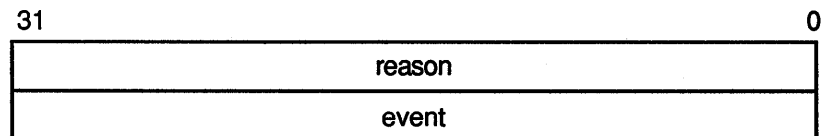
The event data structure contains information about the event that caused the callback. For more information about the data returned in the event data structure, see the *VMS DECwindows Xlib Programming Volume*.

The following is the definition of the minimum callback data structure returned by every XUI Toolkit widget that performs a callback.

```
typedef struct {  
    int      reason;  
    XEvent  *event;  
} DwtAnyCallbackStruct;
```

Figure 2-5 details the VAX definition of this data structure.

Figure 2-5 Widget Callback Data Structure (VAX Binding)



ZK-0091A-GE

Example 2-12 shows the callback routine used with the Hello World! application. The first time this routine is executed, it changes the text of the label in the push button widget. The second time this routine is executed, it causes the application to terminate.

Example 2-12 Hello World! Application Callback Routine

```

.
.
.
❶ static void helloworld_button_activate( widget, tag, callback_data )
    Widget widget;
    char *tag;
    DwtAnyCallbackStruct *callback_data;
{
    Arg arglist[2];
    static int call_count = 0;
    call_count += 1 ;
    switch ( call_count )
    {
        case 1:
❷ XtSetArg( arglist[0], DwtNlabel, DwtLatin1String("Goodbye\nWorld!") );
        XtSetArg( arglist[1], DwtNx, 11 );
❸ XtSetValues( widget, arglist, 2 );
        break ;
        case 2:
            exit(1);
        break ;
    }
}

```

- ❶ The Hello World! callback routine uses the standard three arguments that all callback routines must use.
- ❷ The Hello World! application sets up an argument list in which it assigns values to two push button widget attributes. In the first statement, the callback routine assigns a new text string to the **label** attribute of the push button widget in an argument list. In the second call to the SET ARG intrinsic routine, Hello World! assigns a new value to the x-coordinate of the push button widget. The longer text string requires this change in the x-coordinate to keep the push button widget centered in the dialog box widget.
- ❸ The Hello World! callback routine passes the argument list to the intrinsic routine SET VALUES. This routine replaces the original text string in the push button widget with the new text string and repositions the push button by assigning a new value to the x-coordinate using the x attribute.

2.8.4 Guidelines for Creating Callback Routines

Because your application needs to continually check the incoming event queue, you should write callback routines that execute quickly. If your application enters a callback routine that requires extensive processing time (more than 0.25 of a second), your application could miss processing important events. For example, if a portion of the user interface of your application that had been obscured on the display becomes visible, the widget needs to receive this expose event so that it can repaint the screen.

Creating a VMS DECwindows Application

2.8 Creating a Callback Routine

One way to perform lengthy processing without ignoring the event queue is to perform the processing in a **work procedure**. A work procedure is like a callback routine except that it is not associated with any particular widget. Instead, you register work procedures with your application using the ADD WORK PROC intrinsic routine. When the NEXT EVENT or PROCESS EVENT intrinsic routines have no other events to process, they call the work procedure that you registered. (The main input loop of an application is an infinite loop that continually calls NEXT EVENT to check the input queue for incoming events.) You can register multiple work procedures; however, the last one added is the one that is executed.

You create a work procedure as you would create a callback routine except that a work procedure only takes a tag as a standard argument. As with a callback routine, the tag is any data that you specify. With work procedures, you can use the tag to maintain state information on the progress of the processing so that each time the NEXT EVENT intrinsic routine calls the work procedure, it performs another segment of the processing. As with standard callback routines, your application should not remain in a work procedure for a long time without checking the input queue.

A work procedure should always return a Boolean value. While it returns false, the work procedure that you registered is called repeatedly. Once your work procedure returns true, it is removed from the list of work procedures. You can also explicitly remove a work procedure from the list by calling the REMOVE WORK PROC intrinsic routine.

Example 2-13 illustrates how to add a work procedure.

Example 2-13 Adding a Work Procedure

```
.
.
Boolean work_proc();
int state;
.
.
XtAddWorkProc( work_proc, &state );
.
.
Boolean work_proc( state )
    int *state;
{
    /* perform processing */
}
```

2.9 Manipulating the Interface at Run Time

Callbacks provide a one-way communication path from the interface to the application. To manipulate the interface once it has been displayed, you must use widget manipulation routines.

Creating a VMS DECwindows Application

2.9 Manipulating the Interface at Run Time

The XUI Toolkit provide two types of widget manipulation routines:

- Standard widget manipulation routines
- Widget-specific manipulation routines

2.9.1 **Standard Widget Manipulation Routines**

The standard widget manipulation routines perform general operations on all XUI Toolkit widgets. For example, using the standard widget manipulation routine SET VALUES, you can assign a value to a widget attribute after the widget has been created.

The X Toolkit intrinsics provide the standard widget manipulation routines of the XUI Toolkit. Table 2-5 lists the commonly used intrinsics routines that provide for standard widget manipulation.

Table 2-5 Standard Widget Manipulation Routines

Routine	Function
REALIZE WIDGET	Create the widget window and map it
DESTROY WIDGET	Destroy a widget including its window and data
GET VALUES	Retrieve the value of one or more widget attributes
SET VALUES	Set the value of one or more widget attributes
MANAGE CHILD	Add a widget to a composite widget's list of managed children
MANAGE CHILDREN	Add a list of widgets to a composite widget's list of managed children
UNMANAGE CHILD	Remove a widget from a composite widget's list of managed children
UNMANAGE CHILDREN	Remove a list of widgets from a composite widget's list of managed children
ADD CALLBACKS	Add a callback routine to a widget
REMOVE CALLBACKS	Remove a callback routine from a widget

The Hello World! application uses the standard widget manipulation routine SET VALUES to change the text of the label displayed in the push button widget.

As with any widget attribute, you can use the SET VALUES intrinsic routine to assign a callback routine list as the value of a callback attribute. However, the SET VALUES intrinsic routine destroys the existing callback routine list when it assigns this new value. The callback routine list of a widget can contain callbacks set up internally by the parent widget. For this reason, the XUI Toolkit provides the intrinsic routines ADD CALLBACKS and REMOVE CALLBACKS to add or delete callback routines.

Creating a VMS DECwindows Application

2.9 Manipulating the Interface at Run Time

2.9.2 Widget-Specific Manipulation Routines

In addition to standard widget manipulation routines, the XUI Toolkit includes many widget-specific manipulation routines. These routines perform common operations associated with a particular widget. For example, a toggle button widget, which indicates on and off states, has two associated widget-specific manipulation routines that allow you to read the current state or set the current state.

2.10 Complete Listing of the Hello World! Sample Application

The following sections contain the source code for three versions of the Hello World! sample application. Each version illustrates a different method of creating the widgets in the user interface. All three versions of the program produce the same result.

2.10.1 Using Low-Level Routines to Create the Hello World! User Interface

Example 2-14 is the setup section of the Hello World! application that uses low-level routines to create the widgets of the user interface.

Example 2-14 Setup Section of the Hello World! Application Using Low-Level Routines

```
#include <decw$include/DwtAppl.h>
static void helloworld_button_activate();
static DwtCallback callback_arg[2];
/***** Main Program *****/
int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Widget toplevel, helloworldmain, button, label;
    Arg arglist[5];
/***** Set up the User Interface *****/
    toplevel = XtInitialize ("Hi","helloworldclass",NULL, 0, &argc, argv);
    XtSetArg (arglist[0], XtNallowShellResize, TRUE);
    XtSetValues (toplevel, arglist, 1);
    ❶ helloworldmain = DwtDialogBoxCreate (toplevel, "MAINWIN", NULL, 0);
    XtSetArg (arglist[0], DwtNlabel,
        DwtLatin1String ("Press button once\nto change label;\ntwice to exit.));
    ❷ label = DwtLabelCreate (helloworldmain, "label", arglist, 1);

    callback_arg[0].proc = helloworld_button_activate;
    callback_arg[0].tag = 0;
    callback_arg[1].proc = NULL;
```

(continued on next page)

Creating a VMS DECwindows Application

2.10 Complete Listing of the Hello World! Sample Application

Example 2-14 (Cont.) Setup Section of the Hello World! Application Using Low-Level Routines

```
XtSetArg (arglist[0], DwtNx, 15);
XtSetArg (arglist[1], DwtNy, 40);
XtSetArg (arglist[2], DwtNactivateCallback, callback_arg);
XtSetArg (arglist[3], DwtNlabel, DwtLatin1String("Hello\nWorld!") );
③ button = DwtPushButtonCreate (helloworldmain, "button", arglist, 4);
XtManageChild (label);
XtManageChild (button);
XtManageChild (helloworldmain);
XtRealizeWidget (toplevel);
/***** End of Set Up *****/
.
.
.
```

- ❶ This version of the Hello World! application uses the low-level routine `DIALOG BOX CREATE` to create the dialog box widget. In the example, the dialog box widget is created without setting any widget attributes.
- ❷ Hello World! creates the label widget with a call to the low-level routine `LABEL CREATE`.
- ❸ Hello World! creates the only other widget in the application, the push button widget, using the low-level routine `PUSH BUTTON CREATE`.

2.10.2 Using High-Level Routines to Create the Hello World! User Interface

Example 2-15 is the setup section of the Hello World! application that uses high-level routines to create the widgets of the user interface.

Creating a VMS DECwindows Application

2.10 Complete Listing of the Hello World! Sample Application

Example 2-15 Setup Section of the Hello World! Application Using High-Level Routines

```
#include <decw$include/DwtAppl.h>
static void helloworld_button_activate();
static DwtCallback callback_arg[2];
/***** Main Program *****/
int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Widget toplevel, helloworldmain, button, label;
    Arg arglist[2];
/***** Set up the User Interface *****/
    toplevel = XtInitialize ("Hi","helloworldclass", NULL, 0, &argc, argv);
    XtSetArg (arglist[0], XtNallowShellResize, TRUE);
    XtSetValues (toplevel, arglist, 1);
    ❶ helloworldmain = DwtDialogBox (toplevel, "MAINWIN", TRUE, 0, 0,
        DwtLatin1String ("Hi"), DwtWorkarea, 0, 0 );
    ❷ label = DwtLabel (helloworldmain, "label", 0, 0,
        DwtLatin1String ("Press button once\nto change label;\ntwice to exit."), 0 );
    callback_arg[0].proc = helloworld_button_activate;
    callback_arg[0].tag = 0;
    callback_arg[1].proc = NULL;
    ❸ button = DwtPushButton (helloworldmain, "button", 15, 40,
        DwtLatin1String ("Hello\nWorld!"), callback_arg, 0 );
    XtManageChild (label);
    XtManageChild (button);
    XtManageChild (helloworldmain);
    XtRealizeWidget (toplevel);
/***** End of Set Up *****/
    .
    .
    .
```

- ❶ This version of the Hello World! application uses the high-level routine **DIALOG BOX** to create the dialog box widget.
- ❷ Hello World! creates the label widget with a call to the high-level routine **LABEL**.
- ❸ Hello World! creates the only other widget in the application, the push button widget, using the high-level routine **PUSH BUTTON**.

Creating a VMS DECwindows Application

2.10 Complete Listing of the Hello World! Sample Application

2.10.3 Using UIL and DRM to Create the Hello World! User Interface

Example 2–16 and Example 2–17 are the files that make up the UIL/DRM version of the Hello World! application. Example 2–16 is the UIL specification file for the Hello World! user interface; Example 2–17 is the main application program of the Hello World! application. In this version, the application uses DRM routines to access the user interface database created with UIL.

Example 2–16 Hello World! Application UIL Specification File

```
❶module helloworld
    version = 'v2.0'
    names = case_sensitive

❷procedure
    helloworld_button_activate();

❸object
    helloworld_main : dialog_box {
        controls {
            label        helloworld_label;
            push_button helloworld_button;
        };
    };

❹object
    helloworld_button : push_button {
        arguments {
            x = 15;
            y = 40;
            label_label = compound_string('Hello', separate=true) &
                compound_string('World!');
        };
        callbacks {
            activate = procedure helloworld_button_activate();
        };
    };

❺object
    helloworld_label : label {
        arguments {
            label_label =
                compound_string('Press button once', separate=true) &
                compound_string('to change label;', separate=true) &
                compound_string('twice to exit.');
```

```
❻end module;
```

- ❶ Required statement to begin a UIL module. This statement assigns a name and a version number to the UIL module, and declares that the names of objects in the module are case sensitive.
- ❷ Procedure declaration of the callback routine to be associated with the push button widget. Note that this routine is also declared in the main application program for the Hello World! application.

Creating a VMS DECwindows Application

2.10 Complete Listing of the Hello World! Sample Application

- ③ Object declaration of the main Hello World! application widget, the dialog box widget. Note the listing of its two children in the controls list section of the object declaration.
- ④ Object declaration of the label widget child.
- ⑤ Object declaration of the push button widget.
- ⑥ Required statement to signify the end of the UIL specification file.

Example 2-17 Hello World! Application Using UIL

```
#include <stdio.h>
#include <decw$include/DwtAppl.h>

①static DRMHierarchy      s_DRMHierarchy;
②static char              *vec[]={ "helloworld.uid" };
③static DRMCode          class ;

static void helloworld_button_activate();

④static DRMCount         regnum = 1 ;
⑤static DRMRegisterArg   regvec[] = {
    { "helloworld_button_activate", (caddr_t)helloworld_button_activate }
};

int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    ⑥ Widget toplevel, helloworldmain;
    Arg arglist[1] ;

    /***** Set up the User Interface *****/

    ⑦ DwtInitializedDRM ();

    toplevel = XtInitialize("Hi", "helloworldclass", NULL, 0, &argc, argv);
    XtSetArg (arglist[0], XtNallowShellResize, TRUE) ;
    XtSetValues (toplevel, arglist, 1) ;

    ⑧ if (DwtOpenHierarchy (1, vec, NULL, &s_DRMHierarchy) != DRMSuccess)
    {
        printf ("can't open hierarchy");
    }

    ⑨ DwtRegisterDRMNames (regvec, regnum) ;

    ⑩ if (DwtFetchWidget (s_DRMHierarchy, "helloworld_main", toplevel,
        &helloworldmain, &class) != DRMSuccess)
        printf("can't fetch interface");

    ⑪ XtManageChild(helloworldmain);
    XtRealizeWidget(toplevel);
    .
    .
    .
}
```

- ① The application declares a variable, named *s_DRMHierarchy*, which is a pointer to a DRM data structure. This data structure describes the DRM database hierarchy. The routine *DwtOpenHierarchy* returns the value of *s_DRMHierarchy*.

Creating a VMS DECwindows Application

2.10 Complete Listing of the Hello World! Sample Application

- ② The application specifies the name (or names) of the UID file in an array of pointers to strings, named *vec*. Using DRM, you can specify a hierarchy of UID files. A search for widgets involves a search through this UID file hierarchy.
- ③ The example declares a variable, named *class*, to hold the DRM class value returned by the `FETCH WIDGET` routine. (The class variable is not used by the Hello World! application.)
- ④ The sample application initializes a variable that identifies the number of names DRM must register. This is later used in the call to the DRM routine `REGISTER NAMES`.
- ⑤ The application stores the name of the callback routine *helloworld_button_activate* and its address in an array for later use by the DRM routine `REGISTER NAMES`.
- ⑥ Note that this version of the Hello World! application only declares two variables to hold widget identifiers. This version does not need variables to hold the widget identifiers of the label or push button widgets because it does not manipulate these widgets. DRM manages the widgets automatically. The DRM routine `FETCH WIDGET` returns the widget identifier of the topmost widget in the application hierarchy when it creates the widgets. The application uses this widget identifier when it manages the topmost widget in the hierarchy. You can retrieve the widget identifiers for any widget created by DRM by requesting a **creation callback**.
- ⑦ The DRM routine `INITIALIZE DRM` initializes DRM. An application must initialize DRM before initializing the XUI Toolkit.
- ⑧ Using the DRM routine `OPEN HIERARCHY`, the application opens the UID file that contains the definition of the Hello World! application interface. The application specified the names of the UID files earlier in ②.
- ⑨ The call to the DRM routine `REGISTER DRM NAMES` maps the names of the callback procedures in the UIL specification file to the actual procedures in the program. The names and corresponding addresses were defined in ⑤.
- ⑩ In this call to the DRM routine `FETCH WIDGET`, the application fetches the widget named *helloworld_main* from the hierarchy of UID files. DRM retrieves the widget definition and creates the widget, returning its widget identifier in the variable *helloworldmain*.

At the same time it creates the main widget of the Hello World! application, `FETCH WIDGET` also fetches the definitions for all the children of the specified widget and creates them as well.
- ⑪ The application manages the topmost widget in its widget hierarchy and then realizes the widget, making the interface appear on the screen.

Creating a VMS DECwindows Application

2.10 Complete Listing of the Hello World! Sample Application

2.10.4 The Hello World! Sample Application Main Input Loop and Callback Routine

Example 2-18 is the main input loop and callback routine used by all three versions of the Hello World! application.

Example 2-18 Main Input Loop and Callback Routine of the Hello World! Application

```
.....
/***** Main Input Loop *****/
❶ XtMainLoop();

/***** Callback Routine *****/
❷static void helloworld_button_activate( widget, tag, callback_data )
    Widget widget;
    char *tag;
    DwtAnyCallbackStruct *callback_data;
{
    Arg arglist[2];
    static int call_count = 0;
    call_count += 1 ;
    switch ( call_count )
    {
        case 1:
            XtSetArg (arglist[0], DwtNlabel, DwtLatin1String("Goodbye\nWorld!") );
            XtSetArg (arglist[1], DwtNx, 11 );
            XtSetValues (widget, arglist, 2 );
            break ;
        case 2:
            exit(1);
            break ;
    }
}
```

- ❶ The Hello World! application enters its main input loop by calling the intrinsic routine MAIN LOOP. You call this routine no matter how you have created the interface.
- ❷ The callback routine used by the Hello World! application.

3

Creating a User Interface Using UIL and DRM

This chapter describes how to use the User Interface Language (UIL) to specify a user interface for a VMS DECwindows application. The chapter also describes how to access the compiled interface specification at run time using the XUI Resource Manager (DRM).

3.1 Overview of UIL and DRM

The User Interface Language (UIL) is a specification language for describing the initial state of a user interface for a VMS DECwindows application. The specification describes the objects (for example, menu widgets, dialog box widgets, label widgets, and push button widgets) used in the interface and specifies the routines to be called when the interface changes state (as a result of user interaction). You specify the user interface in a **UIL module**, which you store in a **UIL specification file**.

Using UIL, you can specify the following:

- Objects (widgets and gadgets) that make up your interface
- Arguments (or resources) of the objects you specify
- Callback routines and tags for each object
- The widget hierarchy for your application
- Literal values that can be fetched by the application

The UIL compiler has built-in tables containing information about widgets. For every object (widget or gadget) in the XUI Toolkit, the UIL compiler knows which widgets are valid children of the object, the object's arguments, and the valid callback reasons for the object. The UIL compiler uses this information to check the validity of your interface specification at compile time, thereby helping you reduce run-time errors. The *VMS DECwindows User Interface Language Reference Manual* describes the information stored in the UIL built-in tables.

The UIL compiler translates the UIL module into a **User Interface Definition (UID)** file. You include DRM routine calls in your application program that allow access to the UID file. During execution of the application, DRM builds the arguments list and makes the necessary calls to the widget creation routines in order to create the user interface. By default, the newly created interface conforms to the *XUI Style Guide*. UIL and DRM are components of the XUI Toolkit.

Using UIL and DRM offers many benefits. By specifying the widgets in the interface in a separate UIL module, the size of your application program (particularly in the setup portion) is greatly reduced. (Compare the different versions of the Hello World! application in Section 2.10.) Since the UIL specification exists in a separate file, you can change

Creating a User Interface Using UIL and DRM

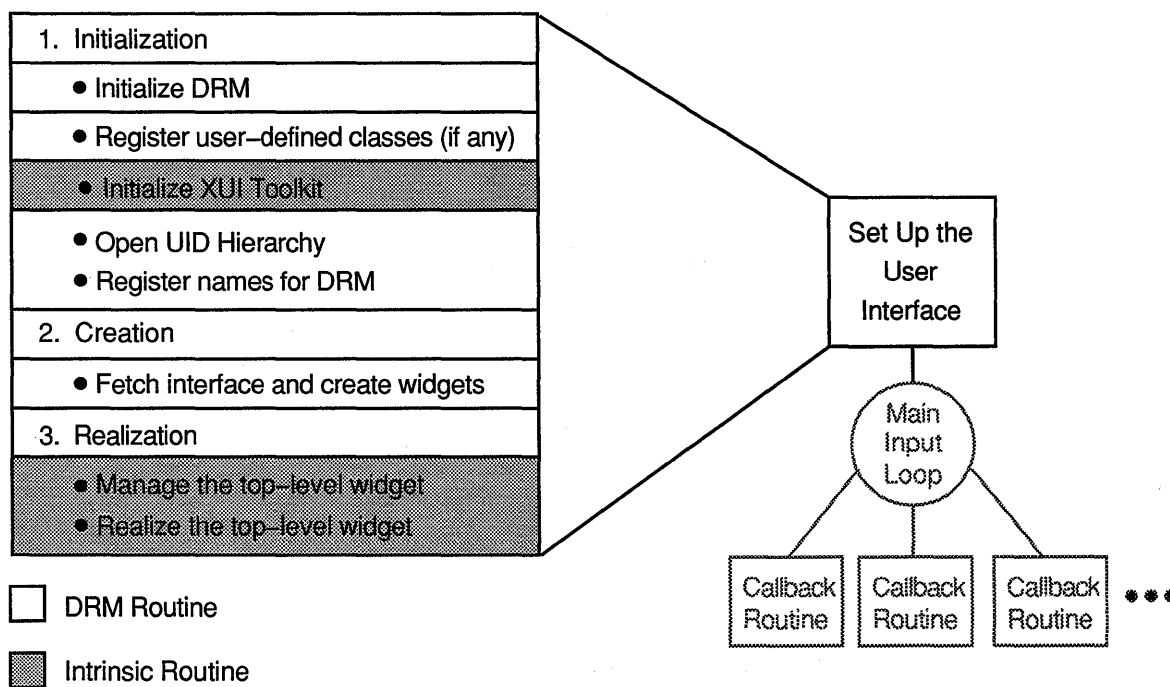
3.1 Overview of UIL and DRM

the user interface with few, if any, changes to the application program. This separation of form and function also allows you to develop multiple interfaces (for example, in different languages) for a single application.

When you use UIL and DRM, you do not call high- or low-level widget creation routines directly in your application program; you let UIL and DRM do much of this work for you. DRM simplifies and automates the widget creation process and allows the fastest possible initialization of a VMS DECwindows application. For example, DRM automatically creates shell widgets; you do not have to specify shell widgets in UIL. Since you do not need to know the format of the widget creation routine calls, UIL can be easier to learn. UIL and DRM are designed to be language independent and to make applications portable. UIL and DRM hide XUI Toolkit data structures and other programming details; you may not have to change your application every time the XUI Toolkit changes.

Figure 3-1 shows the steps involved at run time to set up an interface that was specified with UIL.

Figure 3-1 Setting Up a User Interface Specified with UIL



ZK-0166A-GE

As Figure 3-1 shows, setting up an interface specified with UIL requires the following steps:

1 Initialization

In the initialization step, the application program must make calls to DRM and intrinsic routines in the following sequence:

- Initialize DRM.

Creating a User Interface Using UIL and DRM

3.1 Overview of UIL and DRM

The DRM routine INITIALIZE DRM prepares your application to use DRM widget-fetching facilities.

- Register user-defined classes.

The DRM routine REGISTER CLASS saves the information needed to access the widget creation routine for a user-defined widget and to perform type conversion of user-defined arguments. If you use only XUI Toolkit widgets and gadgets in your interface, you do not call this routine. (Appendix D explains how to build your own widgets.)

- Initialize the XUI Toolkit.

The intrinsic routine INITIALIZE parses the command line used to invoke the application, opens the display, and initializes the XUI Toolkit.

- Open the UID hierarchy.

The **UID hierarchy** is the set of UID files containing the widget definitions for the user interface. The DRM routine OPEN HIERARCHY opens these UID files.

- Register names for DRM.

The DRM routine REGISTER DRM NAMES registers names and associated values for access by DRM. The values can be callback routines, pointers to user-defined data, or any other values. DRM uses this information to resolve symbolic references in UID files to their run-time values.

2 Creation

In the creation step, you call the DRM routine FETCH WIDGET to **fetch** the user interface. Fetching is a combination of widget creation and children management. The DRM routine FETCH WIDGET performs the following operations:

- Locates a widget description in the UID hierarchy
- Creates the widget and recursively creates the widget's children
- Manages all children as specified in the UID descriptions
- Returns the widget identifier

You specify the top-level widget of the application (usually the main window) and its parent (the widget identifier returned by the call to INITIALIZE) in the call to FETCH WIDGET. As a result of this single call, DRM fetches all widgets below the top-level widget in the widget hierarchy.

You can defer fetching portions of an application interface until requested by the end user. For example, you can defer fetching a pull-down menu widget until the user activates the corresponding pull-down menu entry. Consider deferring fetching of some portions of your interface if you need to improve the startup performance of your application. Deferred fetching is explained in Section 3.3.2.

Creating a User Interface Using UIL and DRM

3.1 Overview of UIL and DRM

3 Realization

The steps to manage and realize a user interface created with UIL and DRM are the same as those for an interface created with XUI Toolkit routines:

- Manage the top-level widget.

The intrinsic routine `MANAGE CHILD` adds a child to the top-level widget returned by the call to `INITIALIZE`. The entire widget hierarchy below the top-level widget in the interface (usually the main window widget) is automatically managed as a result of this call to `MANAGE CHILD`.

- Realize the top-level widget.

The intrinsic routine `REALIZE WIDGET` displays the entire fetched interface (the top-level widget and the widget hierarchy below it) on the screen.

DRM's role in a VMS DECwindows application is limited for the most part to widget creation. DRM makes run-time calls that create widgets from essentially invariant information (that is, information that does not change from one invocation of the application to the next). Once DRM fetches a widget (creates it and manages its children), all subsequent operations on the widget—realization, managing children after initialization, getting and setting resource values—must be done by run-time calls. After creation, modification of widgets during application execution is accomplished using widget manipulation routines. (DRM provides widget manipulation routines, which are described in Section 3.3. Section 2.9 describes run-time modification of widget attributes using XUI Toolkit routines.)

The *VMS DECwindows User Interface Language Reference Manual* completely describes UIL. DRM routines are fully described in the *VMS DECwindows Toolkit Routines Reference Manual*.

3.2 Specifying a User Interface Using UIL—A Sample Program

The examples in this section are based on the sample VMS DECwindows application called `DECburger`, which is introduced in Section 1.2. Specifically, this section explains how the interface for the `DECburger` sample application is specified in UIL. Figure 1-4 shows the `DECburger` interface.

Note that although the `DECburger` application is designed to show as many different widgets and UIL coding techniques as possible, this application does not use every feature of UIL. For a complete description of UIL, see the *VMS DECwindows User Interface Language Reference Manual*.

The examples in this section show only relevant portions of the UIL module for the `DECburger` application. Section 3.3 shows the relevant portions of the C language program for the `DECburger` application to illustrate the use of DRM run-time routines.

Note: In this section, reserved UIL keywords are shown in uppercase letters in the text. This is for emphasis only and is not required

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

by the UIL compiler. If you specify that names and keywords in your UIL module are case sensitive (see Section 3.2.4), you *must* put keywords in lowercase letters.

Do not use reserved keywords as names in a UIL module. The *VMS DECwindows User Interface Language Reference Manual* lists reserved and nonreserved keywords.

To specify an interface using UIL, perform the following steps:

- 1 Create one or more UIL specification files with file type UIL.

The number of files you use to completely specify the interface depends on the complexity of the application; the need for variations (for example, English and French versions); and the size of the development project team (on large projects, the UIL module can be distributed over several files to avoid access competition).

- 2 Declare the UIL module (begin a module block).

The module block header contains some module-wide declarations (such as case sensitivity for names and keywords in the module, the default character set for compound strings, and identification of which objects should be interpreted as gadgets).

- 3 Include the file containing useful UIL constants.

This file contains definitions of many constants you need to use to specify objects in UIL (for example, to align label widgets or to orient menu widgets). There is a file of constant definitions for each of the MIT C and VAX bindings.

- 4 Declare the callback routines referenced in the object declarations.

For each object in the UIL module, associate these routines with the callback reasons that the object supports. Define these callback routines in the application program.

- 5 Declare the values (integers, strings, colors, and so on) you will use in the object declarations.

- 6 Declare the interface objects (widgets and gadgets).

Declare interface objects in roughly the same order the objects appear in the widget hierarchy. Figure 6-7 in Section 6.5 shows a portion of the DECBurger widget hierarchy.

- 7 End the module block.

3.2.1 Recommended UIL Coding Techniques

The DECBurger UIL module shows recommended coding practices that should improve your productivity and increase the flexibility of your programs. This section explains how these practices can help you write better UIL modules. Descriptions of the particulars appear in later sections of this chapter. The language elements and semantics of UIL are similar to those in other high-level programming languages.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

3.2.1.1 Naming Values and Objects

The names of constants, labels, colors, icons, and widgets in the DECburger UIL module indicate their purpose in the application. For example, the name for the constant having integer value 12 is *k_burger_rare*. From its name, you can tell that this constant represents the choice Rare on the Hamburgers menu.

Similarly, the names for objects (widgets and gadgets) indicate their purpose in the application. In addition, object names should reflect the object type. For example, you can tell by its name that the *m_copy_button* is a button widget (of some kind) on a menu and is associated with the Copy option.

3.2.1.2 Declaring Values, Identifiers, and Procedures

Group value declarations according to purpose and list them near the beginning of the module. The UIL compiler requires only that values be declared before you reference them. So, although you could have a value section to declare a value immediately preceding an object section in which the value is used, you will be able to look up the definition of a particular value more easily if all declarations are in one place in the module.

In the DECburger UIL module, separate value sections are used to group values as follows:

- Constants for positioning attached dialog boxes
- Constants for callback routines
- Labels and other text strings
- Fonts
- Colors
- Color tables
- Icons

Having the constants for callback routines located in a single value section makes it easier to cut this section from the UIL module and paste it into the accompanying application program (since these constants must be defined in the program as well as in the UIL module).

By setting up all labels as compound string values, rather than hardcoding them in the object declarations, you can more easily change the labels from one language to another. (Specify a string as a compound string by using the UIL built-in function `COMPOUND_STRING`.)

In addition, declaring text strings as values allows the text string to be reused by several objects, thereby saving space in the generated UID file. For example, the value *quantity_label* is used three times in the DECburger order dialog box widget, but only one compound string, "Quantity", is stored in the UID file.

Note that some arguments for the simple text widget and the command window widget accept only null-terminated strings. For example, labels for these widgets must be declared as null-terminated text strings. The UIL compiler automatically converts a null-terminated text string to a

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

compound string if the string value is used to specify an argument that takes a compound string.

Similar to the value section, in the DECburger UIL module, all procedure declarations for callback routines are listed in a single procedure section at the beginning of the module, immediately following the module declaration and include directive.

The DECburger application does not use identifiers (which function like global variables). Treat identifier sections as you would treat value sections. (Identifiers are described in Section 3.5.)

You can isolate visual appearance information in a single section of a UIL module by declaring position and geometry values (for example, arguments **x**, **y**, **width**, and **height**) as UIL values. Having this information readily available in one place in the UIL module is very helpful for people who must translate the interface into another language. Language changes often require changes to widget size or position to accommodate different string lengths.

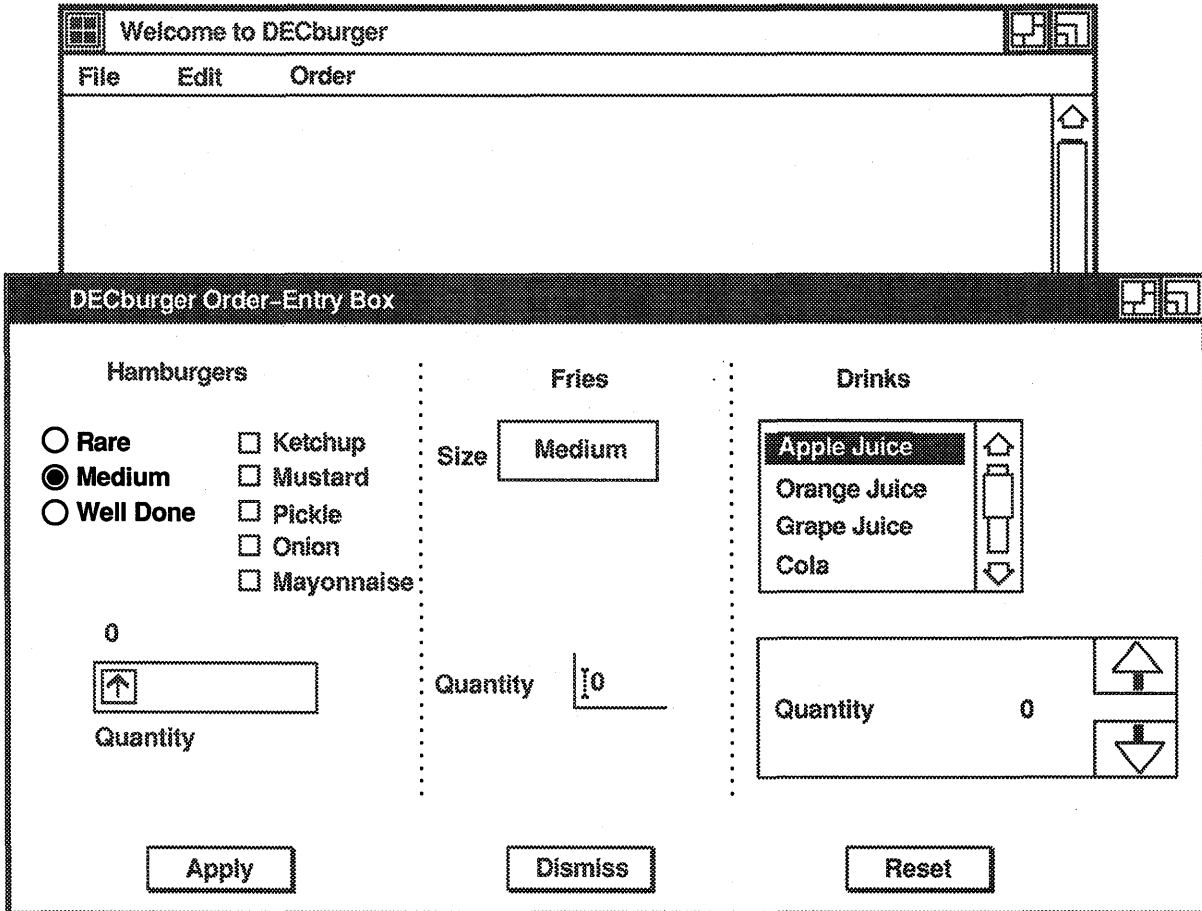
3.2.1.3 Declaring Objects

Once all your values, identifiers, and callback routines are declared, the rest of the UIL module consists of object declarations. The key technique here is to structure your module to reflect the widget hierarchy of the application interface. For example, in the DECburger UIL module, the choices for how the hamburger should be cooked are presented in a radio box widget having three children, which are toggle button widgets. Figure 3-2 shows how this radio box widget looks in the DECburger application interface.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Figure 3–2 Radio Box with Toggle Buttons in the DECburger Application



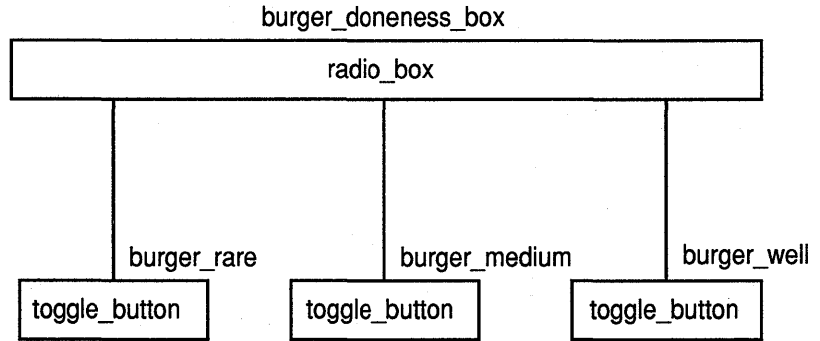
ZK-0160A-GE

Figure 3–3 shows how these widgets are arranged in a hierarchy, which is defined by the controls list for the radio box named *burger_doneness_box*.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Figure 3-3 Widget Hierarchy for the DECburger Radio Box Widget



ZK-0159A-GE

Example 3-1 shows the object declaration in the UIL module for the *burger_doneness_box* widget. Note that the children of the radio box widget (the three toggle button widgets) are declared immediately following the radio box object declaration. By ordering your object declarations in this way, you can get an idea of the overall widget hierarchy for your interface by scanning the UIL module.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3-1 Widget Hierarchy in the DECburger UIL Module

```
object
  burger_doneness_box : radio_box {
    arguments {
      .
      .
      .
    };
    controls {
      toggle_button burger_rare;
      toggle_button burger_medium;
      toggle_button burger_well;
    };
  };

object
  burger_rare : toggle_button {
    .
    .
    .
  };

object
  burger_medium : toggle_button {
    .
    .
    .
  };

object
  burger_well : toggle_button {
    .
    .
    .
  };
```

3.2.1.4 Using Local Definitions for Objects

If you need to define an object that is used as a child of a single parent and that will not be referenced by any other object in the UIL module, define the object in the controls list for its parent rather than in an object section of its own. This simplifies the UIL module and saves you from having to create an artificial name for that object. Example 3-14 in Section 3.2.8.2 shows local definitions for the separator gadgets used in the DECburger interface.

3.2.2 Creating a UIL Specification File

Store the UIL specification for a user interface in a UIL specification file. The UIL specification file contains the definitions of user interface objects and the values and callback routine names used in these definitions. The UIL compiler assumes a default file type of UIL. The compiled version of these definitions is stored in a User Interface Definition (UID) file. Compile a UIL specification file using the DCL command UIL.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

For example, if your interface specification is in the file DECBURGER.UIL, you compile this file by entering the following command:

```
$ UIL DECBURGER
```

By default, the compiled version of DECBURGER.UIL is named DECBURGER.UID. Specify the name of the UID file in the UID hierarchy list in your application program. (Section 3.3 describes how to use DRM to access the information stored in UID files.)

When you compile your UIL specification file, you can use the /VERSION qualifier. The /VERSION qualifier provides upward compatibility between the UIL compiler in VMS Version 5.1 and that in VMS Version 5.3.

In particular, the /VERSION qualifier allows you to continue building interfaces that will run under the XUI Toolkit in VMS Version 5.1 (for example, the processing of newline characters that are embedded in compound strings), while still being able to use the new UIL compiler features implemented for VMS Version 5.3.

Allowable values for the /VERSION qualifier are V1 (for VMS Version 5.1) and V2 (for VMS Version 5.3). The default is /VERSION=V2.

3.2.3 Structure of a UIL Module

The UIL specification file contains a module block that consists of a series of value, identifier, procedure, list, and object sections. There can be any number of these sections in a UIL module. The UIL has an **include directive** that allows you to include the contents of another file in your UIL module. You can use an include directive to specify one or more complete sections. You can place the include directive wherever a section is valid. You cannot specify a part of a section using an include directive.

You can also use the include directive to include a file of useful constants you need to specify values for some attributes (such as style and alignment). Section 3.2.5 describes this include file.

Example 3-2 shows the overall structure of a UIL module.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3-2 UIL Module Structure

```
!+
!      Sample UIL module
!-

module example                ! Module name
!+
!      Place module header clauses here.
!-

!+
!      Declare the VALUES, IDENTIFIERS, PROCEDURES, LISTS, and
!      OBJECTS here.
!-

end module;
```

3.2.4 Declaring the UIL Module

In the module declaration, you name the module and make module-wide specifications by using one or more module header clauses. Table 3-1 briefly explains the optional UIL module header clauses you can use in the module declaration.

Table 3-1 Optional UIL Module Header Clauses

Clause	Purpose	Default	Example
Version	Allows you to ensure the correct version of the UIL module is being used	None	version = 'v2.0'
Case sensitivity	Specifies whether names and keywords in the UIL module are case sensitive	Case insensitive	names = case_sensitive
Default character set	Specifies the default character set for string literals in the compiled UIL module	ISO_LATIN1	character_set = iso_latin6
Object variant	Specifies the default variant of objects defined in the module on a type-by-type basis	Widget	objects = { separator = gadget; push_button = widget; toggle_button = widget; label = gadget; }

Example 3-3 shows the module declaration for the DECBurger UIL module.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3-3 Module Declaration in the DECburger UIL Module

```
module decburger_demo
    version = 'v2.0'
    names = case_sensitive
    objects = {
        separator = gadget ;
        label = gadget ;
        push_button = gadget ;
        toggle_button = gadget ;
    }

include file 'DwtAppl.uil';
```

The name you specify in the UIL module declaration is stored in the UID file when you compile the module. The module declaration for DECburger specifies the following:

- DRM will identify the DECburger interface module by the name *decburger_demo*.
- This is the first version of this module.
- Names are case sensitive.
- All separator, label, push button, and toggle button objects are gadgets unless overridden in specific object declarations. All other types of objects are widgets.

Note: Refer to the UIL built-in tables in the *VMS DECwindows User Interface Language Reference Manual* to verify that you can specify a gadget as a child of a particular object. Some objects support only the widget variant of the push button and the toggle button. You might need to override the default gadget variant when defining a push button or toggle button that will be a child of one of these objects. The definition of the *up_value* push button in Example 3-16 in Section 3.2.9 shows how to override the default gadget variant set for push buttons in the DECburger UIL module.

3.2.5 Using the UIL Constants Include File

The last line in Example 3-3 is an example of a UIL include directive. When you compile the module, the UIL compiler replaces the include directive with the contents of the specified file.

The file containing definitions of UIL constants is named *DwtAppl.uil* for the MIT C binding and *DECW\$DWTDEF.UIL* for the VAX binding. By default, these files are located in the directories associated with the logical names *DECW\$INCLUDE:* and *SYS\$LIBRARY:*, respectively. The UIL constants file must be included before its contents are referenced. Therefore, include the UIL constants file immediately following the module header.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

The logical name `UIL$INCLUDE`, which points to the directory containing the UIL constants file, is defined in the command procedure for building and running the DECburger application (`DECBURGER.COM`). Therefore, the file specification in the include directive does not need to contain a directory specification. As long as you use the `DECBURGER.COM` command procedure to run the DECburger application, the UIL constants file will be included.

If you do not use the `DECBURGER.COM` command procedure to build the DECburger application, you need to define the logical name `UIL$INCLUDE` to point to the directory associated with the logical name `DECW$INCLUDE`, or you must completely specify the UIL constants file in the include directive as follows:

```
include file 'decw$include:DwtAppl.uil';
```

The UIL module for the DECburger application makes use of some of the constants defined in `DwtAppl.uil`. For example, the constants **`DwtModeless`** and **`DwtOrientationVertical`** shown in Example 3-4 come from this include file.

Example 3-4 Constants from Include File in the DECburger UIL Module

object

```
control_box : popup_dialog_box {
  arguments {
    title = k_decburger_title;
    style = DwtModeless;
    x = 300;
    y = 100;
    margin_width = 20;
    background_color = lightblue;
  };
  controls {
    label          burger_label;
    label          fries_label;
    label          drink_label;
    separator      {arguments {
                    x = 110;
                    y = 10;
                    orientation = DwtOrientationVertical;
                    height = 180; }};
  };
};
```

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

3.2.6 Declaring Procedures in UIL

Use a procedure declaration to declare a routine that can be used as a callback routine for an object. You can reference the routine name in object declarations that occur later in the UIL module.

As explained in Section 2.8, callback routines must be defined to accept three parameters: the identifier of the widget triggering the callback, a tag for user-defined information, and the callback data structure (which is unique to each widget). The widget identifier and callback structure parameters are under the control of the XUI Toolkit. The tag, however, is under the control of the application program.

In a UIL module, you can specify the data type of the tag to be passed to the corresponding callback routine at run time by putting the data type in parentheses following the routine name. When you compile the module, the UIL compiler checks that the argument you specify in references to the routine is of this type. Note that the data type of the tag must be one of the valid UIL types (see Table 3–3).

For example, in the following procedure declaration, the callback routine named *toggle_proc* must be passed an integer tag at run time. The UIL compiler checks that the parameter specified in any reference to the routine named *toggle_proc* is an integer.

```
procedure
    toggle_proc    (integer);
```

While you may use any UIL data type to specify the type of a tag in a procedure declaration, you must be able to represent that data type in the high-level language you will be using to write your application program. Some data types (such as integer, Boolean, and string) are common data types recognized by most programming languages. Other UIL data types (such as string tables) are more complicated and may require that you set up an appropriate corresponding data structure in the application in order to pass a tag of that type to a callback routine.

Table 3–2 summarizes the rules the UIL compiler follows for checking the argument type and count. The way you declare the callback routine determines which rule the UIL compiler uses to perform this checking.

Table 3–2 UIL Compiler Rules for Checking Argument Type and Count

Declaration Type	Description of Rule
No parameters	No argument type or argument count checking. You can supply no arguments or one argument in the procedure reference.
()	Checks that argument count is 0.
(ANY)	Checks that argument count is 1. Does not check argument type. Use ANY to prevent type checking on callback routine tags.
(value_type)	Checks for one argument of the specified value type.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3-5 shows that all callback routines in the DECburger UIL module specify that argument type and argument count are to be checked when the module is compiled.

Example 3-5 Procedure Declaration in the DECburger UIL Module

```
procedure
  toggle_proc      (integer);
  activate_proc    (integer);
  create_proc      (integer);
  scale_proc       (integer);
  list_proc        (integer);
  quit_proc        (string);
  show_hide_proc   (integer);
  pull_proc        (integer);
```

You can also use a procedure declaration to specify the creation routine for a user-defined widget. In this case, you must not specify any parameters. The creation routine is invoked by DRM with the standard four arguments passed to all low-level creation routines (see Section 2.4.1). Refer to Section 3.9 for information about working with user-defined widgets in UIL.

3.2.7 Declaring Values in UIL

A value declaration is a way of giving a name to a value expression. The value name can be referenced by declarations that occur later in the UIL module in any context where a value can be used. You must have previously declared a value before you reference it.

Table 3-3 lists the supported value types in UIL. See the *VMS DECwindows User Interface Language Reference Manual* for a complete description of UIL values.

Table 3-3 UIL Value Types

ANY	COLOR_TABLE	INTEGER
ARGUMENT	COMPOUND_STRING	INTEGER_TABLE
ASCIZ_STRING_TABLE	COMPOUND_STRING_TABLE	PIXMAP
BOOLEAN	FLOAT	REASON
CLASS_REC_NAME	FONT	STRING
COLOR	FONT_TABLE	TRANSLATION_TABLE

You can control whether values are local to the UIL module or globally accessible by DRM by specifying one of the following keywords in the value declaration:

- **EXPORTED**—A value that you declare as exported. This value is stored in the UID file as a named resource and can be either referenced by name in other UID files, or fetched from the UID file by the application using DRM literal fetching routines.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

- **IMPORTED**—A value that is defined as a named resource in a UID file. When you declare a value as imported, DRM looks outside the module in which the imported value is declared to get its value at run time. DRM resolves this value declaration with the corresponding exported declaration at application run time.
- **PRIVATE**—A value that is neither imported nor exported and is not stored as a distinct named resource in the UID file. You can reference a private value only in the UIL module containing the value declaration.

EXPORTED, **IMPORTED**, and **PRIVATE** are reserved UIL keywords. By default, values are private.

The DECburger application uses several kinds of values, as shown in the examples in the remainder of this section. There is a separate value section for each type of value to make it easier to find the value declaration during debugging.

3.2.7.1 Defining Arguments for Attached Dialog Box Widgets

Use an attached dialog box widget when you want to position and size the children of the dialog box widget relative to the other children in the dialog box widget or to the dialog box widget itself. Using an attached dialog box widget, you can omit the **x**, **y**, **width**, and **height** arguments in favor of relationship expressions.

The attached dialog box widget is an object that allows the definition of constraint arguments. That is, the attached dialog box widget has arguments that constrain the geometry of its children, thereby overriding the children's arguments that specify position and size.

To supply constraint arguments, you include the constraint arguments in the arguments list of the child object. The following example shows how to define attachments for a push button. The *VMS DECwindows User Interface Language Reference Manual* provides more information about defining constraint arguments.

```
object
  my_dialog_box: dialog_box {
    arguments {
      x = 70;
      y = 20;
      row = 35;
    };
    controls {
      push_button {
        arguments {
          adb_left_attachment = DwtAttachWidget;
          adb_left_offset = 10;
        };
      };
    };
  };
};
```

Note: Do not defer the creation of any widget that is referenced in an attachment. DRM requires all widgets referenced in attachments to be created before the attachments can be resolved. If you defer creation of a widget referenced in an attachment, the UIL module

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

will compile, but DRM will not be able to resolve the attachment; the result is a run-time error.

Refer to Section 7.5 for more information about specifying and using attached dialog box widgets in a VMS DECwindows application.

3.2.7.2 Defining Integer Values

Integer values are defined together in a single value section of the DECburger UIL module. These integers are used as tags in the callback routines. Example 3-6 shows a segment of this value section.

Example 3-6 Defining Integer Values in the DECburger UIL Module

```
value
  k_create_order      : 1;
  k_order_pdme       : 2;
  k_file_pdme        : 3;
  k_edit_pdme        : 4;
  k_nyi              : 5;
  k_apply            : 6;
  k_dismiss          : 7;
  k_noapply          : 8;
  k_cancel_order     : 9;
  k_submit_order     : 10;
  k_order_box        : 11;
  k_burger_rare      : 12;
  k_burger_medium    : 13;
  k_burger_well      : 14;
  k_burger_ketchup   : 15;
  k_burger_mustard   : 16;
  k_burger_onion     : 17;
  k_burger_mayo      : 18;
  k_burger_pickle    : 19;
  k_burger_quantity  : 20;
  .
  .
  .
```

You can also use the `INTEGER_TABLE` function to define an array of integer values. By using this method, you can pass more than one integer value per callback reason. The *VMS DECwindows User Interface Language Reference Manual* provides more information about the `INTEGER_TABLE` function.

3.2.7.3 Defining String Values

The next value section in the DECburger UIL module contains string value declarations (see Example 3-7). These strings are the labels for the various widgets used in the interface. Using values for widget labels rather than hardcoding the labels in the specification makes it easier to modify the interface (for example, from English to German). Putting all label definitions together at the beginning of the module makes it easier to find a label if you want to change it later. Also, a string resource declared as a value can be shared by many objects, thereby reducing the size of the UID file.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

A compound string can be associated with a character set and, optionally, a writing direction. Any text in a UIL module intended for dialog box widget instructions, labels for push button widgets, titles, and so forth should be declared as a compound string. In general, these strings require customization for international markets and must include character set information.

Because the `label_label` argument requires a compound string value, the UIL compiler would have automatically converted these strings to compound strings if they were declared as null-terminated strings. However, the conversion process can waste space in the UID file. (See the *VMS DECwindows User Interface Language Reference Manual* for more information about data storage consumption.)

The exception, `k_0_label_text`, is used to define an argument for the simple text widget; since this widget does not accept compound strings, the value for `k_0_label_text` must be a null-terminated text string.

Because there is no default character set specified in the module header and the individual string values do not specify a character set, the default character set associated with all these compound strings is ISO_LATIN1.

Note that some value names are indented in the value section. This indentation is not required but improves the readability of the UIL module. Specifically, this indentation indicates the widget hierarchy. For example, the widgets labeled *Cut*, *Copy*, *Paste*, *Clear*, and *Select All* are children of the widget labeled *Edit*. (Section 3.2.8 explains how to define the widget hierarchy.)

Example 3-7 Defining String Values in the DECburger UIL Module

```
value
  k_decburger_title      : compound_string("DECburger Order-Entry Box");
  k_nyi_label_text      : compound_string("Feature is not yet implemented");
  k_file_label_text     : compound_string("File");
  k_quit_label_text     : compound_string("Quit");
  k_edit_label_text     : compound_string("Edit");
  k_cut_dot_label_text  : compound_string("Cut");
  k_copy_dot_label_text : compound_string("Copy");
  k_paste_dot_label_text : compound_string("Paste");
  k_clear_dot_label_text : compound_string("Clear");
  k_select_all_label_text : compound_string("Select All");
  k_order_label_text    : compound_string("Order");
  k_show_controls_label_text : compound_string("Show Controls...");
  k_cancel_order_label_text : compound_string("Cancel Order");
  k_submit_order_label_text : compound_string("Submit Order");
  .
  .
  .
```

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

3.2.7.4 Specifying Multiline Compound Strings

In versions of VMS higher than 5.1 (for example, 5.3 and 5.4), the UIL compiler does not consistently process newline characters (\n) that are embedded in compound strings. The effect of a newline character embedded in a compound string depends entirely on the character set you specify, and the result may not always be a multiline compound string.

To guarantee that you create a multiline compound string, you must use the SEPARATE clause in the COMPOUND_STRING function and the concatenation operator (&) to join the segments into a multiline compound string. The SEPARATE clause takes the form SEPARATE = *boolean-expression*. For example, in VMS Version 5.1, the UIL compiler would generate a multiline compound string from the following input:

```
value
    sample_string : compound_string ("Hello\nWorld!");
```

To guarantee the same result in versions of VMS higher than 5.1 (for example, 5.3 and 5.4), you must use the following syntax:

```
value
    sample_line1 : compound_string ("Hello", separate = true);
    sample_line2 : compound_string ("World!");
    sample_string : sample_line1 & sample_line2;
```

To retain VMS Version 5.1 behavior of the newline character (\n) in a compound string, compile your UIL specification file using the /VERSION qualifier as follows:

```
$ UIL/VERSION=V1 MY_FILE.UIL
```

See the *VMS DECwindows User Interface Language Reference Manual* for more information on the COMPOUND_STRING function and the /VERSION qualifier.

3.2.7.5 Defining String Table Values

A string table is a convenient way to express a table of compound strings. Some widgets require a string table argument (such as the list box widget, which is used for drink selection in the DECBurger application).

Example 3–8 shows how to define a string table value in UIL.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3–8 Defining a String Table Value in the DECburger UIL Module

```
value
.
.
k_drinks_label_text      : compound_string("Drinks");
k_0_label_text          : '0';
k_drink_list_text       :
    string_table ('Apple Juice', 'Orange Juice', 'Grape Juice',
                  'Cola', 'Punch', 'Root beer', 'Water',
                  'Ginger Ale', 'Milk', 'Coffee', 'Tea');
k_drink_list_select     : string_table('Apple Juice');
.
.
```

The labels for the types of drinks are elements of the string table named *k_drink_list_text*. Notice that Apple Juice is a single element in the string table named *k_drink_list_select*. This value is passed to the *drink_list_box* widget to show apple juice as the default drink selection. (Refer to Section 8.2.2 for more information about showing default selection for the list box widget.)

The UIL compiler automatically converts the strings in a string table to compound strings, regardless of whether the strings are delimited by quotation marks or apostrophes.

3.2.7.6 Defining Font Values

Use the FONT function to declare a UIL value as a font.

Example 3–9 shows the declaration of a font value in the DECburger UIL module. This value is used later as the value for the **font_argument** attribute of the *apply_button*, *can_button*, and *dismiss_button* push button widgets.

Example 3–9 Declaring a Font Value in the DECburger UIL Module

```
value
k_button_font
: font('-ADOBE-Courier-Bold-R-Normal--14-140-75-75-M-90-ISO8859-1');
```

The *VMS DECwindows Xlib Programming Volume* lists the valid VMS DECwindows font names you can use as the argument to the FONT function.

Note that the UIL compiler converts a font to a font table when the font value is used to specify an argument that requires a font table value.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

3.2.7.7 Defining Color Values

Example 3-10 shows the value section in the DECburger module containing color declarations.

Example 3-10 Defining Colors in the DECburger UIL Module

```
value
  yellow      : color('yellow', foreground);
  red         : color('red', background);
  green       : color('green', foreground);
  magenta     : color('magenta', background);
  gold        : color('gold', foreground);
  lightblue   : color('lightblue', background);
```

By using the COLOR function, you can designate a string as specifying a color and then use that string for arguments requiring a color value. The optional keywords FOREGROUND and BACKGROUND identify how the color is to be displayed on a monochrome device.

The UIL compiler does not have built-in color names. Colors are a server-dependent attribute of a widget. Colors are defined on each server, according to the RGB (Red, Green, Blue) color model, and might have different RGB values on each server. The string you specify as the **color** argument to the COLOR function must be recognized by the server on which your application runs.

In a UID file, colors are represented as a character string. DRM calls X-level translation routines that convert the color string to the device-specific pixel value. If you are running on a monochrome server, all colors translate to black or white. If you are on a color server, the color names translate to their proper colors if the following two conditions are met:

- The color is defined.
- The color map is not yet full.

If the color map is full, even valid colors translate to black (foreground) or white (background).

If you have VMS DECwindows software installed on your system, you can see a listing of the color name strings understood by the VMS DECwindows servers by entering the following command:

```
$ TYPE SYS$MANAGER:DECW$RGB.COM
```

The command procedure DECW\$RGB.COM is executed during VMS DECwindows startup to set up the mapping of color names to RGB color indexes. These names are defined so that you can use reasonable names, rather than specify numeric color levels, to pick colors. (The server stores the equivalent numeric color levels of color names in the XDEFAULTS.DAT file.)

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

3.2.7.8 Defining Pixmap Values

Pixmap values are designed to let you specify labels that are graphic images rather than text strings. Pixmap values are not directly supported by UIL. Instead, UIL supports icons, which are a simplified form of pixmap. You use a character to describe each pixel in the icon.

You can generate pixmaps in UIL in two ways:

- Define an icon using the `ICON` function (and optionally use the `COLOR_TABLE` function to specify colors for the icon).
- Use the `XBITMAPFILE` function, specifying the name of an X bitmap file that you created outside UIL to be used as the pixmap value.

Example 3–11 shows the value section in the `DECburger` module containing a color table declaration.

Example 3–11 Defining a Color Table in the `DECburger` UIL Module

```
value
    button_ct                : color_table(
                              yellow='o'
                              ,red='.'
                              ,background color=' ');
```

The colors you specify when defining a color table must have been previously defined using the `COLOR` function. For example, in Example 3–11, the colors `yellow` and `red` were previously defined (see Example 3–10). Color tables must be private because the UIL compiler must be able to interpret their contents at compilation time to construct an icon. The colors within a color table, however, can be imported, exported, or private.

Example 3–12 shows how the `button_ct` color table is used to specify an icon pixmap. Referring to the definition shown in Example 3–11, each lowercase `o` in the icon defined in Example 3–12 is replaced with the color `yellow`, and each period (`.`) is replaced with the color `red`. Whatever color is defined as the background color when the application is run replaces the spaces.

In UIL, if you specify an argument of type **pixmap**, you should specify an icon or X bitmap file as its value. Example 3–12 is given as the value of the label on the drink quantity push button widget. (Refer to the definition of the `drink_quantity` widget in Section 3.2.9.)

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Some widget arguments accept a widget name (and the widget type) as a value. This use of a widget name is called a **symbolic reference** to a widget identifier and is explained in Section 3.6. You can also use a widget name (and type) as the **tag_value** argument to a callback routine.

All references to an object name must be consistent with the type you specified when you declared the object. (See Table 2-2 for a listing of UIL object types.)

Example 3-13 shows how the *file_menu* widget is declared in the DECburger UIL module.

Example 3-13 Declaring an Object in the DECburger UIL Module

```
object
    file_menu : pulldown_menu {
        arguments {
            label_label = k_file_label_text;
        };
        controls {
            push_button m_print_button;
            push_button m_quit_button;
        };
        callbacks {
            create = procedure create_proc (k_file_menu);
        };
    };
```

As shown in Example 3-13, an object declaration generally consists of three parts: an arguments list, a controls list, and a callbacks list. These parts are explained in the following sections.

3.2.8.1 Specifying Arguments in an Object Declaration

Use an arguments list to specify the arguments (attributes) for an object. An arguments list defines which arguments are to be specified in the **override_arglist** argument when the creation routine for a particular object is called at run time. An arguments list also specifies the values that these arguments are to have. The argument values you specify in UIL take precedence over any other source (for example, user or XUI Toolkit defaults). You identify an arguments list to the UIL compiler by using the keyword **ARGUMENTS**.

Each entry in the list consists of the argument name and the argument value. In Example 3-13, the *file_menu* widget has an argument named **label_label**, and the value of the argument is *k_file_label_text*. (The value *k_file_label_text* is a compound string defined in a value section at the beginning of the module.) The *VMS DECwindows User Interface Language Reference Manual* shows the UIL built-in arguments supported for each widget in the XUI Toolkit (including their UIL data type and default value) in an appendix.

Note: UIL has its own generic data types for arguments that map to VAX or MIT C binding data types. UIL forces you to specify argument values of the correct data type and is more structured than the XUI Toolkit in this regard. When you use UIL to specify an interface, you must use UIL data types as indicated in the UIL

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

built-in tables in the *VMS DECwindows User Interface Language Reference Manual*.

If you use the same argument name more than once in an arguments list, the last entry supersedes all previous entries, and the compiler issues a message.

If your application interface employs a user-defined widget, and this widget has arguments that are not UIL built-ins, you need to define these arguments with the ARGUMENT function. See the *VMS DECwindows User Interface Language Reference Manual* for more information about the ARGUMENT function.

3.2.8.2 Specifying Children in an Object Declaration

You use a controls list to define which widgets are children of, or controlled by, a particular widget. The controls lists for all the widgets in a UIL module define the widget hierarchy for an interface. If you specify that a child is to be managed (which is the default), at run time the widget is created and managed; if you specify that the child is to be removed from the managed list at creation (by including the keyword UNMANAGED in the controls list entry), the widget is only created. You identify a controls list to the UIL compiler using the keyword CONTROLS.

For example, in Example 3-13, the objects *m_print_button* and *m_quit_button* are children of the *file_menu* widget (which is a pull-down menu). (For each widget in the XUI Toolkit, the *VMS DECwindows User Interface Language Reference Manual* lists in an appendix the valid children of the widget.) The objects *m_print_button* and *m_quit_button* are defined as push button widgets, which are valid children of a pull-down menu widget (UIL object type *pull_down_menu*).

In Example 3-14, the pop-up dialog box widget called *control_box* is a top-level composite widget having a variety of widgets as children. Some of these children are also composite widgets, having children of their own. For example, the *button_box* and *burger_doneness_box* widgets are declared later on in the module, and each of these has a controls list.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3-14 Specifying Children in the DECburger UIL Module

```
object
control_box : popup_dialog_box {
  arguments {
    title = k_decburger_title;
    style = DwtModeless;
    x = 300;
    y = 100;
    margin_width = 20;
    background_color = lightblue;
  };
  controls {
    label      burger_label;
    label      fries_label;
    label      drink_label;
    separator  {arguments {
                x = 110;
                y = 10;
                orientation = DwtOrientationVertical;
                height = 180; }};
    separator  {arguments {
                x = 205;
                y = 10;
                orientation = DwtOrientationVertical;
                height = 180; }};
    work_area_menu  button_box;
    radio_box       burger_doneness_box;
    .
    .
    .
  }
}
```

Notice that the separator widgets are defined locally in the controls list for *control_box*, rather than in object sections of their own. As a result, the separator widgets do not have names and cannot be referenced by other objects in this UIL module. However, the local definitions make it easier for someone reading the UIL specification file to tell that the separator widgets are used only by the *control_box* widget. When you define an object locally, you do not need to create an artificial name for that object.

Unlike the arguments list (and the callbacks list, described in Section 3.2.8.3), when you specify the same widget in a controls list more than once, DRM creates multiple instances of the widget at run time when it creates the parent widget.

3.2.8.3 Specifying Callbacks in an Object Declaration

Use a callbacks list to define which callback reasons are to be processed by a particular widget at application run time. As shown in Example 3-13, each entry in a callbacks list has a reason (in this example, **create**) and the name of a callback routine (in this example, *create_proc*).

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

For XUI Toolkit widgets, the reason must be a UIL built-in name. For a user-defined widget, you can reference a user-defined reason that you previously specified by using the REASON function (see the *VMS DECwindows User Interface Language Reference Manual*). If you use a built-in reason in a widget definition, the UIL compiler ensures that the reason is supported by the type of widget you are defining. The *VMS DECwindows User Interface Language Reference Manual* lists built-in reasons for each widget in an appendix.

If you use the same reason more than once in a callbacks list, the last entry that uses that reason supersedes all others, and the UIL compiler issues a message.

You must have previously defined the routine name in a procedure declaration. For an example of a procedure declaration, see Example 3-5. In this example, the routine *activate_proc* was declared in the beginning of the UIL module.

Since the UIL compiler produces a UID file rather than an object module, the binding of the UIL name to the address of the routine entry point is not done by the VMS Linker. Instead, the binding is established at run time using the DRM routine REGISTER DRM NAMES. You call this routine prior to fetching any widgets, giving it the UIL names and the addresses of each callback routine. The name you register with DRM in the application program must match the name you specified in the UIL module. Section 3.3 explains how the DECburger callback routine names are registered with DRM.

UIL also allows you to specify multiple procedures per callback reason by defining the procedures as a type of list. Just as with any other list type, you can define a procedures list either locally in an object declaration or in a separate list section that you reference by name.

If you define a reason more than once (for example, when the reason is defined in a referenced procedures list and in the callbacks list for the object), all previous definitions are overridden by the latest definition.

Example 3-15 shows how to specify multiple callback procedures for the **activate** reason locally in the callbacks list of an object declaration.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3–15 Specifying Multiple Procedures per Callback Reason

```
object
  m_quit_button: push_button (
    arguments {
      .
      .
    };
  callbacks {
    activate = procedures {
      quit_proc ( 'normal demo exit' ); /* First proc for activate reason */
      shutdown (); /* Second proc for activate reason */
    };
  };
};
```

The *VMS DECwindows User Interface Language Reference Manual* contains more information on how to specify multiple callback procedures per reason.

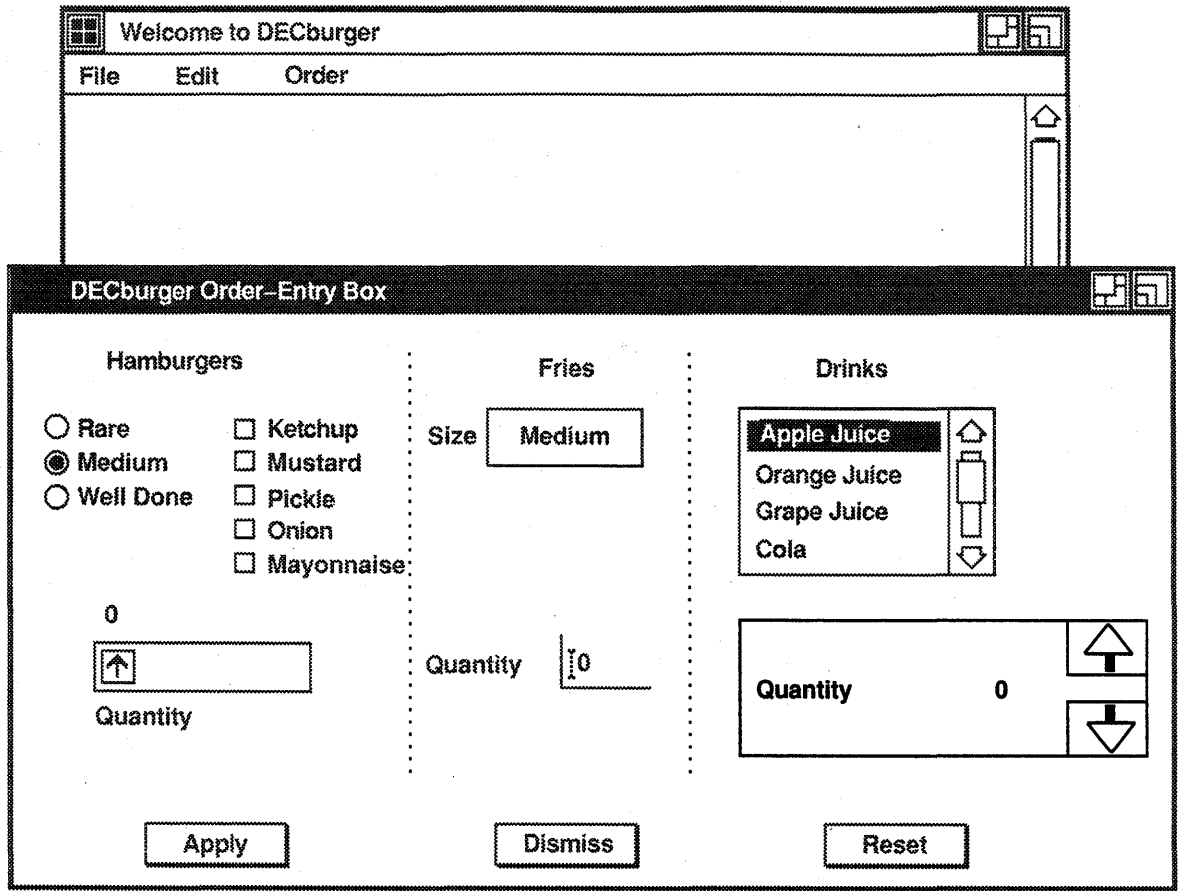
3.2.9 Specifying an Icon as a Widget Label

Figure 3–4 highlights the drink quantity selector. This widget in the user interface for the DECBurger application uses icons for the labels on its push button widgets. When the user clicks on the up arrow icon, the drink quantity increases. When the user clicks on the down arrow icon, the drink quantity decreases.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Figure 3-4 Using an Icon in the DECburger Application Interface



ZK-0161A-GE

Example 3-12 defined the icon named *drink_up_icon*. Example 3-16 shows how to specify this icon as a label for a push button widget. In the DECburger UIL module, the icon named *drink_up_icon* is a pixmap label argument to the *up_value* push button widget. In turn, the *up_value* widget is controlled by the *drink_quantity* attached dialog box widget.

Creating a User Interface Using UIL and DRM

3.2 Specifying a User Interface Using UIL—A Sample Program

Example 3-16 Using an Icon as a Label in the DECburger UIL Module

```
object
    drink_quantity : attached_dialog_box {
        arguments {
            x = 230;
            y = 85;
        };
        controls {
            label          quantity_label;
            label          value_label;
            push_button    up_value;
            push_button    down_value;
        };
    };
.
.
object
    up_value : push_button widget {
        arguments {
            y = 0 ;
            adb_left_attachment = DwtAttachWidget;
            adb_left_offset = 10 ;
            adb_left_widget = label value_label ;
            label_label_type = DwtPixmap;
            label_pixmap = drink_up_icon;
        };
        callbacks {
            activate = procedure activate_proc (k_drink_add);
        };
    };
};
```

3.3 Creating a User Interface at Run Time Using DRM

The XUI Resource Manager (DRM) creates interface objects based on definitions in UID files. Call DRM routines in your application program to initialize DRM, to provide information required by DRM to interpret information in UID files, and to create objects using UID definitions.

DRM also has routines that allow an application to read literal definitions from UID files. You create these literal definitions when you declare a value in UIL as exported. You can use these literals in your application program for any purpose. Section 3.3.3 explains how to read literals from UID files.

Similar to the way you can set values for a widget at run time using the XUI Toolkit routine SET VALUES, DRM provides a routine that allows you to set values at run time based on values stored in the UID file. Section 3.3.4 explains how to set values using values in the UID file.

You can use a DRM routine to fetch a widget and override widget attribute values or set values in addition to those you specified in a UIL module. In effect, a single object definition can be used like a template. Section 3.3.5 describes this routine.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

All definitions required to use DRM are contained in the file `decw$include:DwtAppl.h` for the MIT C binding or in the file `SYSS$LIBRARY:DECW$DWTDEF.H` for the VAX binding.

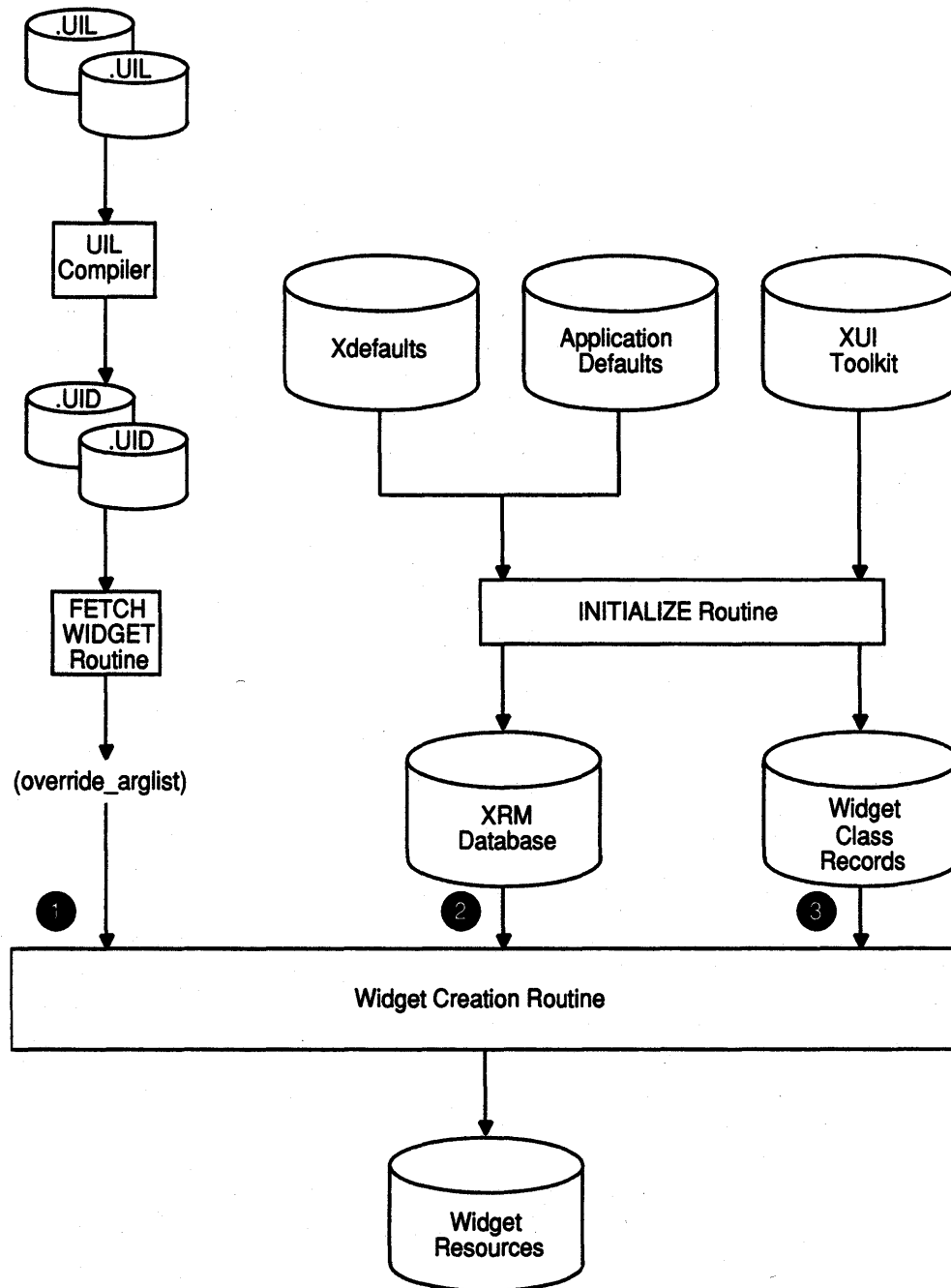
DRM does not replace, but rather complements, the X Resource Manager. The X Resource Database (an in-memory database, stored in the `XDEFAULTS.DAT` file) supplies default values. When you use UIL to specify a user interface, you do not need to specify all argument values (resources); you need to specify an argument only when you want to override the default value stored in the X Resource Database. DRM generates the `override_arglist` argument for the appropriate low-level widget creation routines at run time.

Figure 3-5 shows how widget argument values are applied inside the DRM fetch operation. The numbers ①, ②, and ③ indicate the sequence in which DRM searches for argument values and, therefore, the order of precedence. (Once DRM finds an argument definition, it stops searching.)

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Figure 3-5 Widget Creation in a DRM Fetch Operation



ZK-0128A-GE

The examples showing how to create a user interface at run time using DRM are based on the C program for the DECburger application. (The DECburger application can be found in the examples directory)

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

DECW\$EXAMPLES:. Section 1.2 describes how you can access these files.) The DECburger application demonstrates the most commonly used DRM routines. Table 3-4 briefly describes the DRM routines available to you; a complete description of these routines is given in the *VMS DECwindows Toolkit Routines Reference Manual*.

Table 3-4 DRM Routines and Functions

Routine or Function Name	Description
CLOSE HIERARCHY	Closes a UID hierarchy
DRM FREE RESOURCE CONTEXT	Frees a resource context
DRM GET RESOURCE CONTEXT	Sets up a resource context
DRM HGET INDEXED LITERAL	Fetches indexed (named) literals from a UID hierarchy (preferred method for fetching literal is to use either the FETCH COLOR LITERAL, the FETCH ICON LITERAL, or the FETCH LITERAL routine)
DRM RC BUFFER	Returns a pointer to the resource context buffer
DRM RC SET TYPE	Modifies the resource context type
DRM RC SIZE	Returns the size of the resource context
DRM RC TYPE	Returns the resource context type
FETCH COLOR LITERAL	Fetches a named color literal from a UID hierarchy
FETCH ICON LITERAL	Fetches a named icon literal from a UID hierarchy
FETCH INTERFACE MODULE	Fetches all the objects defined in some interface module in the UID hierarchy
FETCH LITERAL	Fetches a named string literal from a UID hierarchy
FETCH SET VALUES	Fetches the values to be set from literals stored in UID files
FETCH WIDGET	Fetches any indexed (named) application widget
FETCH WIDGET OVERRIDE	Fetches any indexed (named) application widget and overrides values stored in the UIC file with those supplied in the routine call
INITIALIZE DRM	Prepares an application to use DRM widget fetching facilities

(continued on next page)

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Table 3-4 (Cont.) DRM Routines and Functions

Routine or Function Name	Description
OPEN HIERARCHY	Allocates a hierarchy descriptor and opens all the UID files in the hierarchy
REGISTER CLASS	Saves the information needed to access the widget creation routine for a user-defined widget using the information in UID files and to perform type conversion of an arguments list
REGISTER DRM NAMES	Registers a vector of names and associated values for access by DRM

3.3.1 Accessing the UID File at Run Time

As explained in Section 3.1, a VMS DECwindows application whose interface is specified in UIL must contain calls to the following routines:

- **INITIALIZE DRM**—Prepare the application for fetching and other DRM facilities.
- **REGISTER CLASS**—Register user-defined widget classes with DRM (not required for XUI Toolkit objects).
- **INITIALIZE**—Parse the command line used to invoke the application, open the display, and initialize the XUI Toolkit.
- **OPEN HIERARCHY**—Bind the application program with the appropriate interface definition.
- **REGISTER DRM NAMES**—Register values used to resolve symbolic references in the interface definition.

The call to the **INITIALIZE DRM** routine must come before the call to the **INITIALIZE** routine. Example 3-17 shows the initialization of DRM and the XUI Toolkit in the DECburger application.

Example 3-17 Initializing DRM and the XUI Toolkit in the DECburger Application

```
unsigned int main(argc, argv)
    unsigned int argc;
    char *argv[];
{
    DwtInitializeDRM();

    toplevel_widget = XtInitialize("Welcome to DECburger",
        "example",
        NULL,
        0,
        &argc,
        argv);
}
```

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

The compiled interface, described in one or more UIL modules, is connected to the application by setting up a UID hierarchy at run time. The names of the UID files containing the compiled interface definitions are stored in an array. Because compiled UIL files are not object files (OBJ extension), this run-time connection is necessary to bind an interface with an application program. The DECBURGER application has a single UIL module (DECBURGER.UIL), so the UID hierarchy consists of one file (DECBURGER.UID). Example 3-18 shows the declaration of the UID hierarchy for DECBURGER.

Example 3-18 Declaring the UID Hierarchy for the DECBURGER Application

```
static DRMHierarchy s_DRMHierarchy;
.
.
.
static char *db_filename_vec[] =
    {"decburger.uid"
    };
```

The name of the UID hierarchy is *s_DRMHierarchy*. The array containing the names of the UID files in the UID hierarchy is *db_filename_vec*. In Example 3-19, the application opens this UID hierarchy. At this point in the application, DRM has access to the DECBURGER interface definition and can fetch widgets.

Example 3-19 Opening the UID Hierarchy for the DECBURGER Application

```
if (DwtOpenHierarchy(db_filename_num,
    db_filename_vec,
    NULL,
    &s_DRMHierarchy)
    !=DRMSuccess)
    s_error("can't open hierarchy");
```

The final step in preparing to use DRM to fetch widgets is to register a vector of names and associated values. These values can be the names of callback routines, pointers to user-defined data, or any other values. DRM uses the information provided in this vector to resolve symbolic references that occur in UID files to their run-time values. For callback routines, the vector provides addresses required by the XUI Toolkit. For names used as variables in UIL (identifiers), this information provides whatever mapping the application requires. (The use of identifiers is explained in Section 3.5.)

Example 3-20 shows the declaration of the names vector in the DECBURGER C program. In the DECBURGER application, the names vector contains only names of callback routines and their addresses.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Example 3-20 Declaring a Vector of Names to Register for DRM in the DECburger Application

```
static DRMRegisterArg reglist[] = {
    {"activate_proc", (caddr_t) activate_proc},
    {"create_proc", (caddr_t) create_proc},
    {"list_proc", (caddr_t) list_proc},
    {"pull_proc", (caddr_t) pull_proc},
    {"quit_proc", (caddr_t) quit_proc},
    {"scale_proc", (caddr_t) scale_proc},
    {"show_hide_proc", (caddr_t) show_hide_proc},
    {"show_label_proc", (caddr_t) show_label_proc},
    {"toggle_proc", (caddr_t) toggle_proc}
};

static int reglist_num = (sizeof reglist / sizeof reglist [0]);
```

The names are registered in a call to the REGISTER DRM NAMES routine, as shown in Example 3-21.

Example 3-21 Registering Names for DRM in the DECburger Application

```
DwtRegisterDRMNames(reglist, reglist_num);
```

3.3.2 Deferring Fetching

DRM allows you to defer fetching off-screen widgets until the application needs to display these widgets. There are two types of off-screen widgets: pull-down and pop-up. Whenever DRM fetches an off-screen widget, it also fetches the entire widget hierarchy below that widget. By deferring fetching of off-screen widgets, you can reduce the time it takes to start up your application.

The DECburger application uses deferred fetching. The pull-down menu widgets for the File, Edit, and Order options are not fetched when the main window widget is fetched. Instead, these menus are fetched and created by individual calls to the FETCH WIDGET routine when the corresponding pull-down menu entry widget is activated (selected by the end user). You can use the FETCH WIDGET routine at any time to fetch a widget that was not fetched at application startup.

The UIL module for the DECburger application is set up to allow either deferred fetching or a single fetch to create the entire widget hierarchy. To fetch the entire interface at once, remove the comment character (!) from the controls list for the *file_menu_entry*, *edit_menu_entry*, and *order_menu_entry* widgets. As long as the comment characters remain on the controls list for the pull-down menu entries, their associated pull-down menu widgets are no longer children; they are top-level widgets and can be fetched individually. Example 3-22 shows the object declaration for the *file_menu_entry*.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Example 3-22 DECburger UIL Module Setup for Deferred Fetching

```
object
    file_menu_entry : pulldown_entry {
        arguments {
            label_label = k_file_label_text;
        };
        !
        !     controls {
        !         pulldown_menu file_menu;
        !     };
        callbacks {
            pulling = procedure pull_proc (k_file_pdme);
            create = procedure create_proc (k_file_pdme);
        };
    };
```

When you remove the comment characters, the controls list on each pull-down entry widget specifies the pull-down menu widget as a child. The pull-down menu widgets are no longer top-level widgets; instead, they are loaded when the pull-down entry is created (that is, when the DECburger main window widget is fetched).

3.3.3 Retrieving Literal Values from UID Files

Using the literal fetching routines (FETCH COLOR LITERAL, FETCH ICON LITERAL, and FETCH LITERAL), you can retrieve any named, exported UIL value from a UID file at run time. These literal fetching routines are particularly useful when you want to use a value in a context other than for specifying an object. The three literal fetching routines allow you to treat the UID file as a repository for all the programming variables you need for your application interface. For example, you can store the following as named, exported literals in a UIL module for run-time retrieval:

- All the error messages to be displayed in a message box (stored in a string table)
- An ASCIZ string used to query the operating system (for example, to retrieve the correct version of the help library for a portable application)
- Language-dependent strings

In the C program for the DECburger application, the text string displayed in the title bar of the main window widget is supplied directly to the INITIALIZE routine, as shown in Example 3-23.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Example 3-23 Title Bar String for DECburger Application

```
toplevel_widget = XtInitialize ("Welcome to DECburger",
    "example",
    NULL,
    0,
    &argc,
    argv);
```

Alternatively, this string could be specified in a UIL module as a named, exported string, and retrieved from the UID file at run time with the `FETCH LITERAL` routine.

In the following example, the string for the DECburger title bar is defined in the UIL module:

```
value
    k_welcome_text      : exported 'Welcome to DECburger';
```

Example 3-24 shows the changes needed in the `DECBURGER.C` program to get the title bar string from the UID file.

Example 3-24 Getting a Value from the UID File for the DECburger Application

```
❶ static char * welcome_text_ptr;
❷ static int dtype;
❸ DwtFetchLiteral (s_DRMhierarchy, "k_welcome_text", NULL, &welcome_text_ptr, &dtype);
❹ XtFree (welcome_text_ptr);
```

- ❶ A character pointer to the string containing the text for the application title that will be retrieved from the UID hierarchy. This pointer is passed to the `FETCH LITERAL` routine, which is shown in ❸.
- ❷ Data type of the returned literal.
- ❸ The first parameter to the `FETCH LITERAL` routine is the identifier of the UID hierarchy containing the named value (literal) to be fetched.

The second parameter specifies the named value (as specified in UIL) to fetch from the UID hierarchy. This call to the `FETCH LITERAL` routine fetches the literal named `k_welcome_text` from the UID hierarchy named `s_DRMhierarchy`. Note that DRM does not do any type conversion when it retrieves literal values from a UID file.

The third parameter to the `FETCH LITERAL` routine is the display. You need to provide this parameter when fetching fonts and font lists. You can use the intrinsic `DISPLAY` function on any widget identifier to retrieve the display value.

- ❹ The intrinsic routine `FREE` is used to free the memory used for the welcome text string. You are responsible for freeing all allocated storage after the fetched value is no longer needed.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

3.3.4 Setting Values at Run Time Using UID Resources

The DRM routine `FETCH SET VALUES` allows you to modify at run time an object that has already been created. The `FETCH SET VALUES` routine works like the `SET VALUES` routine except that DRM fetches the values to be set from named, exported resources (literals) in the UID file. The fetched values are converted to the correct data type, if necessary, and placed in the `args` argument for a call to the XUI Toolkit routine `SET VALUES`. Since the `FETCH SET VALUES` routine looks for the literal values in a UID file, the argument names you provide to the `FETCH SET VALUES` routine must be UIL argument names (not XUI Toolkit attribute names).

You can think of the `FETCH SET VALUES` routine as a convenience routine that packages the functions provided by `FETCH LITERAL` and `SET VALUES`.

The value member of the name and value pairs passed to `FETCH SET VALUES` is the UIL name of the value, not an explicit value. When the application calls `FETCH SET VALUES`, DRM looks up the names in the UID file, then uses the values corresponding to those names to override the original values in the object declaration. The `FETCH SET VALUES` routine, therefore, allows you to keep all values used in an application in the UIL module and not in the application program. (The values you pass to the `FETCH SET VALUES` routine must be named, exported literals in the UIL module.)

The `FETCH SET VALUES` routine offers the following advantages:

- It performs all the necessary UIL resource manipulation to make the fetched UIL values usable by the XUI Toolkit. (For example, the `FETCH SET VALUES` routine enables a UIL icon to act as a pixmap.)
- It lets you isolate a greater amount of interface information from the application program (to achieve further separation of form and function).

There are some limitations to the `FETCH SET VALUES` routine:

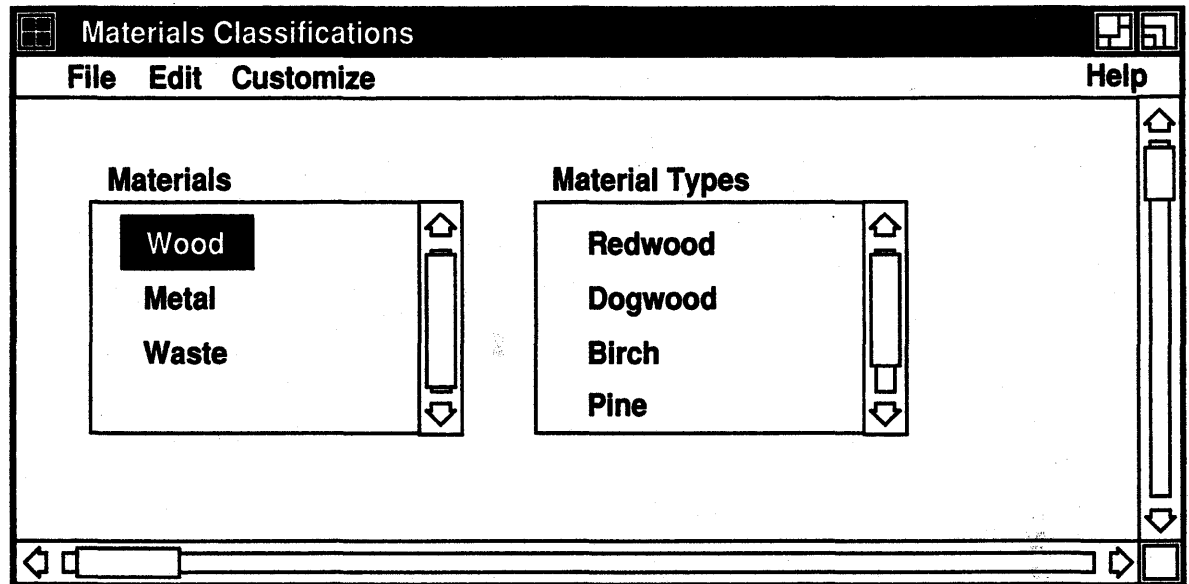
- All values in the `args` argument *must* be names of exported resources listed in a UIL module (UID hierarchy); therefore, the application *cannot* provide computed values from within the program itself as part of the arguments list.
- It uses the `SET VALUES` routine, ignoring the possibility of the less costly high-level routine that the widget itself may provide.

The examples in this section are based on a simple application that displays text in two list box widgets. The text displayed in the second list box widget depends on what the user selected in the first. Figure 3-6 shows the interface for this application.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Figure 3-6 Sample Application Using the FETCH SET VALUES Routine



ZK-0540A-GE

This application is well-suited to using the DRM routine `FETCH SET VALUES` for the following reasons:

- The data (list box widget contents) is all known in advance; that is, the values themselves do not need to be computed at run time.
- The data consists of tables of compound strings that appear in the user interface and, therefore, must be translated for international markets. (Strings that must be translated should be stored in a UID file.)
- The `FETCH SET VALUES` routine performs all the necessary manipulations to make the string table usable by the list box. Because the program will not be using the fetched string table directly but intends only to modify the visual appearance of a widget based on items in the table, the `FETCH LITERAL` routine is less convenient to use.

Example 3-25 shows the UIL module for this application; Example 3-26 shows the C program. The segment of the UIL module shown in Example 3-25 assumes that the module header, procedure declarations, include files, and value declarations for each of the names used in the example are in place.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Example 3-25 UIL Module for the FETCH SET VALUES Application

```
value
  ❶cs_wood      : compound_string("Wood");
    cst_materials_selected : string_table(cs_wood);

  ❷cst_materials : exported string_table(
    cs_wood,      ! material type 1
    "Metal",      ! material type 2
    "Waste");     ! material type 3

  ❸  cst_type_1  : exported string_table(      ! Materials for type 1 (wood)
    "Redwood", "Dogwood", "Birch", "Pine", "Cherry");
  l_count_type_1 : exported 5;

  cst_type_2  : exported string_table(      ! Materials for type 2 (metal)
    "Aluminum", "Steel", "Titanium", "Iron", "Linoleum");
  l_count_type_2 : exported 5;

  cst_type_3  : exported string_table(      ! Materials for type 3 (waste)
    "Toxic", "Solid", "Biodegradable", "Party Platforms");
  l_count_type_3 : exported 4;

  k_zero : exported 0;

object
  materials_ListBox : list_box
  {
    arguments
    {
      x = k_tst_materials_ListBox_x;
      y = k_tst_materials_ListBox_y;
      width = k_tst_materials_ListBox_wid;
      visible_items_count = 4;
      items = cst_materials;
      selected_items = cst_materials_selected;
      single_selection = true;
      resize = DwtResizeFixed;
    };
    callbacks
    {
      help = procedure tst_help_proc(k_tst_materials_ListBox_key);
      create = procedure tst_create_proc(k_tst_materials_ListBox);
      single = procedure tst_single_proc(k_tst_materials_ListBox);
    };
  };
};
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Example 3-25 (Cont.) UIL Module for the FETCH SET VALUES Application

```
types_ListBox      : list_box
(
  arguments
  (
    x = k_tst_types_ListBox_x;
    y = k_tst_types_ListBox_y;
    width = k_tst_types_ListBox_wid;
    visible_items_count = 4;
    items = cst_type_1;
    single_selection = true;
    resize = DwtResizeFixed;
  );
  callbacks
  (
    help = procedure tst_help_proc(k_tst_types_ListBox_key);
    create = procedure tst_create_proc(k_tst_types_ListBox);
    single = procedure tst_single_proc(k_tst_types_ListBox);
  );
);
```

- ① Prefixes on value names indicate the type of value. For example, *cs_* means compound string, *cst_* means compound string table, and *l_* means longword integer.
- ② This string table provides the contents for the Materials list box widget (on the left in Figure 3-6). This string table does not need to follow the naming scheme for the string table in the Material Types list box widget (that is, *cst_type_n*) because the contents of the Materials list box does not change once the application is realized. (The numbering of the string tables in ③ is vital to the proper functioning of the Material Types list box widget. The string table for the Materials list box widget could have been named anything.)
- ③ These string tables provide the contents for the various versions of the Material Types list box widget (on the right in Figure 3-6). Each one of these lists of strings corresponds (in order) to the string names in the first list box widget (Materials). These tables are numbered to facilitate programming. When the user selects an item in the Materials list box widget, the index of the selected item will be concatenated with the string "cst_type_" to form the name of one of these tables. This named table will be retrieved with the FETCH SET VALUES routine and placed in the Material Types list box widget.

Note that in addition to the string table, a count of the number of items in the table is declared as an exported value. This is done because using the SET VALUES routine on a list box widget requires that three arguments be set: **items**, **item_count**, and **selected_item_count** (which must be set to 0).

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

In the C program shown in Example 3–26, note the activation procedure named *tst_single_proc*, where the user's selection causes the program to act.

Example 3–26 C Program for the FETCH SET VALUES Application

```
#define k_zero_name "k_zero"
#define k_table_name_prefix "cst_type_"
#define k_table_count_name_prefix "l_count_type_"

❶ void tst_single_proc(w,object_index,callbackdata)
    Widget      w;
    int         *object_index;
    DwtListBoxCallbackStruct *callbackdata;
{
    ❷ char *t_number;
    ❸ char t_table_name[32] = k_table_name_prefix;
    char t_table_count_name[32] = k_table_count_name_prefix;
    ❹ Arg r_override_arguments[3] =
        {{DwtNitems,NULL},{DwtNitemsCount,NULL},{DwtNselectedItemsCount,k_zero_name}};
    switch (*object_index)
    {
        ❺ case k_tst_materials_ListBox:
            {
                ❻ sprintf(&t_number,"%d",callbackdata->item_number);

                ❼ strcpy(&t_table_name[sizeof(k_table_name_prefix)-1],&t_number);
                XtSetArg(r_override_arguments[0],DwtNitems,&t_table_name);

                ❸ strcpy(&t_table_count_name[sizeof(k_table_count_name_prefix)-1],&t_number);
                XtSetArg(r_override_arguments[1],DwtNitemsCount,&t_table_count_name);

                ❹ DwtFetchSetValues(ar_DRMHierarchy,
                    object_ids[k_tst_types_ListBox],
                    r_override_arguments,3);

                break;
            };
        ❺ case k_tst_types_ListBox:
            {
                :
                :
                break;
            };
    };
};
```

- ❶ Routine that handles the **single** callback functions for any object. When the user selects an item in a list box widget, the contents of a neighboring list box are replaced. This routine uses the list box widget callback structure named *DwtListBoxCallbackStruct*. This structure contains the following fields: *reason*, *event pointer*, *item*, *item_length*, and *item_number*.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

- ② Used to form the string version of the item number.
- ③ Local character storage.
- ④ Override argument list for the FETCH SET VALUES routine.
- ⑤ User has selected an item from the Materials list box widget. The application needs to place a new items list in the Material Types list box widget. The string tables stored in the UID file are named *cst_type_ "index number"* and their count names are *l_count_type_ "index number"* (where "index number" corresponds to the item's position in the list box widget). Using the index of the selected item from this box, the application forms the name of the appropriate compound string table.

Using the item number instead of the text value of the selection separates the function of the application from the form (in this case, the contents of the list box widgets) and reduces complexity. If the program used the text value of the selected item as the means to determine what to display, it would need to deal with possible invalid characters for a UIL name in the text and would have to convert the text value (a compound string) to a null-terminated string so that the string could be passed to the XUI Toolkit routine SET VALUES.

- ⑥ Form the string version of the item number.
- ⑦ Form the name of the string table.
- ⑧ Form the name of the string table count.
- ⑨ Fill the Material Types list box widget with a new list of items.
- Similar selection recording code goes here.

3.3.5 Using an Object Definition as a Template

The UID file stores object definitions that contain argument value specifications. When DRM fetches the object, these values override XUI Toolkit default values for the specified arguments when the object is created.

You can use the **FETCH WIDGET OVERRIDE** routine to create a new instance of an object, based on an existing object definition in the UID file, and override values or set new values in addition to those you originally specified in the UIL module. You do not have to define the new object in the UIL module; instead, you supply the argument values in the call to the **FETCH WIDGET OVERRIDE** routine. In effect, you can use an existing object definition in the UIL module as a template, modifying the template as needed when you create additional instances of the object at run time.

When you call the **FETCH WIDGET OVERRIDE** routine, you pass a vector of name and value pairs as the **override_args** argument. The name and value pairs consist of the XUI Toolkit attribute name and an explicit value for that attribute. These name and value pairs take precedence over any arguments you specified in the UIL module. Note that since callbacks are XUI Toolkit attributes, it is also possible to override the callbacks for an object using the **FETCH WIDGET OVERRIDE** routine.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

The `FETCH WIDGET OVERRIDE` routine is useful for specifying in the application program those things that cannot be represented in UIL (such as user-defined arguments that are data structures).

You can also use the `FETCH WIDGET OVERRIDE` routine when you have to create many widgets that are very similar. Consider an application interface that has a large number of push button widgets contained in a dialog box widget. The push button widgets are the same except for their `y` position and label. Instead of declaring each push button widget individually, you can declare one push button and use the DRM routine `FETCH WIDGET OVERRIDE` to use that definition as a template, modifying the `y` position and label for each additional push button widget at run time.

When you use the `FETCH WIDGET OVERRIDE` routine in an application, you can use UIL identifiers to specify unique tag values for each callback routine. Ordinarily, the tag is specified in the callback structure and cannot be changed unless the callback is deleted and replaced. The callback structure is not stored in the widget data, but is instead stored by the intrinsics.

If you do not use identifiers for tag values, your callback routines must contain a check for the parent of the calling widget or some other field of the widget (as opposed to checking only the tag value) because it is not possible to override just the tag value with the `FETCH WIDGET OVERRIDE` routine. (Note, however, that it is possible to override the entire callback declaration given in the UID file.) If you do not use an identifier for the tag value, all instances of the multiply fetched object return identical tag values for all callbacks. If the callback routine checks only the tag value, the callback routine could not distinguish which instance made the call. Section 3.5 explains how to use UIL identifiers.

Another practical use of the `FETCH WIDGET OVERRIDE` routine is to create objects with arguments whose values can be determined only at run time (that is, values that are not known at UIL compilation time). For example, the help widget (called `help_box` in UIL) has an argument called `help_library_spec`, which is a full file specification (including the device and directory). When developing portable applications, the form and content of this file specification will vary depending on the target system when the program is compiled. Using the `FETCH WIDGET OVERRIDE` routine, you can set the value of the `help_library_spec` argument at run time when the help widget is created.

Similarly, the help widget has a `first_topic` argument, which specifies the help frame the user sees when the help widget initially appears on the screen. You can significantly improve the performance of your application by setting the value of the `first_topic` argument when the help widget is fetched rather than setting this value using the `SET VALUES` routine after the widget is created. (See Section 12.2 for information on how to construct keys for retrieving help topics.)

Example 3-27 shows how to declare an object in UIL to take advantage of the `FETCH WIDGET OVERRIDE` routine.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Example 3-27 UIL Module Setup for the FETCH WIDGET OVERRIDE Routine

```
procedure
    burger_help_proc(compound_string);
value
    k_fries_quantity : compound_string("DECburger fries_quantity");
object
    DECburger_help_box : help_box
    {
        arguments
        {
            title = k_decburger_help_title;
            default_position = true;
            cols = 55;
            application_name = k_decburger_name;
        };
        callbacks
        {
            create = procedure create_proc(k_help_box);
        };
    };
object
    fries_quantity : simple_text
    {
        .
        .
        .
        callbacks
        {
            help = procedure burger_help_proc(k_fries_quantity);
        };
    };
};
```

Example 3-28 shows the FETCH WIDGET OVERRIDE routine in a C program.

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

Example 3-28 Using the FETCH WIDGET OVERRIDE Routine in a C Program

```
❶ #ifndef VMS
#   define HELP_FILE_LOCATION      "DECW$DECBURGER"
#else
#   define HELP_FILE_LOCATION      "/usr/lib/help/decburger"
#endif

❷ static unsigned int    burger_help_proc (w, tag, somecallbackstruct)

Widget          *w;
DwtCompString   *tag;
unsigned int     *somecallbackstruct;

{
  DRMType *dummy_class;
  Arg arglist[2];
  int i = 0;

  if (widget_array[k_help_box] == 0)
  {
    XtSetArg(arglist[i], DwtNlibrarySpec,
             DwtLatin1String(HELP_FILE_LOCATION));    i++;

    ❸ XtSetArg(arglist[i], DwtNfirstTopic, tag); i++;

    if (DwtFetchWidgetOverride (
        S_DRMHierarchy,
        "DECBurger_help_box",
        parent,
        NULL,
        arglist,
        i,
        &widget_array[k_help_box],
        &dummy_class) != DRMSuccess)
    {
      printf ("DECBurger: Can't fetch help window\n");
      return 0;
    }
  }

  ❹ else
  {
    XtSetArg(arglist[0], DwtNfirstTopic, tag);
    XtSetValues(widget_array[k_help_box], arglist, 1);
  }

  ❺ if (!XtIsManaged(widget_array[k_help_box]))
    XtManageChild(widget_array[k_help_box]);
}

};
```

-
- ❶ Locations of the help library file on either VMS or ULTRIX.
 - ❷ The *burger_help_proc* routine must be added to the registration list of callback routines. The formal parameters for this routine are as follows:
 - Identifier of the widget for which the user requests help
 - Tag containing the key to the Help topic for the widget
 - The standard callback structure tag is not required by the *burger_help_proc* routine. The tag field contains all the information relevant to the **help** reason (namely, the key to the Help topic

Creating a User Interface Using UIL and DRM

3.3 Creating a User Interface at Run Time Using DRM

- ③ Set the first topic as the help widget is created.
- ④ The help widget has already been fetched, so reuse it.
- ⑤ Display the help widget.

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

UIL offers the advantage of separating the form an interface takes from the functions of the application. The form of the interface can change, while the functions the application performs remain the same. By specifying these varying forms of the interface in separate UIL modules, you can change the interface by changing the definition of the UID hierarchy (the set of UID files) in the application program and recompiling and relinking the application.

For example, you can use a UID hierarchy to provide an application interface in several languages. The text on title bars, menu widgets, and other interface objects can be displayed in the language of the end user with minimal changes to the application program. In this case, the multiple UIL modules are alternatives from which to choose at run time.

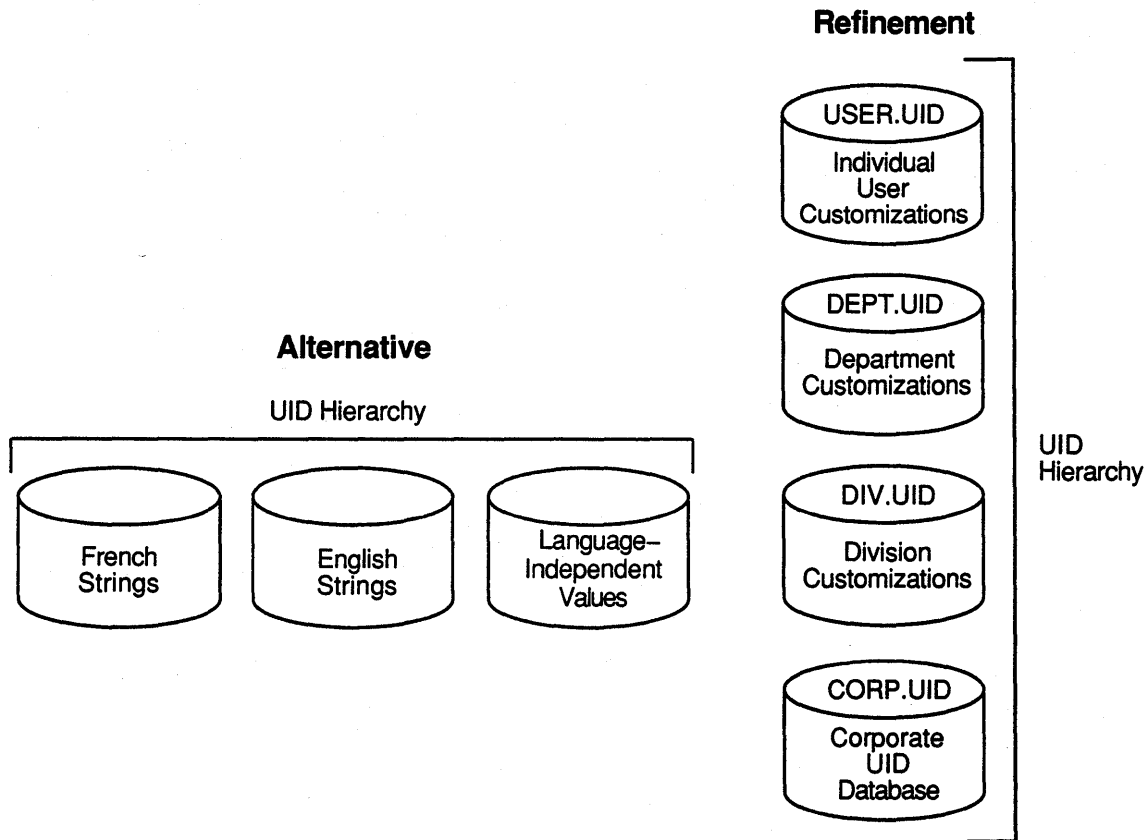
Another use of the UID hierarchy feature might be to isolate individual, department, and division customizations to a corporate-style interface by placing the customizations in separate UIL modules. In the application program, the UID hierarchy declaration would list these files in the following order: USER.UID, DEPT.UID, DIV.UID, and CORP.UID. Starting with the first file in the list, DRM searches for value definitions. If a value is defined in USER.UID (representing the user's preferences), that value is used to create the object. If a value is not specified in the USER.UID file, DRM searches for the definition in the DEPT.UID file, and so on. In this case, the multiple UIL modules represent refinements to a base interface.

Figure 3-7 shows the alternative and refinement models for DRM hierarchies.

Creating a User Interface Using UIL and DRM

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

Figure 3-7 Using UID Hierarchies to Provide Alternatives or Refinements to an Interface



ZK-0137A-GE

3.4.1 Designing an International Application Using UIL and DRM

You might need to develop a variety of interfaces for a VMS DECwindows application, particularly if the application will be used by people who speak different languages. This section describes two methods for designing an international VMS DECwindows application, using the DECburger application as an example.

In this section, the DECburger application is redesigned to support a French version of the interface in addition to the English version. To develop an international version of DECburger you must make the following changes:

- Use the following files when writing the interface specification for the international version of DECburger:

Creating a User Interface Using UIL and DRM

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

Module Name	Contents
ENGLISH.UIL	English string values appearing in the interface, declared as exported compound strings
FRENCH.UIL	French translation of these string values, declared as exported compound strings
DECBURGER_INTL.UIL	Imported string declarations (to be read from either the French or English UIL module), all other value declarations, procedure declarations, and object definitions

- Concatenate differently the strings displayed in the list box to confirm the user's order. This avoids problems with noun-adjective order.
- Use a logical name in the application to switch between the two versions of the interface. You must define this logical name to point to either the FRENCH.UID file or the ENGLISH.UID file; place the logical name before the DECBURGER_INTL.UID file in the UID hierarchy list.

These design changes are described in the following sections.

DECburger is a simple application, and therefore does not have some common widgets such as a caution box or a help box. However, more complex applications with common widgets would use two additional files to help developers internationalize applications. They are:

Module Name	Contents
DwtXLatArg.UIL	Common widgets whose labels (translatable text) are created by default. These widgets include the following: <ul style="list-style-type: none">• Caution Box• Command Window• File Selection• Help Box• Message Box
DwtXLatText.UIL	Any language string values of the widgets that appear in the DwtXLatArg.UIL.

The purpose of these two extra files is to make it easier and quicker for developers to internationalize their applications. For example, most applications use caution boxes, command windows, and help boxes. If you are creating a French version of your specific application, you could modify the text in DwtXLatText.UIL. Then you could append DwtXLatText.UIL to FRENCH.UIL, and DwtXLatArg.UIL to DECBURGER_INTL.UIL to create common French labels. You would also use FRENCH.UIL for labels that are specific to your application. Other developers who are creating French interfaces would also be able to use the same DwtXLatText.UIL and DwtXLatArg.UIL files.

Use of these two files is described in the following sections.

Creating a User Interface Using UIL and DRM

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

3.4.2 Specifying the User Interface for an International Application

As described in Section 3.4.1, the design for the international version of DECburger calls for at least three separate UIL modules.

The UIL module in Example 3-29 shows the compound string literals for the DECburger interface in French. This is a separate UIL module (called FRENCH.UIL), not an edited version of the original DECburger UIL module. There is a similar UIL module containing the English translation of these strings. The name of this UIL module is ENGLISH.UIL.

Note that the default character set, ISO_LATIN1, contains the glyphs required to represent French letters such as é and ç.

Example 3-29 French UIL Module for the International DECburger Application

```
module french_literals
  version = 'v2.0'
  names = case_sensitive

value
  k_welcome_text           : exported 'Bienvenue au DECburger';
  k_decburger_title       : exported "DECburger - Commandes";
  k_file_label_text       : exported "Fichier";
  k_quit_label_text       : exported "Quitter";
  k_quit_text             : exported "Quitter";
  k_edit_label_text       : exported "Edition";
  k_cut_dot_label_text    : exported "Couper";
  k_copy_dot_label_text   : exported "Copier";
  k_paste_dot_label_text  : exported "Coller";
  k_clear_dot_label_text  : exported "Effacer tout";
  k_select_all_label_text : exported "Sélectionner tout";
  k_order_label_text      : exported "Commande";
  k_show_controls_label_text : exported "Voir codes...";
  k_cancel_order_label_text : exported "Annuler commande";
  k_submit_order_label_text : exported "Transmettre commande";
  k_create_order_label_text : exported "Commence";
  k_dismiss_order_label_text : exported "Terminé";
  k_hamburgers_label_text : exported "Hamburgers";
  k_rare_label_text       : exported "Saignant";
  k_medium_label_text     : exported "A point";
  k_well_done_label_text  : exported "Très cuit";
  k_ketchup_label_text    : exported "Ketchup";
  k_mustard_label_text    : exported "Moutarde";
  k_onion_label_text      : exported "Oignons";
  k_mayonnaise_label_text : exported "Mayonnaise";
  k_pickle_label_text     : exported "Cornichons";
  k_quantity_label_text   : exported "Quantité";
  k_fries_label_text      : exported "Frites";
  k_size_label_text       : exported "Taille";
  k_tiny_label_text       : exported "Minuscule";
  k_small_label_text      : exported "Petit";
  k_large_label_text      : exported "Gros";
  k_huge_label_text       : exported "Enorme";
  k_drinks_label_text     : exported "Boissons";
  k_0_label_text         : exported "0";
  k_apple_juice_text      : exported "Jus de pomme";
  k_drink_list_text       : exported
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

Example 3-29 (Cont.) French UIL Module for the International DECBurger Application

```
string_table ("Jus de pomme", "Jus d'orange",
             "Jus de raisin", "Cola", "Punch",
             "Root beer", "Eau", "Ginger Ale",
             "Lait", "Café", "Thé");
k_drink_list_select      : exported string_table("Jus de pomme");
k_u_label_text          : exported "U";
k_d_label_text          : exported "D";
k_apply_label_text      : exported "Appliquer";
k_reset_label_text      : exported "Remise à 0";
k_cancel_label_text     : exported "Annulation";
k_dismiss_label_text    : exported "Terminé";

end module;
```

In the main UIL module for the international DECBurger application (called `DECBURGER_INTL.UIL`), the corresponding string literals are declared as imported compound strings. For example, the declaration for the label named `k_fries_label_text` is as follows:

```
k_fries_label_text      : imported compound_string;
```

Both the French UIL module (shown in Example 3-29) and the English UIL module (not shown) specify the corresponding values as exported and give their definitions. For example, the English definition of the `k_fries_label_text` label is "*Fries*"; the French definition is "*Frites*". You choose which of these UIL modules to use at run time as explained in Section 3.4.3.

If DECBurger were a more complex application with a caution box or a message box, the UIL module in Example 3-29 would remain essentially the same. However, if you had common labels they would be defined in `DwtXLatText.UIL`, and you would append `DwtXLatTest.UIL` to `FRENCH.UIL` so that common labels and application-specific labels would be in one file.

You would also append `DwtXLatArg.UIL`, which lists the arguments that are translatable for the common widgets, to the `DECBURGER_INTL.UIL` file.

You can find examples of `DwtXLatText.UIL` and `DwtXLatArg.UIL` in the `DECW$INCLUDE` area of your DECwindows development environment.

3.4.3 Creating the User Interface for an International Application

To create the interface for the international version of the DECBurger application, based on the redesigned UIL specification, you must make several changes to the C program. Example 3-30 shows the relevant portions of the C program for the international version of DECBurger.

Creating a User Interface Using UIL and DRM

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

Example 3-30 C Program for the International DECburger Application

```
.
.
1 static char * welcome_text_ptr;
.
.
2 static DwtCompString latin_separator;
.
.
3 static DRMResourceContextPtr resource_ctx;
.
.
static char *db_filename_vec[] =
4 { "decburger$text",
  "decburger_intl.uid",
  };
.
.
5 void get_literal (lit, ptr, compound)
char * lit;
char ** ptr;
int compound;
{
  if (compound)
    (* ptr) = DwtLatin1String( DwtDrmRCBuffer (resource_ctx) );
  else
    (* ptr) = DwtDrmRCBuffer (resource_ctx);
}
.
.
6 if (DwtDrmGetResourceContext (
  NULL, /* Allocation routine */
  NULL, /* Deallocation routine */
  100, /* Size of buffer - arbitrary value */
  & resource_ctx ) !=DRMSuccess)
  s_error ("can't get resource context");
.
.
7 Dwtfetchliteral ("k_welcome_text", & welcome_text_ptr, 0);
.
.
8 toplevel_widget = XtInitialize(welcome_text_ptr,
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

Example 3-30 (Cont.) C Program for the International DECburger Application

```
    "example",
    NULL,
    0,
    &argc,
    argv);
.
.
.
❶ Dwtfetchliteral ("k_apple_juice_text", & current_drink, 1);
   Dwtfetchliteral ("k_tiny_label_text", & current_fries, 1);
.
.
❷ Dwtfetchliteral ("k_create_order_label_text", & latin_create, 1);
   Dwtfetchliteral ("k_dismiss_order_label_text", & latin_dismiss, 1);
   latin_space = DwtLatin1String(" ");
   latin_separator = DwtLatin1String(": ");
   latin_zero = DwtLatin1String(" 0 ");
}
.
.
.
❸ static void activate_proc(w, tag, reason)
.
.
.
{
.
.
.
    switch (widget_num) {
        case k_apply:
            if (quantity_vector[k_burger_index] > 0) {
                list_txt = name_vector[k_burger_index];
                list_txt = DwtCStrcat(list_txt, latin_separator);
                sprintf(list_buffer, "%d ", quantity_vector[k_burger_index]);
                list_txt = DwtCStrcat(list_txt, DwtLatin1String(list_buffer));
                for (i = k_burger_min; i <= k_burger_max; i++)
                    if (toggle_array[i - k_burger_min]) {
                        get_something(widget_array[i], DwtNlabel, &txt);
                        list_txt = DwtCStrcat(list_txt, txt);
                        list_txt = DwtCStrcat(list_txt, latin_space);
                    }
                DwtListBoxAddItem(widget_array[k_total_order], list_txt, 0);
            }
        .
        .
        .
    }
}
```

-
- ❶ Pointer to title string. See Section 3.3.3 for details about fetching a literal value from a UID file.
 - ❷ A variable initialized to the string ": ", used for concatenation in several places throughout the application (see ❸).

Creating a User Interface Using UIL and DRM

3.4 Customizing a VMS DECwindows Interface Using UIL and DRM

- ③ Resource context required for the HGET INDEXED LITERAL call in the *get_literal* procedure (see ⑤).
- ④ Logical name for the UID file containing the strings to be displayed in the interface (either English or French) listed as the first element of the UID hierarchy array. This logical name must be defined to either ENGLISH.UID or FRENCH.UID prior to running the application.
- ⑤ Procedure to get the application title from the UID hierarchy. This procedure is described in Section 3.3.3.
- ⑥ Routine call to set up the resource context for retrieving strings.
- ⑦ Retrieve the application title string from the UID hierarchy.
- ⑧ Title string pointer is passed to the INITIALIZE routine (instead of to the actual string itself).
- ⑨ Initialize the current values of various items to match their initial values in the UID hierarchy.
- ⑩ Set up the required compound strings. The strings are fetched from the UID hierarchy in the international version.
- ⑪ Callback routine called by all push button widgets in DECburger. This routine uses the tag to determine which widget made the call, then displays the current order information in the list box widget. The difference between this version of DECburger and the original version is the manner in which the displayed strings are built. In the international version, an ordered item is displayed in the list box as follows:

Hamburgers: 2 medium

The routine gets the name of the qualifier (in this example, medium) from the widget and adds the qualifier to the displayed string. This allows orders to be displayed consistently regardless of noun-adjective order in a particular language. Note the use of the *latin_separator* literal (see ②).

Similar statements occur later in the C program to display orders for drinks and fries.

To run the international version of DECburger, follow these steps:

- 1 Compile the files FRENCH.UIL, ENGLISH.UIL, and DECBURGER_INTL.UIL.

If you had a more complex program, you would have appended the DwtXLatText.UIL file to the FRENCH.UIL or ENGLISH.UIL file before compiling.

- 2 Define the logical name DECBURGER\$TEXT to either FRENCH.UID or ENGLISH.UID, depending on the language in which you want to display the interface.
- 3 Compile, link, and run the C program.

3.5 Using Identifiers in UIL

Identifiers provide run-time binding of data to names that you specify in a UIL module. Identifiers work like global variables in a programming language.

List the names of identifiers in an identifier section in a UIL module. An identifier section consists of the reserved keyword `IDENTIFIER` followed by a list of names, with each name followed by a semicolon. You can use these names later in the UIL module as either the value of an object argument or the tag value to a callback routine. At run time, use the DRM routine `REGISTER DRM NAMES` to bind the identifier name with the data associated with the identifier. (See the *VMS DECwindows Toolkit Routines Reference Manual* for information about the `REGISTER DRM NAMES` routine.)

Since UIL has a single name space, you cannot use the name you used in a value, object, or procedure declaration as an identifier name.

Your application can successively call the routine `REGISTER DRM NAMES` with the same identifier names to supersede the value of that name for all subsequent calls to DRM that might use these identifiers. For example, you would use this procedure to change callback tags for objects created from a template definition (see Section 3.3.5).

Example 3-31 shows an identifier section in a UIL module.

Example 3-31 Using Identifiers in a UIL Module

```

MODULE id_example
  NAMES = CASE_INSENSITIVE

  IDENTIFIER
    my_x_id;
    my_y_id;
    my_focus_id;

  PROCEDURE
    my_focus_callback ( STRING );

  OBJECT my_main : MAIN_WINDOW {
    ARGUMENTS {
      x = my_x_id;
      y = my_y_id;
    };

    CALLBACKS {
      focus = PROCEDURE my_focus_callback ( my_focus_id );
    };
  };

END MODULE;
```

The UIL compiler does not do any type checking on the use of identifiers in a UIL module. Unlike a UIL value, an identifier does not have a UIL data type associated with it. You can use an identifier as an object argument or callback routine tag, regardless of the data type specified in the object or procedure declaration.

Creating a User Interface Using UIL and DRM

3.5 Using Identifiers in UIL

To reference these identifier names in a UIL module, use the name of the identifier wherever you want its value to be used. The value is determined at run time. The UIL module in Example 3-31 shows identifiers used as argument values and callback routine tags. However, you can reference a identifier in any context where you can reference a value.

The identifiers *my_x_id* and *my_y_id* are used as argument values for the main window widget, *my_main*. The position of the main window widget may depend on the screen size of the terminal on which the interface is displayed. Using identifiers, you can provide the values of *x* and *y* at run time.

The identifier *my_focus_id* is specified as the tag to the callback routine *my_focus_callback*. In the application program, you could allocate a data structure and use *my_focus_id* to store the address of that data structure. When the **focus** reason occurs, the data structure is passed as the tag to routine *my_focus_callback*.

3.6

Using Symbolic References to Widget Identifiers in UIL

The UIL compiler allows you to refer to a widget identifier symbolically by using its name. This mechanism addresses the problem that the UIL compiler views objects by name and the XUI Toolkit views objects by widget identifier. Widget identifiers are defined at run time and are therefore unavailable for use in a UIL module.

When you need to supply an argument that requires a widget identifier, you can use the UIL name of that widget and its object type as the argument. For example, the menu bar widget has an argument **DwtNMenuHelpWidget** that expects the identifier of a widget (a pull-down menu entry widget, for instance). You can give the name and object type of the pull-down menu entry widget as the value for this argument. Another practical use of a symbolic reference is to specify the default push button widget (in a dialog box widget or radio box widget).

Note: To specify a symbolic reference completely in UIL, you must include the object type with the object name.

Example 3-32 shows the use of a symbolic reference.

Creating a User Interface Using UIL and DRM

3.6 Using Symbolic References to Widget Identifiers in UIL

Example 3-32 Using Symbolic References in a UIL Module

```
MODULE symbolic_ref_example
  NAMES = CASE_INSENSITIVE

  OBJECT my_dialog_box : DIALOG_BOX {
    ARGUMENTS {
      default_button = PUSH_BUTTON yes_button;
    };
    CONTROLS {
      PUSH_BUTTON yes_button;
      PUSH_BUTTON no_button;
    };
  };

  OBJECT yes_button : PUSH_BUTTON {
    ARGUMENTS {
      label_label = 'yes';
    };
  };

  OBJECT no_button : PUSH_BUTTON {
    ARGUMENTS {
      label_label = 'no';
    };
  };

END MODULE;
```

In Example 3-32, two push button widgets are defined as *yes_button* and *no_button*. In the definition of the dialog box widget, the name *yes_button* is given as the value for the **default_button** argument. Usually, the default push button argument accepts a widget identifier. When you use a symbolic reference (the object type and name of the *yes_button* widget) as the value for the default push button argument, DRM substitutes the widget identifier of the *yes_button* push button widget for its name at run time.

There is a restriction on the use of symbolic references: the object name you reference must be a descendant of the object being fetched in order for DRM to find the referenced object; you cannot reference an arbitrary object. DRM checks this at run time.

The UIL built-in tables listed in an appendix in the *VMS DECwindows User Interface Language Reference Manual* indicate where symbolic referencing of widget identifiers is acceptable by showing the term *object reference* as the type of an argument.

3.7 Developing and Testing Prototypes Using UIL

UIL allows you to separate the form and function of a VMS DECwindows application. Because changes in the representation or layout of the interface do not require changes to the application program, you can quickly see the impact of design changes on the interface. Once you have in place the standard XUI Toolkit routine calls to create, manage, and realize the interface, you can change the interface design by editing the UIL module, recompiling only the UIL module, and rerunning the application program.

Creating a User Interface Using UIL and DRM

3.7 Developing and Testing Prototypes Using UIL

The direct manipulation semantics of a VMS DECwindows interface (that is, the appearance and behavior of the interface when the end user interacts with it) are built into the XUI Toolkit objects themselves. When you present the end user with an interface prototype, the end user immediately gets the look and feel of the interface. For example, when the user clicks on a push button widget, the highlighting feedback occurs automatically. This reaction to manipulation by the user does not require application routines.

The combination of these features (the separation of form and function and the built-in look and feel of interface objects) can significantly shorten the time required to develop a VMS DECwindows application. Interface designers and application programmers can work essentially independently (and, therefore, concurrently) without relying on one another to finish.

Eventually, the interface and the functional routines are brought together and tested as a unit. The DECburger application demonstrates a useful technique you can use to test whether the callback routines in the application are correctly registered with DRM, and whether the routines are called correctly in response to the user's interaction with the interface. This technique does not require all the functional routines to be in place, so it is particularly useful during the prototyping phase.

In the DECburger application, the callback routine named *activate_proc* is used to exercise the callbacks for features that are not yet implemented in the application program. The *activate_proc* routine displays a message box widget bearing the message "Feature is not yet implemented" whenever the user activates one of the nonfunctional features. Section 3.7.1 explains what you need to do in UIL to use this prototype testing technique, and Section 3.7.2 explains what you need to do in the application program.

3.7.1 Setting Up the UIL Module for Prototype Testing

To use the prototype testing technique demonstrated in the DECburger application, you need to declare the following in the UIL module:

Resource Name in DECburger	Declaration
<code>activate_proc</code>	Routine to be called when the user activates an interface object
<code>k_nyi</code>	Callback tag that will be passed to this routine
<code>nyi</code>	Message box to be fetched and displayed when the routine is called with the callback tag
<code>k_nyi_label_text</code>	String literal to define the message box label

All objects not fully implemented in DECburger (for example, the operations on the Edit pull-down menu widget) use this technique. Example 3-33 shows these declarations in the DECburger UIL module.

Creating a User Interface Using UIL and DRM

3.7 Developing and Testing Prototypes Using UIL

Example 3-33 Declarations in the DECburger UIL Module for Prototype Testing

```
procedure
.
.
.
① activate_proc (integer);
.
.
value
② k_nyi          : 5;
.
.
value
③ k_nyi_label_text
      : compound_string("Feature is not yet implemented");
.
.
object
④ nyi : message_box {
      arguments {
⑤       label_label = k_nyi_label_text;
          default_position = true;
      };
      callbacks {
⑥       create = procedure create_proc (k_nyi);
      };
};
.
.
.
```

- ① Declares the activate routine (*activate_proc*) and specifies that the routine must be passed an integer when called.
- ② Declares an integer literal, named *k_nyi*, to be used as the callback tag passed to the activate routine (see Example 3-34).
- ③ Declares a compound string literal, named *k_nyi_label_text*, to be used to specify the label of the message box widget that is fetched and displayed when the activate routine is called (see ⑤).
- ④ Declares an instance of a message box widget to be fetched and displayed when the user activates an object that does not have functional code in the application.
- ⑤ Value of the string literal declared in ③ will be used as the label of the message box widget.
- ⑥ Here, *k_nyi* is used as the callback tag to the creation routine, *create_proc*, to identify the message box widget as the widget that is being created (note that the name of the message box widget is *nyi*). Do not confuse this with the use of *k_nyi* as the callback tag passed to the activate routine (see Example 3-34).

Creating a User Interface Using UIL and DRM

3.7 Developing and Testing Prototypes Using UIL

Example 3-34 shows the definition of the push button widget associated with the Copy operation on the Edit menu widget. The Copy operation in DECburger is not implemented. Note that the *k_nyi* callback tag is passed to the activate routine for this push button widget. When the user clicks on the Copy operation, the message box widget pops up, displaying the "Feature is not yet implemented" message. The example in Section 3.7.2 shows the definition of the activate routine.

Example 3-34 Declaring an Unimplemented Object in the DECburger UIL Module

```
object
  m_copy_button : push_button {
    arguments {
      label_label = k_copy_dot_label_text;
    };
    callbacks {
      activate = procedure activate_proc (k_nyi);
    };
  };
```

3.7.2 Setting Up the Application Program for Prototype Testing

In the C program for the DECburger application, the *k_nyi* callback and the activate routine are defined as shown in Example 3-35. All push button widgets in the DECburger application call back to this routine. Ordinarily, the callback tag identifies which widget made the call. In the case where the user selects an unimplemented feature, the callback tag causes the application to display the "Feature is not yet implemented" message.

Example 3-35 Definition of the Activate Routine in the DECburger Application

```
.
.
#define k_nyi 5
.
.
static void activate_proc(w, tag, reason)
  Widget w;
  int *tag;
  unsigned long *reason;
{
  ① int widget_num = *tag;
  int i, value, fries_num;
  char *txt, *fries_text, *list_txt, list_buffer[20];
  switch (widget_num) {
    ② case k_nyi:
      ③ if (widget_array[k_nyi] == NULL)
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.7 Developing and Testing Prototypes Using UIL

Example 3–35 (Cont.) Definition of the Activate Routine in the DECburger Application

```
        {  
            if (DwtFetchWidget(s_DRMHierarchy, "nyi", toplevel_widget,  
                &widget_array[k_nyi], &dummy_class) != DRMSuccess) {  
                s_error("can't fetch nyi widget");  
            }  
        }  
  
④ XtManageChild(widget_array[k_nyi]);  
    break;  
    .  
    .  
    .
```

- ① Converts the tag to a widget number.
- ② Sends a message when the user activates a push button widget associated with a nonfunctional feature.
- ③ Fetches the message box widget from the UID file the first time the activate routine is called with the *k_nyi* tag. Once the message box has been fetched, it will be redisplayed (but not re-created) upon subsequent calls with this tag.
- ④ Pops up the message box widget saying "Feature is not yet implemented".

3.8 Using UIL on Large Projects

When several programmers are working together to specify the interface for a VMS DECwindows application, competition for access to the UIL module can develop. Access competition can be eased if the UIL module is broken up into several small files, with each containing a segment of the total interface specification.

One approach to breaking up the UIL module is to construct a main UIL file containing the following information.

- Comments describing copyright information, module history, project information, and other relevant information.
- Global declarations, such as case sensitivity, objects clause, and procedure declarations.
- A series of INCLUDE FILE statements (include directives). Each include directive points to a UIL specification file containing some portion of the interface specification.

Once you create a main UIL file, you should rarely need to change its contents.

Example 3–36 shows a sample main UIL file. Note that there is no technical reason to divide the user interface specification as in this example. The purpose of using multiple UIL files here is simply to make

Creating a User Interface Using UIL and DRM

3.8 Using UIL on Large Projects

it easier for large programming project teams to work concurrently on the same application interface.

Example 3-36 Sample Main UIL File

```
module big_project
  version = '2.0'

!*****
!*
!*  COPYRIGHT (c) 1990 BY
!*  XYZ CORPORATION
!*  ALL RIGHTS RESERVED.
!*
!******
!
!++
!
!  CREATION DATE:  19-Apr-1990
!
!  MODIFIED BY:
!    04/19/90    JMK Create this main UIL file.
!    04/19/90    VPR Add some context-sensitive help text.
!
!++
!  NOTE:  This file includes several other UIL specification files
!         that collectively specify the user interface for BIG_PROJECT.
!--

names = case_insensitive

!++
!  These are the callback routines for the big_project application.
!--

procedure
  BPROJ$Create_Callback_Routine    (integer);
  BPROJ$Destroy_Callback_Routine   (integer);
  BPROJ$Help_Callback_Routine      (integer);
  BPROJ$Universal_Callback_Routine (integer);

!++
!  The following file contains value definitions (the "k_..." literals).
!--

include file 'lib$:bprojlits';

!++
!  The following files contain segments of the big_project
!  application interface.
!
!  These files are self-contained and do not have any include directives.
!--

include file 'lib$:bprojwindow';    ! Defines windows and pull-down menus.
include file 'lib$:bprojdialog';    ! Defines dialog boxes
include file 'lib$:bprojother';     ! Defines miscellaneous objects such as
                                     ! caution boxes and pop-up menus.

end module;
```

Creating a User Interface Using UIL and DRM

3.8 Using UIL on Large Projects

In Example 3–36, the UIL specification for an application interface is divided into the following files:

- Shared literals

The first included file defines all literals shared between the UIL module and the application source code. These are the constants used as tags to the callback routines (see Section 3.2.1).

- Main window widget

The second included file defines the main window widget for the application. This might include a menu bar widget with associated pull-down menu entry widgets, the work region, and other relevant pieces.

- Dialog box widgets

The third included file defines all the dialog box widgets used in the application.

- Other interface objects

The fourth included file defines all the other objects that do not fit into the first three categories. This file might include display windows with their menu bar widgets and work regions, pop-up menu widgets, and the command dialog box widget.

It is a matter of style whether the included files themselves contain include directives. Some programmers prefer to work with a single main UIL file and know that this file names all of the remaining files needed to complete the interface specification. Having a list of all needed files visible in the main UIL file can be helpful, for example, to someone translating the user interface into another language. All files can be accounted for easily and included in the translation.

You can further simplify the translator's job by isolating in a separate include file all items that vary visually (for example, strings, x- and y-coordinates, and width and height attributes) as a result of translation. In this way, the translator can find in a single file all the values that need to be translated. (Declare these items as values in the separate file and reference the values in the object declarations in the primary UIL module.)

3.9 Working with User-Defined Widgets in UIL

You can extend the XUI Toolkit by building your own widgets. In UIL, a widget you build yourself is called a **user-defined widget** and is identified by the UIL object type `user_defined`. A user-defined widget can accept any UIL built-in argument or callback reason. If needed, you can use UIL to define your own arguments and callback reasons for a user-defined widget. You can specify any object as a child of a user-defined widget.

To use a user-defined widget in an application interface, follow these steps in the UIL module:

- 1 Define the arguments and callback reasons for the user-defined widget that are not UIL built-ins. This can be done in line when declaring an

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

instance of the user-defined widget or in one or more value sections (as shown in Example 3–37).

- 2 Declare the creation routine for the user-defined widget.
- 3 Declare an instance of the user-defined widget. Use `user_defined` as the object type and include the name of the widget creation routine in the declaration.

In the application program, you must register the class of the user-defined widget using the DRM routine `REGISTER CLASS`. Part of the information you provide to the `REGISTER CLASS` routine is the name of the widget creation routine. By registering the class (and creation routine), you allow DRM to create a user-defined widget using the same mechanisms used to create XUI Toolkit objects. You can specify the widget using UIL and fetch the widget with DRM.

The examples in this section are based on a previously built user-defined widget called the XYZ Widget. (Appendix D explains how to build a user-defined widget.) The remainder of this section explains how to include the XYZ Widget in an application interface using UIL and how to create the widget at run time using DRM.

3.9.1 Defining Arguments and Reasons for a User-Defined Widget

The UIL compiler has built-in arguments and callback reasons that are supported by objects in the XUI Toolkit. A user-defined widget can be built having only standard XUI Toolkit arguments and reasons as its resources. If your application interface uses a user-defined widget of this type, you can use the UIL built-in argument names and callback reasons directly when you declare an instance of the user-defined widget. If the user-defined widget supports arguments and reasons that are not built into the UIL compiler, you need to define these arguments and reasons using the `ARGUMENT` and `REASON` functions, respectively, before specifying them.

Example 3–37 shows a UIL specification file that defines arguments and callback reasons, and declares the creation routine, for the XYZ Widget. This UIL specification file should be included in any UIL module in which you declare an instance of the XYZ Widget.

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

Example 3-37 User-Defined XYZ Widget

```
① value
  xyz_font_level_0 :      argument ('fontLevel0' , font);
  xyz_font_level_1 :      argument ('fontLevel1' , font);
  xyz_font_level_2 :      argument ('fontLevel2' , font);
  xyz_font_level_3 :      argument ('fontLevel3' , font);
  xyz_font_level_4 :      argument ('fontLevel4' , font);
  xyz_indent_margin :     argument ('indentMargin' , integer);
  xyz_unit_level :        argument ('unitLevel' , integer);
  xyz_page_level :        argument ('pageLevel' , integer);
  xyz_root_widget :       argument ('rootWidget' , integer );
  xyz_root_entry :        argument ('rootEntry' , integer);
  xyz_display_mode :      argument ('displayMode' , integer);
  xyz_fixed_width_entries : argument ('fixedWidthEntries' , boolean);

② value
  xyz_select_and_confirm : reason ('selectAndConfirmCallback');
  xyz_extend_confirm :     reason ('extendConfirmCallback');
  xyz_entry_selected :     reason ('entrySelectedCallback');
  xyz_entry_unselected :   reason ('entryUnselectedCallback');
  xyz_help_requested :     reason ('helpCallback');
  xyz_attach_to_source :   reason ('attachToSourceCallback');
  xyz_detach_from_source : reason ('detachFromSourceCallback');
  xyz_alter_root :         reason ('alterRootCallback');
  xyz_selections_dragged : reason ('selectionsDraggedCallback');
  xyz_get_entry :          reason ('getEntryCallback');
  xyz_dragging :           reason ('draggingCallback');
  xyz_dragging_end :       reason ('draggingEndCallback');
  xyz_dragging_cancel :    reason ('draggingCancelCallback');

③ value
  XyzPositionTop :      1;
  XyzPositionMiddle :   2;
  XyzPositionBottom :   3;

  XyzDisplayOutline :   1;
  XyzDisplayTopTree :   2;

④ procedure XyzLowLevelCreate();
```

- ① Defines UIL argument names for the XYZ Widget that are *not* built-in XUI Toolkit arguments. The strings you pass to the ARGUMENT function must match the names listed in the resource list structure in the widget class record for the XYZ Widget. (Section D.2.2 describes the contents of the widget class record.)

In addition to the string, specify the data type of the argument. Just as for built-in arguments, when you declare an instance of the XYZ Widget in a UIL module, the UIL compiler checks the data type of the values you specify for these arguments. For example, the UIL compiler checks that the value you specify for the *xyz_indent_margin* argument is an integer.

- ② Defines the XYZ Widget's callback reasons that are *not* UIL built-in reasons. The strings you pass to the REASON function must match the names listed in the resource list structure in the widget class record for the XYZ Widget. (Callback reasons, like UIL arguments, are considered to be widget-specific attributes in the XUI Toolkit and are defined as resources.)

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

- ③ Defines some integer literals for specifying arguments of the XYZ Widget. These literals have names configured in the MIT C binding style. In the VAX binding style, the names of these integer literals would be configured as follows:

```
XYZ$C_POSITION_TOP  
XYZ$C_POSITION_MIDDLE  
XYZ$C_POSITION_BOTTOM  
XYZ$C_DISPLAY_OUTLINE  
XYZ$C_DISPLAY_TOP_TREE
```

- ④ Declares the widget creation routine for the XYZ Widget. This creation routine is registered with DRM through the REGISTER CLASS routine (see Example 3-39).

3.9.2 Using a User-Defined Widget in an Interface Specification

Example 3-38 shows how to specify the XYZ Widget in a UIL module. This UIL module includes the UIL specification file shown in Example 3-37 as XYZ_WIDGET.UIL.

Example 3-38 Declaring the User-Defined XYZ Widget in a UIL Module

```
module xyz_example  
    names = case_sensitive  
  
    include file 'decw$include:DwtAppl.uil';  
① include file 'xyz_widget.uil';  
② procedure  
    XyzAttach      ();  
    XyzDetach      ();  
    XyzExtended    ();  
    XyzConfirmed   ();  
    XyzGetEntry    ();  
    XyzSelected    ();  
    XyzUnselected  ();  
    XyzDragged     ();  
    XyzDragging    ();  
    XyzDraggingEnd ();  
    create_proc    ();  
    MenuQuit       ();  
    MenuExpandAll  ();  
    MenuCollapseAll ();
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

Example 3-38 (Cont.) Declaring the User-Defined XYZ Widget in a UIL Module

```
③ object
    main : main_window
        { arguments
            {
                x = 0;
                y = 0;
                height = 0;
                width = 0;
            };
        controls
        { menu_bar main_menu;
          user_defined xyz_widget;
        };
    };

④ xyz_widget : user_defined procedure XyzLowLevelCreate
    { arguments
        {
            x = 0;
            y = 0;
            height = 600;
            width = 400;
            xyz_display_mode = XyzDisplayOutline;
        };
    callbacks
    { xyz_attach_to_source = procedure XyzAttach();
      xyz_detach_from_source = procedure XyzDetach();
      xyz_get_entry = procedure XyzGetEntry();
      xyz_select_and_confirm = procedure XyzConfirmed();
      xyz_extend_confirm = procedure XyzExtended();
      xyz_entry_selected = procedure XyzSelected();
      xyz_entry_unselected = procedure XyzUnselected();
      xyz_selections_dragged = procedure XyzDragged();
      xyz_dragging = procedure XyzDragging();
      xyz_dragging_end = procedure XyzDraggingEnd();
    };
    create = procedure create_proc();
    };

⑤ };

⑥ main_menu: menu_bar
    { arguments
        { orientation = DwtOrientationHorizontal;
        };
    controls
    { pulldown_entry file_menu;
    };
    };

⑦ };
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

Example 3-38 (Cont.) Declaring the User-Defined XYZ Widget in a UIL Module

```
file_menu: pulldown_entry
{ arguments
  { label_label = 'File';
  };
controls
  { pulldown_menu
    { controls
      { push_button expand_all_button;
      push_button collapse_all_button;
      push_button quit_button;
      };
    };
  };
};

expand_all_button: push_button
{ arguments
  { label_label = "Expand All";
  };
callbacks
  { activate = procedure MenuExpandAll();
  };
};

collapse_all_button: push_button
{ arguments
  { label_label = "Collapse All";
  };
callbacks
  { activate = procedure MenuCollapseAll();
  };
};

quit_button: push_button
{ arguments
  { label_label = "Quit";
  };
callbacks
  { activate = procedure MenuQuit();
  };
};

end module;
```

- 1 Include directive to include the definition of the XYZ Widget shown in Example 3-37.
- 2 Declarations for the callback routines defined in the application program.
- 3 Declaration for the main window widget. The main window widget has two children: a menu bar widget and the XYZ Widget.
- 4 Declaration for the XYZ Widget. Note that the object type is `user_` defined and that the creation routine, `XYZLowLevelCreate`, is included in the declaration.

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

- ⑤ The `xyz_display_mode` argument, defined with the `ARGUMENT` function in Example 3-37, is specified using one of the integer literals also defined in that example.
- ⑥ All widgets support the `create` reason.
- ⑦ The remaining objects declarations comprise the menu bar widget and its pull-down menu widgets.

3.9.3 Accessing a User-Defined Widget at Run Time

Example 3-39 shows a C application program that displays the XYZ Widget (defined in Example 3-37 and declared in Example 3-38).

Example 3-39 C Program for Displaying the XYZ User-Defined Widget

```
#include <decw$include/DwtAppl.h>
① #include <decw$include/DECwWsXyz.h>
② globalref int xyzwidgetclassrec;
③ extern void XyzAttach      ();
   extern void XyzDetach    ();
   extern void XyzGetEntry  ();
   extern void XyzConfirmed ();
   extern void XyzExtended  ();
   extern void XyzSelected  ();
   extern void XyzUnselected ();
   extern void XyzHelpRoutine ();
   extern void XyzDragged   ();
   extern void XyzDragging  ();
   extern void XyzDraggingEnd ();
   extern void create_proc  ();
   extern void MenuQuit     ();
   extern void MenuExpandAll ();
   extern void MenuCollapseAll ();

④ static DRMRegisterArg register_vector[] =
{
  { "XyzAttach",      (caddr_t) XyzAttach },
  { "XyzDetach",     (caddr_t) XyzDetach },
  { "XyzGetEntry",   (caddr_t) XyzGetEntry },
  { "XyzConfirmed",  (caddr_t) XyzConfirmed },
  { "XyzExtended",   (caddr_t) XyzExtended },
  { "XyzSelected",   (caddr_t) XyzSelected },
  { "XyzUnselected", (caddr_t) XyzUnselected },
  { "XyzHelpRoutine", (caddr_t) XyzHelpRoutine },
  { "XyzDragged",    (caddr_t) XyzDragged },
  { "XyzDragging",   (caddr_t) XyzDragging },
  { "XyzDraggingEnd", (caddr_t) XyzDraggingEnd },
  { "create_proc",   (caddr_t) create_proc },
  { "MenuQuit",      (caddr_t) MenuQuit },
  { "MenuExpandAll", (caddr_t) MenuExpandAll },
  { "MenuCollapseAll", (caddr_t) MenuCollapseAll }
};
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

Example 3-39 (Cont.) C Program for Displaying the XYZ User-Defined Widget

```
#define register_vector_length ( (sizeof register_vector) / \
                                (sizeof register_vector[0]))

5 static DRMHierarchy    hierarchy_id ;
   static char          *vec[]={"xyz_example.uid"};
   static DRMCode      class ;

   Widget toplevel;
   Widget mainwindow;

6 int main (argc, argv)
   unsigned int argc;
   char **argv;
{
7   Arg arguments[1];
8   DwtInitializeDRM();
9   if( DwtRegisterClass
       ( DRMwcUnknown,
         XyzClassName,
         "XyzLowLevelCreate",
         XyzLowLevelCreate,
         &xyzwidgetclassrec )
       != DRMSuccess )
   {
   printf ("Can't register XYZ widget");
   }
10  toplevel = XtInitialize ("xyz", "xyz", NULL, 0, &argc, argv);
11  if( DwtOpenHierarchy
       ( 1,
         vec,
         NULL,
         &hierarchy_id )
       != DRMSuccess )
   {
   printf ("Can't open hierarchy");
   }
12  DwtRegisterDRMNames( register_vector, register_vector_length );
   XtSetArg (arguments[0], XtNallowShellResize, TRUE);
   XtSetValues (toplevel, arguments, 1);

13  if( DwtFetchWidget
       ( hierarchy_id,
         "main",
         toplevel,
         &mainwindow,
         &class )
       != DRMSuccess )
   {
   printf ("Can't fetch interface ");
   }

   XtManageChild (mainwindow);
   XtRealizeWidget (toplevel);
```

(continued on next page)

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

Example 3-39 (Cont.) C Program for Displaying the XYZ User-Defined Widget

```
XtMainLoop();  
return (0);  
}  
.  
.  
.
```

- ❶ Includes XYZ declarations. (See Section D.3.3 for information on widget subclassing in private .h files.)
- ❷ Provides a reference to the widget class record for the XYZ Widget (named *xyzwidgetclassrec*). (Section D.2.1.1 explains how to construct a widget class record for a user-defined widget.)
- ❸ Declares callback routines defined (but not shown) later in the program.
- ❹ Defines the mapping between UIL procedure names and their addresses.
- ❺ Specifies the UID hierarchy list. The UID hierarchy for this application consists of a single UID file, the compiled version of XYZ_EXAMPLE.UIL. (Assume the UIL specification file has the same name as the UIL module; see the module header in Example 3-38. The file named XYZ_EXAMPLE.UIL includes the file XYZ_WIDGET.UIL, shown in Example 3-37.)
- ❻ Main routine.
- ❼ Arguments for the widgets.
- ❽ Initializes DRM.
- ❾ Registers the XYZ widget class with DRM. This allows DRM to use standard creation mechanisms to create the XYZ Widget (see ❸). The arguments passed to the REGISTER CLASS routine are as follows:
 - DRMwcUnknown—Indicates that class is user-defined
 - XyzClassName—Class name of XYZ widget, defined in DECwWsXyz.h.
 - "XyzLowLevelCreate"—Name of the low-level creation routine
 - XyzLowLevelCreate—Address of the low-level creation routine
 - &xyzwidgetclassrec—Pointer to the widget class record
- ❿ Initializes the XUI Toolkit.
- ⓫ Defines the UID hierarchy.
- ⓬ Registers callback routine names with DRM.

Creating a User Interface Using UIL and DRM

3.9 Working with User-Defined Widgets in UIL

- Fetches the interface (the main window widget with a menu bar widget and the XYZ Widget in the work area). Note that the XYZ Widget is treated like any XUI Toolkit widget. DRM calls the XYZ Widget's low-level creation routine (*XYZLowLevelCreate*) and passes this routine the values for the **x**, **y**, **width**, **height**, and **xyz_display_mode** arguments as specified in the UID file, using the standard low-level routine format.

Note: In cases where one widget will not allow another type of widget to be its child, you can declare as "user-defined" the widget that you want to be a child. For example, menu bars do not allow attached dialog boxes as children. However, if you want to make the attached dialog box the child of a menu bar, declare the attached dialog box as a "user-defined" widget.

4 Creating a Main Window Widget

This chapter provides the following:

- An overview of the main window widget in the XUI Toolkit
- A detailed description of how to use the main window widget in an application

In addition, this chapter describes the three other window widgets closely related to the main window widget:

- Command window widget
- Scroll window widget
- Window widget

4.1 Overview of Window Widgets

The first task your application program must perform is to create a window on the display. Windows are the way your application communicates with a user.

While all widgets create a window on a display, you typically base your application widget hierarchy on one window, called a **main window**. The main window presents all the primary functions of your application. In addition, the main window usually provides a blank work area you can fill in any way appropriate to your application. To provide these capabilities, the XUI Toolkit includes the main window widget.

Other XUI Toolkit widgets provide blank areas, such as the dialog box widgets (see Chapter 7). However, the main window widget provides services these other widgets do not. For example, the main window widget is the only XUI Toolkit widget that can automatically manage the wrapping of a menu bar widget when necessary.

4.2 Children of a Main Window Widget

A main window widget can have any number of child widgets; however, only five of the managed children can be visible at any one time. Based on widget type, the main window widget places each visible child within its borders to create a standard layout. The following lists the five widgets that can be visible children of a main window widget (all of these widgets are optional):

- Menu bar widget
- Command window widget
- Horizontal scroll bar widget

Creating a Main Window Widget

4.2 Children of a Main Window Widget

- Vertical scroll bar widget
- Work area widget

4.2.1 Menu Bar Widget

A menu bar widget allows you to present a list of choices to the user. Many applications use menu bar widgets to provide access to basic functions, such as exiting, copying, and cut and paste. For this reason, menu bar widgets are often used with the main window widget. See Section 6.5 for more information about the menu bar widget.

The main window widget places the menu bar widget at the top of the main window widget's window. By default, the main window widget sizes the menu bar widget so that its width extends across the entire window. The menu bar widget determines its height by what it needs to display the choices it contains.

4.2.2 Command Window Widget

The command window widget provides users of your application with the ability to enter commands on a command line using a keyboard. The command widget contains a text entry area in which users of your application can enter commands as text strings. Previously entered commands can be recalled and edited. By default, the command window displays the last two commands in a display area above the text entry area. You can specify that more than two lines of command history appear in this display. Section 4.6 provides more information about the command window widget.

The main window widget places the command window widget at the bottom of the main window widget's window. By default, the main window widget sizes the command window widget so that its width extends across the entire window. Once the main window widget has been realized, you cannot alter the height of the command window widget.

4.2.3 Scroll Bar Widgets

The scroll bar widgets enable users to view areas of the work area widget that are not currently visible. The work area widget may not be able to fit its entire contents into the size provided by the layout of the main window widget. In this case, you can include scroll bar widgets in your main window widget. Section 10.4 describes how to create scroll bar widgets.

The main window widget places the horizontal scroll bar widget just above the command window widget. If there is no command window widget, the horizontal scroll bar appears at the bottom of the main window widget. You can only specify the height of the horizontal scroll bar widget. The main window widget determines the width of the horizontal scroll bar widget so that it extends across the entire window. If the main window widget includes a vertical scroll bar widget, the width of the horizontal scroll bar widget is the width of the main window widget minus the width of the vertical scroll bar.

Creating a Main Window Widget

4.2 Children of a Main Window Widget

The main window widget places the vertical scroll bar widget on the right edge of the main widget window widget. The vertical scroll bar appears below the menu bar widget and above the command window widget, if either of these widgets is present. You can only specify the width of a vertical scroll bar widget. The main window widget determines the height of the vertical scroll bar widget in relation to the height of the work area window. If the main window widget includes a horizontal scroll bar widget, the height of the vertical scroll bar widget is adjusted by the height of the horizontal scroll bar widget.

4.2.4 Work Area Widget

The work area widget child comprises the remainder of the main window widget. The main window widget places the work area widget in the area under the menu bar widget, to the left of the vertical scroll bar widget, and above the horizontal scroll bar widget, if any of these widgets is present. You can specify both height and width dimensions of the work area widget.

For example, you can make a dialog box widget the work area widget of a main window widget. You can then add as many children as you want to the dialog box widget.

The scroll window widget is commonly used as a work area widget because it can automatically update the size of the slider in the scroll bar widget. The slider represents the portion of the work area widget that is currently visible. If you use a scroll window widget, you do not have to have scroll bar widgets as children of the main window widget. Section 4.4 describes the scroll window widget.

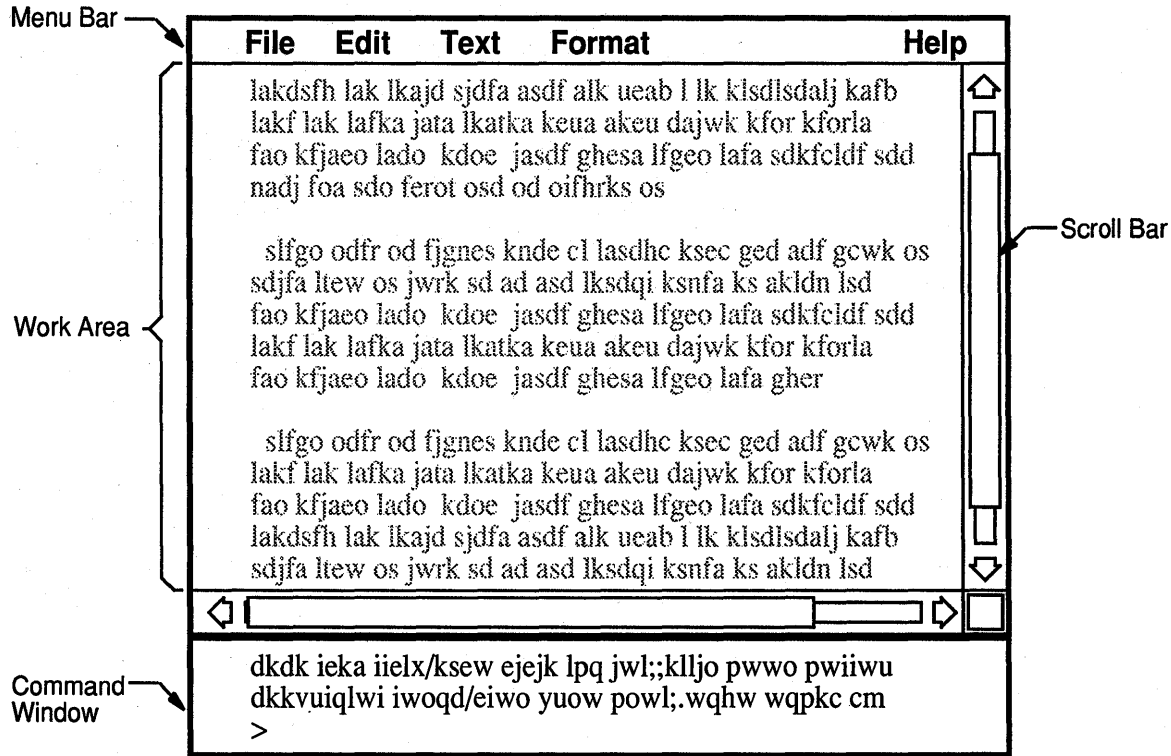
The window widget is another widget that can be used as a work area widget. The window widget is an empty rectangle that places no restrictions on what it contains. The window widget is the only XUI Toolkit widget that supports graphics operations.

Figure 4-1 illustrates the layout of a main window widget.

Creating a Main Window Widget

4.2 Children of a Main Window Widget

Figure 4-1 Main Window Widget



ZK-0405A-GE

4.3 Creating a Main Window Widget

To create a main window widget, perform the following steps:

- 1 Create the main window widget.

Use any of the widget creation mechanisms listed in Table 4-1. The choice of mechanism depends on the attributes you need to access.

Table 4-1 Main Window Widget Creation Mechanisms

High-level routine	Use the MAIN WINDOW routine to create a main window widget.
Low-level routine	Use the MAIN WINDOW CREATE routine to create a main window widget.
UIL object type	Use the main_window object type to define a main window widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

Creating a Main Window Widget

4.3 Creating a Main Window Widget

2 Create the child widgets of the main window widget.

You create the child widgets specifying the main window widget as their parent. For information about specifying where you want the main window widget to place the children, see Section 4.3.1.

3 Manage the child widgets of the main window widget.

Use the `MANAGE CHILD` or the `MANAGE CHILDREN` intrinsic routine to manage children of the main window widget. In UIL this step is not necessary, since by default the DRM routine `FETCH WIDGET` manages the widgets it creates at run time.

4 Manage the main window widget.

Use the intrinsic routine `MANAGE CHILD` to manage the main window widget. In UIL this step is not necessary, since by default the DRM routine `FETCH WIDGET` manages the widgets it creates at run time.

After completing these steps, if the parent of the main window widget has been realized, the main window widget will appear on the display.

Low-level routines and UIL provide access to the complete set of widget attributes at creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available in a high-level routine, use the `SET VALUES` intrinsic routine after the widget has been created.) The *VMS DECwindows Toolkit Routines Reference Manual* describes the complete list of attributes supported by the main window widget. Table 4-2 lists the attributes available using the high-level routine `MAIN WINDOW`. Pass the values for these attributes as arguments to the high-level routine.

Table 4-2 Widget Attributes Accessible Using the High-Level Routine `MAIN WINDOW`

x	The x-coordinate of the upper left corner of the widget
y	The y-coordinate of the upper left corner of the widget
width	The width of the widget
height	The height of the widget

4.3.1 Adding Children to a Main Window Widget

You can add children to a main window widget in three ways:

- Use the `SET VALUES` intrinsic routine
- Use the `MAIN WINDOW SET AREAS` routine
- Accept the defaults of the main window widget

Creating a Main Window Widget

4.3 Creating a Main Window Widget

4.3.1.1 Using SET VALUES to Add Children to a Main Window Widget

The main window widget supports attributes that identify the child widget to be used for each of its five designated areas (described in Section 4.2). However, you cannot set these attributes at widget creation time because you do not know the widget identifier of the child until you create the children. You can only set these attributes after the main window widget has been created.

As with any widget attribute, you can use the SET VALUES intrinsic routine to assign values to these attributes after the widget has been created. Specify the widget identifier of the child widget as the value of these attributes. Table 4-3 lists these attributes of the main window widget.

Table 4-3 Child Widget Attributes of the Main Window Widget

Attribute	Value
command_window	The widget identifier of the command window widget child
work_window	The widget identifier of the widget that implements the work area
menu_bar	The widget identifier of the menu bar widget
horizontal_scroll_bar	The widget identifier of the horizontal scroll bar widget
vertical_scroll_bar	The widget identifier of the vertical scroll bar widget

4.3.1.2 Using the MAIN WINDOW SET AREAS Routine

As a convenience, you can use the MAIN WINDOW SET AREAS routine to specify all the child widgets to be used with a main window widget in one call. This routine takes the following arguments:

- The widget identifier of the main window widget
- The widget identifier of the menu bar widget
- The widget identifier of the work area widget
- The widget identifier of the command window widget
- The widget identifier of the scroll bar widget with horizontal orientation
- The widget identifier of the scroll bar widget with vertical orientation

You use the MAIN WINDOW SET AREAS routine after you have created the main window widget and each of its children. Pass a null value as an argument for any child widget not included in the main window.

4.3.1.3 Accepting Main Window Widget Defaults

If you do not explicitly specify which child widget should be used for each area of a main window widget using the SET VALUES intrinsic routine or the MAIN WINDOW SET AREAS routine, the main window widget selects the widget to be used from its list of managed children. The main window widget determines where to place its children based on the following rules:

- Any child widget that is a menu widget is used as the menu bar widget.

Creating a Main Window Widget

4.3 Creating a Main Window Widget

- Any child widget that is a command window widget is used as the command window widget.
- Any child widget that is a scroll bar widget is used as the scroll bar widget. A scroll bar widget with its **orientation** attribute set to horizontal is used as the horizontal scroll bar; a scroll bar widget with its **orientation** attribute set to vertical is used as the vertical scroll bar.
- A child widget of any other type is the work area widget.

The main window widget only considers currently managed children when determining which children will implement its areas. If you manage multiple children of the same type, the main window widget selects the first one to appear in its window. A main window widget can have any number of managed children; however, only five of these children can be visible at any one time.

When a main window widget is resized, it recalculates the layout of its children according to the same rules.

4.3.2 Customizing the Main Window Widget

The main window widget supports attributes that enable you to specify its size and position.

You can specify the size of a main window widget using the common widget attributes **width** and **height**. Specify these dimensions in pixels. If you create the main window widget with these attributes set to 0, the main window widget sizes itself to accommodate the size of all of its children. If you specify values for these attributes, the main window sizes the children to fit into the space allotted.

Specify the position of a main window widget using **x** and **y** attributes. Specify these values in pixels.

Example 4-1 is the section from the DECburger UIL module in which the main window is defined. Note that the width and height are explicitly set to 0. The size of the main window widget will be determined by the size requirements of its two children: the menu bar and the list box widgets.

Creating a Main Window Widget

4.3 Creating a Main Window Widget

Example 4-1 Main Window Created in the DECburger UIL Module

```
.
.
.
object
    S_MAIN_WINDOW : main_window {
        arguments {
            x = 10;
            y = 20;
            width = 0;
            height = 0;
        };
        controls {
            menu_bar      s_menu_bar;
            list_box      total_order;
        };
    };
.
.
.
```

4.3.3 Associating Callback Routines with a Main Window Widget

The main window widget executes a callback when it accepts the input focus. When a user clicks MB1 on the title bar, the main window widget will attempt to give the input focus to the work area widget or the command window widget (in that order). If neither of these children accepts the input focus and the **accept_focus** attribute is set to true, the main window widget will accept the input focus.

To associate a callback routine with this callback, pass a callback routine list to the main window widget as the value of the **focus_callback** attribute.

The main window widget does not support the help callback.

4.4 Creating a Scroll Window Widget

A scroll window widget can be used as the work area widget of a main window widget. In this case, the actual work area widget and the two scroll bar widgets are children of the scroll window widget, not the main window widget.

If the **shown_value_automatic_horiz** attribute is set to true, the scroll window widget automatically sizes and positions the slider in the horizontal scroll bar widget when your application moves the work area widget horizontally in relation to the scroll window widget. If the **shown_value_automatic_vert** attribute is set to true, the scroll window widget automatically sizes and positions the slider in the vertical scroll bar widget when your application moves the work area widget vertically in relation to the scroll window widget.

To create a scroll window widget, perform the following steps:

- 1 Create the scroll window widget.

Creating a Main Window Widget

4.4 Creating a Scroll Window Widget

Use any of the widget creation mechanisms listed in Table 4-4. The choice of mechanism depends on the attributes you need to access.

Table 4-4 Scroll Window Widget Creation Mechanisms

High-level routine	Use the SCROLL WINDOW routine to create a scroll window widget.
Low-level routine	Use the SCROLL WINDOW CREATE routine to create a scroll window widget.
UIL object type	Use the scroll_window object type to define a scroll window widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

2 Create the children of the scroll window widget.

The scroll window widget can have three children: a widget that implements the work area and two scroll bar widgets. Use any of the widget creation mechanisms to create these children.

3 Manage the children of the scroll window widget.

Use the MANAGE CHILD intrinsic routine to manage a single child. Use MANAGE CHILDREN to manage a group of children. In UIL this step is not necessary, since by default the DRM routine FETCH WIDGET manages the widgets it creates at run time.

4 Manage the scroll window widget.

Use the intrinsic routine MANAGE CHILD to manage the widget. In UIL this step is not necessary, since by default the DRM routine FETCH WIDGET manages the widgets it creates at run time.

After you complete these steps, if the parent of the scroll window widget has been realized, the scroll window widget will appear on the display.

Low-level routines and UIL provide access to the complete set of widget attributes at creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available using the high-level routine, use the SET VALUE intrinsic routine.) The *VMS DECwindows Toolkit Routines Reference Manual* describes the complete list of attributes supported by the scroll window widget. Table 4-5 lists the attributes available using the high-level routine SCROLL WINDOW. Pass values for these attributes as arguments to the routine.

Table 4-5 Widget Attributes Accessible Using the High-Level Routine SCROLL WINDOW

x	The x-coordinate of the upper left corner
y	The y-coordinate of the upper left corner
width	The width of the widget
height	The height of the widget

Creating a Main Window Widget

4.4 Creating a Scroll Window Widget

4.4.1 Adding Children to a Scroll Window Widget

As with the main window widget, there are three ways to add children to a scroll window widget:

- Use the SET VALUES intrinsic routine
- Use the SCROLL WINDOW SET AREAS routine
- Accept the defaults of the scroll window widget

4.4.1.1 Using SET VALUES to Add Children to a Scroll Window Widget

The scroll window widget supports attributes that identify the child widget to be used for each of its three designated areas. However, you cannot set these attributes at widget creation time because you do not know the widget identifier of the child until you create the children. You can only set these attributes after the scroll window widget has been created.

As with any widget attribute, you can use the SET VALUES intrinsic routine to assign values to these attributes after the widget has been created. Specify the widget identifier of the child widget as the value of these attributes. Table 4–6 lists these attributes of the main window widget.

Table 4–6 Child Widget Attributes of the Scroll Window Widget

Attribute	Value
work_window	The widget identifier of the widget that implements the work area
h_scroll	The widget identifier of the horizontal scroll bar widget
v_scroll	The widget identifier of the vertical scroll bar widget

4.4.1.2 Using the SCROLL WINDOW SET AREAS Support Routine

As a convenience, you can use the SCROLL WINDOW SET AREAS routine to specify all the child widgets to be used with a main window widget in one call. This routine takes the following arguments:

- The widget identifier of the scroll window widget
- The widget identifier of the scroll bar widget with horizontal orientation
- The widget identifier of the scroll bar widget with vertical orientation
- The widget identifier of the work area widget

You use the SCROLL WINDOW SET AREAS routine after you have created the scroll window widget and each of its children. Pass a null value as an argument for any child widget not included in the scroll window widget.

Creating a Main Window Widget

4.4 Creating a Scroll Window Widget

4.4.1.3 Accepting Scroll Window Widget Defaults

If you do not explicitly specify which child widget should be used for each area of a scroll window widget using the `SET VALUES` intrinsic routine or the `SCROLL WINDOW SET AREAS` routine, the scroll window widget selects the widget to be used from its list of managed children. The scroll window widget determines where to place its children based on the following rules:

- Any child widget that is a scroll bar widget is used as the scroll bar widget. A scroll bar widget with its **orientation** attribute set to horizontal is used as the horizontal scroll bar; a scroll bar widget with its **orientation** attribute set to vertical is used as the vertical scroll bar.
- A child widget of any other type is the work area widget.

The scroll window widget considers only currently managed children in its calculations. If you try to manage multiple children of the same type, the scroll window widget only manages the first.

4.5 Creating a Window Widget

The window widget provides a blank, rectangular work space and imposes no restrictions on what it contains. The window widget is the only XUI Toolkit widget that supports graphics operations.

To create a window widget, perform the following steps:

1 Create the window widget.

Use any of the widget creation mechanisms listed in Table 4–7. The choice of mechanism depends on the attributes you need to access.

Table 4–7 Window Widget Creation Mechanisms

High-level routine	Use the <code>WINDOW</code> routine to create a window widget.
Low-level routine	Use the <code>WINDOW CREATE</code> routine to create a window widget.
UIL object type	Use the window object type to define a window widget in a UIL module. At run time, the DRM routine <code>FETCH WIDGET</code> creates the widget according to this definition.

2 Manage the window widget.

Use the intrinsic routine `MANAGE CHILD` to manage the window widget. In UIL this step is not necessary, since widgets created using UIL are managed by default.

After you complete these steps, if the parent of the window widget has been realized, the window widget will appear on the display.

Low-level routines and UIL provide access to the complete set of widget attributes at creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available in a high-level routine, use the `SET`

Creating a Main Window Widget

4.5 Creating a Window Widget

VALUES intrinsic routine after the widget has been created.) The *VMS DECwindows Toolkit Routines Reference Manual* describes the complete list of attributes supported by the window widget. Table 4–8 lists the attributes available using the high-level routine WINDOW. Pass values for these attributes as arguments to the routine.

Table 4–8 Widget Attributes Accessible Using the High-Level Routine WINDOW

x	The x-coordinate of the upper left corner
y	The y-coordinate of the upper left corner
width	The width of the widget
height	The height of the widget
callback	The address of a callback routine list

4.5.1 Drawing Graphics in a Window Widget

To draw graphics in a window widget, create a callback routine that contains the graphics operations and associate the callback routine with the expose callback of the window widget. Whenever the window widget becomes visible on the screen, either when it is first created or when it becomes visible after being obscured, it executes this callback routine. You should always perform graphics operations from an expose callback routine because your application is responsible for repainting your window whenever an expose event occurs.

Example 4–2 draws a star using the DRAW LINES Xlib routine.

Example 4–2 Performing Graphics Operations in a Window Widget

```
#include <stdio>
#include <decw$include/DwtAppl.h>
Widget toplevel, graphics_window;
①Display *dpy;
②Window win;
③GC gc;

static void draw_in_window();
DwtCallback cb_list[2];

int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Arg arglist[15];
    int ac = 0;
    Screen *screen;
    XSetWindowAttributes xswa;
    XGCValues xgcv;

    toplevel = XtInitialize("Graphics Example", "exampleclass", NULL, 0, &argc, argv);
```

(continued on next page)

Using the Label, Separator, and Button Widgets

5.1 Overview of Label, Separator, and Button Widgets and Gadgets

A **toggle button widget**, like a push button widget, is a text string or pixmap inside a rectangular box with input and output capabilities. A toggle button widget maintains state information. A user can turn a toggle button widget on or off by clicking MB1. A toggle button widget usually contains an **indicator** to distinguish it from a push button widget. An indicator is a square or an oval, appearing at the left of the toggle button label, that provides a visual cue to the current state of the toggle button. For example, when the toggle button widget is on, the indicator is filled.

Use a push button widget to invoke an immediate action. Use a toggle button widget to implement functions that can be in on or off states.

Because the label, separator, and button widgets have such widespread usefulness, the XUI Toolkit provides the high-performance gadget version of these widgets. Gadgets provide the same functional capabilities as their widget counterparts but are not as customizable. By using gadgets instead of widgets wherever customization is not essential, you can improve the performance of your application. You can only use gadgets as children of menu widgets or dialog box widgets.

The label, separator, push button, and toggle button widgets, with the pull-down menu entry widget described in Section 6.2.1, are the only widgets in the XUI Toolkit with gadget counterparts.

Creating a Label Widget or Gadget

To create a label widget or gadget, perform the following steps:

- 1 Create the label widget or gadget.

Use one of the widget or gadget creation mechanisms listed in Table 5-1. Your choice of creation mechanism should depend on how much you need to customize the widget or gadget. Section 5.2.2 describes the attributes supported by the label gadget.

Table 5-1 Label Widget and Gadget Creation Mechanisms

Mechanism	Widget	Gadget
High-level routine	Use the LABEL routine to create a label widget.	There is no high-level gadget creation routine.
Low-level routine	Use the LABEL CREATE routine to create a label widget.	Use the LABEL GADGET CREATE routine to create a label gadget.
UIL object type	Use the label object type to define a label in a UIL module. At run time, the DRM routine FETCH WIDGET will create a label widget according to this definition.	Use the label object type with the gadget qualifier.

The label widget and the label gadget creation mechanisms return widget identifiers to the application; the XUI Toolkit does not define a gadget identifier.

5

Using the Label, Separator, and Button Widgets

This chapter provides the following:

- An overview of the label, separator, and button widgets and gadgets
- A detailed description of how to include the label, button, and separator widgets and gadgets in your application
- A description of how to use compound strings
- A description of how to define an additional mode of access, called an accelerator, to functions associated with buttons

5.1

Overview of Label, Separator, and Button Widgets and Gadgets

Labels, separators, and buttons provide much of the basic input and output capabilities in a VMS DECwindows application. Labels and separators allow you to output text and graphics to a user interface. (To handle text input, use the text widgets described in Chapter 9.) Push buttons and toggle button widgets allow you to provide users of your application with access to functions using a pointing device.

The XUI Toolkit includes a label widget, a separator widget, and two button widgets: a push button and a toggle button. These widgets are primitive widgets; that is, they cannot be parents of other widgets.

A **label widget** is a text string or pixmap inside a rectangular box. By default, the borders of the rectangle do not appear on a display, although you can make them visible. A label widget is an inactive interface object; it does not support callbacks.

A **separator widget** is a vertical or horizontal dotted line. A separator widget is an inactive interface object. Separator widgets can be thought of as label widgets containing a predefined pixmap, which is a dotted line.

A **push button widget** is a text string or pixmap inside a rectangular box with both input and output capabilities. When a user moves the pointer cursor onto a push button widget and presses MB1, the widget highlights to indicate a change in state. If the user then releases MB1 within the borders of the push button widget, the widget performs a callback to your application. Push button widgets can be thought of as label widgets with added input capabilities: the text string or pixmap provides the output capabilities, and the callback mechanism provides the input capabilities.

You can simulate push button activation using the `ACTIVATE_WIDGET` convenience routine. This routine causes the push button widget you specify to highlight and perform a callback to your application. This capability can be useful if you provide users with more than one way to access a function associated with a push button widget. When the user employs the alternate access, you can activate the push button widget to maintain a consistent interface.

Creating a Main Window Widget

4.5 Creating a Window Widget

Example 4-2 (Cont.) Performing Graphics Operations in a Window Widget

```
ac = 0;
XtSetArg( arglist[ac], XtNallowShellResize, TRUE ); ac++;
XtSetArg( arglist[ac], XtNx, 150 ); ac++;
XtSetArg( arglist[ac], XtNy, 150 ); ac++;
XtSetValues( toplevel, arglist, ac );

cb_list[0].proc = draw_in_window;
cb_list[0].tag = 0;
cb_list[1].proc = NULL;

ac = 0;
XtSetArg( arglist[ac], DwtNwidth, 600 ); ac++;
XtSetArg( arglist[ac], DwtNheight, 600 ); ac++;
XtSetArg( arglist[ac], DwtNexposeCallback, cb_list ); ac++;

④ graphics_window = DwtWindowCreate( toplevel, "gwindow", arglist, ac );
XtManageChild( graphics_window );
XtRealizeWidget( toplevel );

⑤ dpy = XtDisplay( graphics_window );
⑥ win = XtWindow( graphics_window );
⑦ screen = DefaultScreenOfDisplay(dpy);
/* Create graphics context. */
xgcv.foreground = BlackPixelOfScreen(screen);
xgcv.background = WhitePixelOfScreen(screen);
xgcv.line_width = 1;

⑧ gc = XCreateGC(dpy, win, GCForeground | GCBackground
| GCLineWidth, &xgcv);

XtMainLoop();
}

⑨ static void draw_in_window( w, tag, callback_data )
Widget w;
char *tag;
DwtWindowCallbackStruct *callback_data;
{
XPoint pt_arr[6];

pt_arr[0].x = 75;
pt_arr[0].y = 500;
pt_arr[1].x = 300;
pt_arr[1].y = 100;
pt_arr[2].x = 525;
pt_arr[2].y = 500;
pt_arr[3].x = 50;
pt_arr[3].y = 225;
pt_arr[4].x = 575;
pt_arr[4].y = 225;
pt_arr[5].x = 75;
pt_arr[5].y = 500;
```

(continued on next page)

Creating a Main Window Widget

4.5 Creating a Window Widget

Example 4-2 (Cont.) Performing Graphics Operations in a Window Widget

```
XDrawLines( dpy, win, gc, &pt_arr, 6, CoordModeOrigin );  
}
```

- ❶ This variable will hold a pointer to the display.
- ❷ This variable will hold a window identifier.
- ❸ This variable is a **graphics context**. For information about this data structure, see the *VMS DECwindows Xlib Programming Volume*.
- ❹ The WINDOW CREATE routine creates the window widget. In the argument list passed to the creation routine, the example specifies the size of the window widget and the callback routine to be associated with the expose callback of the window widget.
- ❺ The DISPLAY intrinsic routine returns a pointer to the display associated with the window widget.
- ❻ The WINDOW intrinsic routine returns the identifier of the window associated with the window widget.
- ❼ The DEFAULT SCREEN OF DISPLAY Xlib routine returns a pointer to the screen on which the window widget is displayed.
- ❽ The call to the CREATE GC Xlib routine defines the visible characteristics of the line used in the drawing in a graphics context structure. The line will be drawn in black.
- ❾ In the callback routine associated with the expose event, the DRAW LINES Xlib routine draws the star-shaped figure in the window widget. The display, window, and graphics context are specified as arguments to this routine. Whenever an expose event occurs in the window widget, this callback routine will be executed, causing the star to be drawn again.

4.5.2 Associating Callback Routines with a Window Widget

The window widget executes a callback when an expose event occurs within its borders. An expose event occurs when the window widget is mapped. Mapping occurs when the widget is realized for the first time, when your application goes from iconified state to active state, or when a portion of the widget that had previously been obscured by another widget becomes visible. When the window widget performs a callback, it returns the reason, the event structure that triggered the callback, and the identifier of the window in which the exposure event occurred. For more information about the data returned in the callback of the window widget, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a window widget, pass a callback routine list to the widget in the **expose_callback** attribute. See Example 4-2 for an illustration.

Creating a Main Window Widget

4.6 Creating a Command Window Widget

4.6 Creating a Command Window Widget

To create a command window widget, perform the following steps:

1 Create the command window widget.

Use any of the widget creation mechanisms listed in Table 4–9. The choice of mechanism depends on the attributes you need to access.

Table 4–9 Command Window Widget Creation Mechanisms

High-level routine	Use the COMMAND WINDOW routine to create a command window widget.
Low-level routine	Use the COMMAND WINDOW CREATE routine to create a command window widget.
UIL object type	Use the command_window object type to define a command window widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

2 Manage the command window widget.

Use the intrinsic routine MANAGE CHILD to manage the command window widget. In UIL this step is not necessary, since widgets created with UIL are managed by default.

After you complete these steps, if the parent of the command window widget has been realized, the command window widget will appear on the display.

Low-level routines and UIL provide access to the complete set of widget attributes at creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available using a high-level routine, use the SET VALUES intrinsic routine after the widget has been created.) The *VMS DECwindows Toolkit Routines Reference Manual* describes the complete list of attributes supported by the command window widget. Table 4–10 lists the attributes available using the high-level routine COMMAND WINDOW. Pass the values for these attributes as arguments to the routine.

Table 4–10 Widget Attributes Accessible Using the High-Level Routine COMMAND WINDOW

prompt	String used as command line prompt
lines	Number of command history lines displayed
callback	Address of a callback routine list
help_callback	Address of a callback routine list

Creating a Main Window Widget

4.6 Creating a Command Window Widget

4.6.1 Command Window Widget Support Routines

The XUI Toolkit provides a set of support routines that perform commonly needed operations on a command window widget (listed in Table 4–11). Use these routines to specify the text of the command line, append a string to the current contents of the command line, or display error messages.

Table 4–11 Command Window Widget Support Routines

COMMAND APPEND	Appends a text string onto the end of the text string currently in the command line.
COMMAND ERROR MESSAGE	Outputs an error message in the form of a text string. The message appears in the command history window of the command window widget.
COMMAND SET	Replaces the contents of the command line with the text string specified.

4.6.2 Specifying the Contents of the Command Line

After the command window widget appears on the display, the user of your application can enter a command string in its text entry area. Your application can specify the initial contents of the text area of the command window widget by assigning the address of text string as the value of the **value** attribute. Note that this text string does not have to be converted into a compound string.

If the text string ends with a carriage return or a line-feed character, the command window widget executes the command, notifies your application using the callback mechanism, moves the command line into the history window, and issues a new prompt. The text string can also represent multiple command lines.

To change the value of the command window widget after the widget has been created, you can assign a new string as the value of the **value** argument using the SET VALUES intrinsic routine or you can use the command window support routine COMMAND SET. The COMMAND SET routine takes the following arguments:

- The widget identifier of the command window widget
- The text to be placed in the command line

To add text to a command line, use the COMMAND APPEND support routine. This routine takes the following two arguments:

- The widget identifier of the command window widget
- The text to be added to the command line

4.6.3 **Displaying Error Messages in the Command Window Widget**

To display error messages generated by command line execution, use the `COMMAND ERROR MESSAGE` support routine. This routine accepts the following arguments:

- The widget identifier of the command window widget
- The text of the error message to be displayed

The error message appears in the command history area of the command window widget.

4.6.4 **Defining Accelerators for the Command Window Widget**

You can define the actions performed by the command window widget upon certain keyboard events using the `t_translations` attribute. Pass a parsed translation table as the value of this attribute. You typically use this attribute to define accelerators. See Section 6.7 for details about translation tables.

4.6.5 **Customizing the Appearance of the Command Window Widget**

The attributes of the command window widget enable you to customize the following aspects of its appearance:

- The command line prompt
- The number of command lines visible in the command history window

You can assign values to widget attributes when you create the widget using any of the widget creation mechanisms, or after the widget has been created using the intrinsic routine `SET VALUES`.

4.6.5.1 **Specifying the Command Line Prompt**

You can specify the string of characters used as the command line prompt using the `prompt` attribute. Specify the prompt as a text string. The default prompt is the right angle bracket (`>`).

Note that you must convert the prompt text string into a compound string before passing it to the command window widget.

4.6.5.2 **Specifying the Size and Content of the Command History Window**

You can use the `lines` attribute to specify how many command lines appear in the command history window of the command window widget. Specify the number of lines as an integer. By default, the command window widget displays two command lines in its command history window.

Creating a Main Window Widget

4.6 Creating a Command Window Widget

4.6.6 **Associating Callback Routines with the Command Window Widget**

Using the callback mechanism, a command window widget notifies your application when a change is made to the contents of the command line or when a command is executed. A command is executed when the user presses the Return key or your application passes a string containing a return or line-feed character. When a command is executed, the command window widget removes it from the command entry field and places it in the command history. Your application must parse the command string and execute the command in a callback routine.

The command window widget also performs a callback when it accepts the input focus.

When a command window widget performs a callback, it returns callback data to the application. In this callback data, the command window widget returns the text string that is the contents of the command line and the length of the command line. For complete information about the data returned in a callback by the command window widget, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a command window widget callback, pass a callback routine list to one of the command window widget callback attributes. Table 4–12 describes what conditions trigger these callbacks and the widget attributes you use to associate callback routines with them.

Table 4–12 Command Window Widget Callbacks

Callback Attribute	Description
value_callback	The contents of the command line changed.
command_entered_callback	The user has pressed the Return or the Line Feed key.
focus_callback	The command window widget has received the input focus.

Using the Label, Separator, and Button Widgets

5.2 Creating a Label Widget or Gadget

2 Manage the label widget or gadget.

Use the intrinsic routine `MANAGE CHILD` to manage the widget or gadget.

After you complete these steps, if the parent of the label widget or gadget has been realized, the label widget or gadget will appear on the display.

Low-level routines and `UIL` provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these attributes at widget creation time. (To access attributes not available in a high-level routine, use the `SET VALUES` intrinsic routine after the widget has been created.) Table 5-2 lists the attributes you can set if you use the high-level routine `LABEL` to create a label widget. Pass the values of these attributes as arguments to the routine.

Table 5-2 Attributes Accessible Using the High-Level Routine `LABEL`

<code>x</code>	The x-coordinate of the upper left corner
<code>y</code>	The y-coordinate of the upper left corner
<code>label¹</code>	The text or pixmap to be displayed in the label widget
<code>help_callback</code>	The address of a callback routine list

¹The high-level routines use this spelling for the label attribute to avoid conflicts with programming languages in which "label" is a reserved word.

5.2.1 Customizing a Label Widget

The attributes of the label widget enable you to customize the following aspects of its appearance and functioning:

- Size and position
- Alignment
- Margins
- Content

5.2.1.1 Specifying the Size and Position of a Label Widget

Use the common widget attributes `width` and `height` to specify the size of a label widget. By default, a label widget sizes itself to fit the text string or pixmap it contains. The parent widget of the label widget can also determine the size of a label widget. For example, menu widgets determine the dimensions of the widgets that implement the menu items they contain.

You can specify that a label widget always attempt to fit the text or pixmap it contains using the `conform_to_text` attribute. If you set this attribute to true, the label widget will grow or shrink as the text or pixmap it contains grows or shrinks.

Use the common widget attributes `x` and `y` to specify the position of a label widget. You do not always need to specify the position of the label widget, because the parent of the label widget will determine its position.

Using the Label, Separator, and Button Widgets

5.2 Creating a Label Widget or Gadget

5.2.1.2 Specifying the Alignment in a Label Widget

Use the **alignment** attribute to position the text string within the borders of the label widget. You cannot align pixmaps contained in a label widget. You can center the text string within the label widget, or you can align the text string to the right side or to the left side of the label widget. The *VMS DECwindows Toolkit Routines Reference Manual* lists the constants used to indicate types of alignment.

5.2.1.3 Specifying Margins in a Label Widget

The label widget supports six margin attributes that you can use to determine the amount of space surrounding the text or pixmap the widget contains.

Specify the amount of space between the left border of the label widget and the beginning of the text string or pixmap it contains in the **margin_width** attribute. This value is also used as the right margin.

Specify the amount of space between the top border of the label widget and the top of the text or pixmap in the label widget in the **margin_height** attribute. This value is also used to determine the amount of space left between the bottom side of the label widget and the bottom of the text or pixmap it contains.

The other four margin attributes, **margin_left**, **margin_right**, **margin_top**, and **margin_bottom**, determine the space surrounding the text or pixmap contained in the label widget. A text string or pixmap is contained within its own rectangle. Note that the borders of this inner rectangle are distinct from the borders of the label widget. You cannot make this inner rectangle visible. Using these attributes, you can specify margins within this rectangle. For example, the distance between the left side of the label widget and the first character in a text string can be the sum of the **margin_width** and **margin_left** attributes.

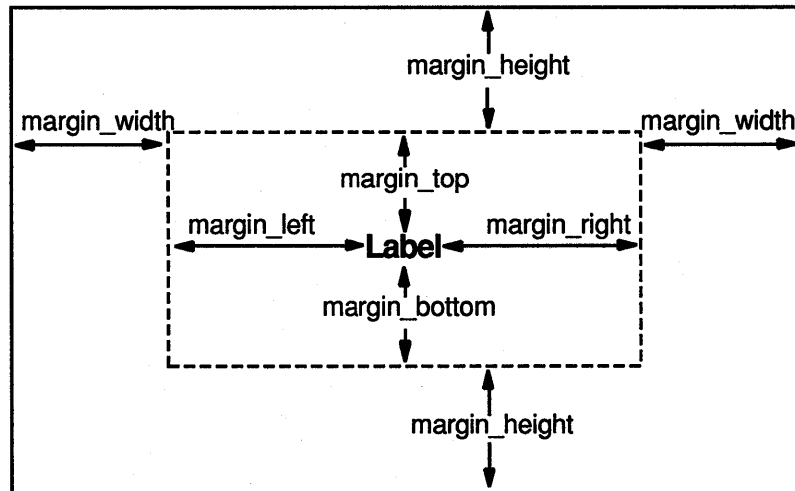
Figure 5-1 illustrates these margins in a label widget.

Using the Label, Separator, and Button Widgets

5.2 Creating a Label Widget or Gadget

Figure 5-1 Attributes for Setting Margins

Label Widget



ZK-0199A-GE

5.2.1.4 Specifying the Content of a Label Widget

Use the **label** attribute to specify the text the label widget will contain. You must pass the text to the label widget in the form of a compound string. Section 5.6 describes how to convert text strings to compound strings. Identify the type of label as a text string in the **label_type** attribute.

Use the **pixmap** attribute to specify the pixmap used in a label widget. Pass the identifier of the pixmap to the label in this attribute. You can create a pixmap in the following three ways:

- Use the bitmap editor supplied with Xlib.
- Use the UIL built-in ICON function, described in Section 3.2.7.8.
- Use the DECpaint application, described in the *VMS DECwindows Desktop Applications Guide*.

When using a pixmap in a label widget, you must specify the type of label in the **label_type** attribute.

5.2.2 Customizing a Label Gadget

The label gadget provides access to only a subset of the attributes provided by the label widget. The following list summarizes aspects of the label gadget that you can customize:

- Size and position
- Alignment

Using the Label, Separator, and Button Widgets

5.2 Creating a Label Widget or Gadget

- Text content of the label

For information about assigning values to these attributes, see Section 5.2.1.

The primary label widget attributes not supported by label gadgets are the margin and pixmap attributes. However, the gadget version also does not support certain common widget attributes supported by the label widget. The attributes the gadget does not support deal mainly with aspects of the appearance of the widget that relate to properties of the widget window. To reduce their overhead and improve performance, gadgets do not have an associated window. For the attributes of the label gadget you cannot customize, the label gadget uses the value contained in its parent.

Specifically, the label gadget imposes the following restrictions:

- You cannot specify margins.
- You cannot specify a pixmap label.
- You cannot specify the color of the foreground, background, or border.
- You cannot specify the pixmap used as the foreground, background, or border of a widget.
- You cannot specify a font.

5.3 Creating a Separator Widget or Gadget

To create a separator widget or gadget, perform the following steps:

- 1 Create the separator widget or gadget.

Use any of the widget or gadget creation mechanisms listed in Table 5-3. Your choice of which creation mechanism to use depends on how you want to configure the separator widget and which attributes you need to set.

Table 5-3 Separator Widget and Gadget Creation Mechanisms

Mechanism	Widget	Gadget
High-level routine	Use the SEPARATOR routine to create a separator widget.	There is no high-level gadget creation routine.
Low-level routine	Use the SEPARATOR CREATE routine to create a separator widget.	Use the SEPARATOR GADGET CREATE routine to create a separator gadget.
UIL object type	Use the separator object type to define a separator in a UIL module. At run time, the DRM routine FETCH WIDGET creates the object according to this definition.	Use the separator object type with the gadget qualifier.

- 2 Manage the separator widget or gadget.

Use the intrinsic routine MANAGE CHILD to manage a separator widget or gadget.

Using the Label, Separator, and Button Widgets

5.3 Creating a Separator Widget or Gadget

After you complete these steps, if the parent of the separator has been realized, the separator widget or gadget will appear on the display.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these attributes at widget creation time. (To access attributes not available in a high-level routine, use the SET VALUES intrinsic routine after the widget has been created.) Table 5-4 lists the attributes you can set if you use the high-level routine SEPARATOR to create a separator widget. Pass the values of these attributes as arguments to the routine.

Table 5-4 Attributes Accessible Using the High-Level Routine SEPARATOR

x	Specifies the x-coordinate of the upper left corner
y	Specifies the y-coordinate of the upper left corner
orientation	Specifies whether the separator widget is vertical or horizontal

5.3.1 Customizing a Separator Widget or Gadget

The separator widget and gadget support all of the attributes supported by the label widget and gadget. For information about customizing a label widget, see Section 5.2.1.

In addition, the separator widget and gadget support an attribute with which you can specify their orientation. Separator widgets and gadgets can have either a horizontal or a vertical orientation. Specify the orientation of the separator widget or gadget in the **orientation** attribute using the constants listed in the *VMS DECwindows Toolkit Routines Reference Manual*.

5.4 Creating a Push Button Widget or Gadget

To create a push button widget or gadget, perform the following steps:

- 1 Create the push button widget or gadget.

Use any of the three widget or gadget creation mechanisms listed in Table 5-5. The choice of creation mechanism depends on how you want to configure the push button widget or gadget and which attributes you need to set.

Using the Label, Separator, and Button Widgets

5.4 Creating a Push Button Widget or Gadget

Table 5–5 Push Button Widget and Gadget Creation Mechanisms

Mechanism	Widget	Gadget
High-level routine	Use the PUSH BUTTON routine to create a push button widget.	There is no high-level gadget creation routine.
Low-level routine	Use the PUSH BUTTON CREATE routine to create a push button widget.	Use the PUSH BUTTON GADGET CREATE routine to create a push button gadget.
UIL object type	Use the push_button object type to define a push button widget in a UIL module. At run time, the DRM routine FETCH WIDGET will create the object according to this definition.	Use the push_button object type with the gadget qualifier.

2 Manage the push button widget or gadget.

Use the intrinsic routine `MANAGE CHILD` to manage a push button widget or gadget.

After you complete these steps, if the parent of the push button widget or gadget has been realized, the push button widget or gadget will appear on the display.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these attributes at widget creation time. (To access attributes not available using the high-level routine, use the `SET VALUES` intrinsic routine after the widget has been created.) Table 5–6 lists the attributes you can set if you use the high-level routine `PUSH BUTTON` to create a push button widget. Pass the values of these attributes as arguments to the routine.

Table 5–6 Attributes Accessible Using the High-Level Routine PUSH BUTTON

x	The x-coordinate of the upper left corner
y	The y-coordinate of the upper left corner
labl ¹	The text to be displayed in the push button widget
callback	The address of a callback routine list
help_callback	The address of a callback routine list

¹The high-level routines use this spelling for the label attribute to avoid conflicts with programming languages in which "label" is a reserved word.

Example 5–1 is the section from the `DECburger UIL` module in which the `DECburger` option menu widget is defined. The example creates the individual items in the option menu widget as push button gadgets. This is a typical use of push button widgets or gadgets.

Using the Label, Separator, and Button Widgets

5.4 Creating a Push Button Widget or Gadget

Example 5-1 Push Button Gadgets in the DECburger Option Menu

```
.
.
.
①object
  fries_option_menu : option_menu {
    arguments {
      x = 130;
      y = 22;
      label_label = k_size_label_text;
      menu_history = push_button medium_fries;
    };
    controls {
      pulldown_menu fries_menu;
    };
  };
②object
  fries_menu : pulldown_menu {
    controls {
      push_button    tiny_fries;
      push_button    small_fries;
      push_button    medium_fries;
      push_button    large_fries;
      push_button    huge_fries;
    };
  };
③object
  tiny_fries : push_button {
    arguments {
      label_label = k_tiny_label_text;
    };
    callbacks {
      activate = procedure activate_proc (k_fries_tiny);
    };
  };
.
.
.
```

- ① The object declaration of the option menu widget lists its only child, the pull-down menu widget. (See Section 6.6 for more information about the option menu widget.)
- ② In this object declaration of the pull-down menu widget, the five push button gadgets that implement the menu items are listed as children of the pull-down menu widget.
- ③ This is the object declaration of the first push button gadget used in the pull-down menu widget. Note that, because DECburger defines gadgets as the default type for all the push buttons it uses, it does not have to explicitly qualify the `push_button` object type with the gadget qualifier. See Section 3.2.4 for more information about specifying default object types.

In the object declarations for each push button gadget, DECburger only specifies the text the gadget will contain. DECburger allows the pull-down menu widget to determine the size and position of the push button gadgets that are its children.

Using the Label, Separator, and Button Widgets

5.4 Creating a Push Button Widget or Gadget

5.4.1 Customizing a Push Button Widget

The push button widget supports all attributes supported by a label widget. For information about customizing a label widget, see Section 5.2.1.

In addition, the push button widget supports its own unique attributes that enable you to customize the following aspects of its appearance and functioning:

- Highlighting behavior
- Shadowing
- Pixmap used to indicate insensitive state

5.4.1.1 Specifying Highlighting Behavior

Use the **border_highlight** or the **fill_highlight** attribute to specify how a push button widget is highlighted when selected by a user. If you set the **border_highlight** attribute to true, the push button widget indicates it has been selected by highlighting its border. (This is the default behavior for push button widgets and gadgets in menus.) If you set the **fill_highlight** attribute to true, the entire push button widget changes color to indicate it has been selected by a user.

You also can use the common widget attributes **highlight_pixel** and **highlight_pixmap** to specify the color or pixmap pattern used as the highlight.

If you want your application to conform to the recommendations of the *XUI Style Guide*, accept the default values determined by the use of the push button widget.

5.4.1.2 Specifying Shadowing

The **shadow** attribute enables the application to choose whether the push button widget should appear with a shadow. The shadow provides push button widgets with a three-dimensional look.

5.4.1.3 Specifying the Insensitive Pixmap

Use the **insensitive_pixmap** attribute to specify the pixmap the push button widget should contain when it is insensitive to user input.

5.4.2 Customizing a Push Button Gadget

Push button gadgets do not provide access to any attributes beyond those supported by the label gadget. For information about customizing a label widget, see Section 5.2.1.

Using the Label, Separator, and Button Widgets

5.4 Creating a Push Button Widget or Gadget

5.4.3 Associating Callback Routines with a Push Button Widget or Gadget

When activated, a push button widget or gadget notifies an application using the callback mechanism. The push button widget or gadget is activated when a user moves the pointer cursor onto it and clicks MB1.

In addition, the push button widget performs callbacks when a user moves the pointer cursor onto the push button and holds down MB1. This user interaction is said to **arm** the push button widget. The push button widget also performs a callback when a user moves the pointer cursor off the push button without releasing MB1. This user interaction is said to **disarm** the push button widget. The push button gadget does not support the **arm** or **disarm** callback reasons.

The push button widget and gadget both perform callbacks when the user presses the Help key while simultaneously clicking MB1 inside a push button widget or gadget.

When the push button widget or gadget performs a callback, it returns callback data to your application. For complete information about the data returned by the push button widget or gadget in a callback, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a push button widget or gadget, pass a callback routine list to one of the callback attributes supported by the widget or gadget. Table 5-7 lists the callback attributes supported by the push button widget and gadget and the conditions that trigger these callbacks.

Table 5-7 Push Button Widget and Gadget Callbacks

Callback Attribute	Description
activate_callback	A user has clicked MB1 on the push button widget or gadget.
arm_callback	A user has moved the pointer cursor onto the push button widget and is holding down MB1 (widget only).
disarm_callback	A user has moved the pointer cursor off the push button widget without releasing MB1 (widget only).
help_callback	A user has pressed the Help key while the pointer cursor is in the push button widget or gadget.

All the push button widgets and gadgets in the DECburger sample application execute the same callback routine, called *activate_proc*, when activated. DECburger uses the tag to determine which push button widget or gadget performed the callback and then performs whatever processing is required. Example 5-2 is a fragment from the callback routine in which the callbacks from the option menu widget are handled. When a push button gadget in the option menu widget is activated, DECburger reads the text label in the activated push button gadget to retrieve the value of the user's selection.

Using the Label, Separator, and Button Widgets

5.4 Creating a Push Button Widget or Gadget

Example 5-2 Push Button Callback Procedure in the DECburger Application

```
.
.
static void activate_proc(w, tag, reason)
    Widget w;
    int *tag;
    unsigned long *reason;
{
    int widget_num = *tag;
    int i, value, fries_num;
    char *txt, *fries_text, *list_txt, list_buffer[20];
    switch (widget_num)
    {
        .
        .
        .
        case k_fries_tiny:
        case k_fries_small:
        case k_fries_medium:
        case k_fries_large:
        case k_fries_huge:
            get_something(w, DwtNlabel, &current_fries);
            break;
        .
        .
        .
    }
}
```

5.5 Creating a Toggle Button Widget or Gadget

To create a toggle button widget or gadget, perform the following steps:

- 1 Create the toggle button widget or gadget.

Use any of the widget creation mechanisms listed in Table 5-8. The choice of creation mechanism depends on how you want to customize the toggle button widget and which attributes you need to set.

Table 5-8 Toggle Button Widget and Gadget Creation Mechanisms

Mechanism	Widget	Gadget
High-level routine	Use the TOGGLE BUTTON routine to create a toggle button widget.	There is no high-level gadget creation routine.
Low-level routine	Use the TOGGLE BUTTON CREATE routine to create a toggle button widget.	Use the TOGGLE BUTTON GADGET CREATE routine to create a toggle button gadget.
UIL object type	Use the toggle_button object type to define a toggle button widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.	Use the toggle_button object type with the gadget qualifier.

Using the Label, Separator, and Button Widgets

5.5 Creating a Toggle Button Widget or Gadget

2 Manage the toggle button widget or gadget.

Use the intrinsic routine `MANAGE CHILD` to manage a toggle button widget or gadget.

After you complete these steps, if the parent of the toggle button widget or gadget has been realized, the toggle button widget or gadget will appear on the display.

Low-level routines and `UIL` provide access to the complete set of widget attributes at widget creation time. High-level routines provide access to only a subset of these attributes at widget creation time. (To access attributes not available in a high-level routine, use the `SET VALUES` intrinsic routine after the widget has been created.) Table 5-9 lists the attributes you can set if you use the high-level routine `TOGGLE BUTTON` to create a toggle button widget. Pass the values of these attributes as arguments to the routine.

Table 5-9 Attributes Accessible Using the High-Level Routine `TOGGLE BUTTON`

<code>x</code>	The x-coordinate of the upper left corner
<code>y</code>	The y-coordinate of the upper left corner
<code>labl¹</code>	The text to be displayed in the toggle button widget
<code>value</code>	The state of the toggle button widget
<code>callback</code>	The address of a callback routine list
<code>help_callback</code>	The address of a callback routine list

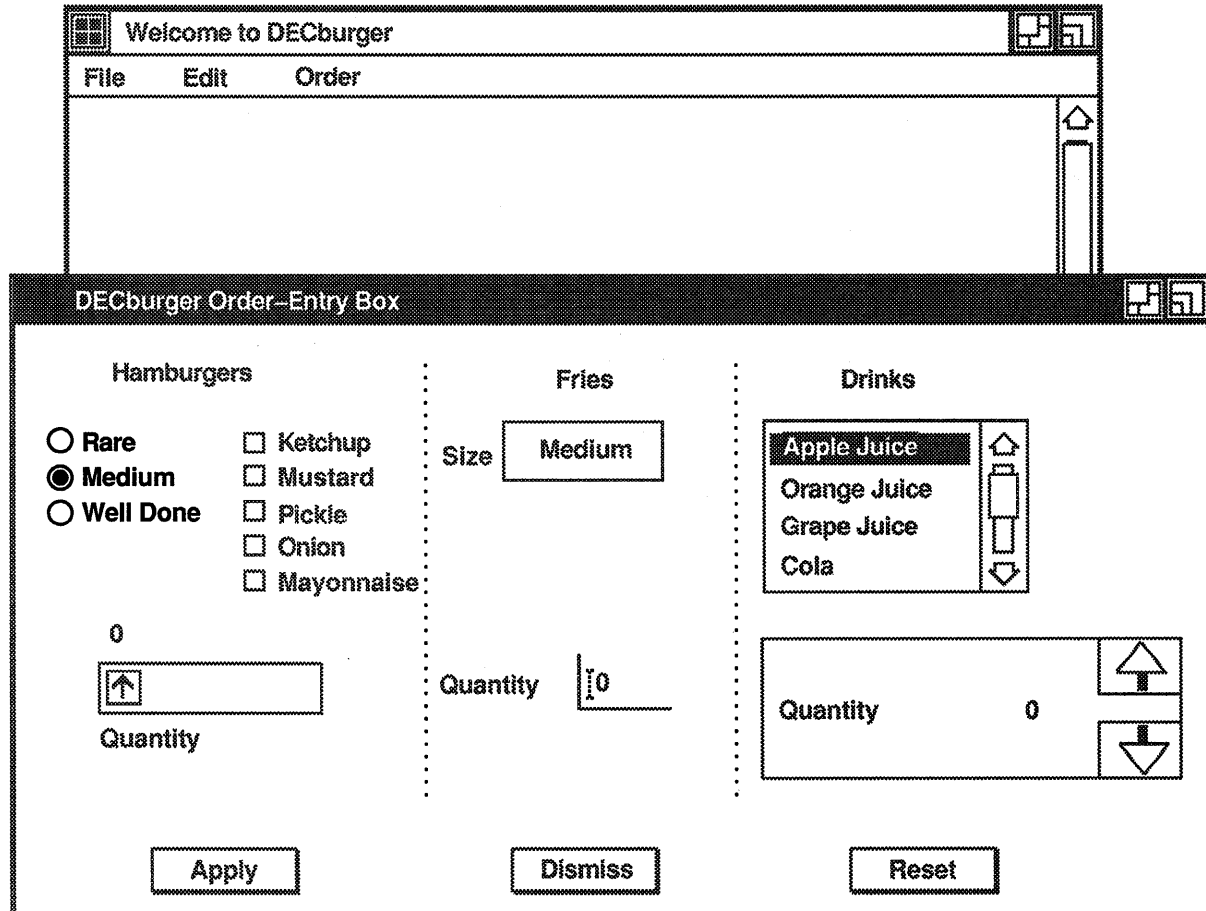
¹The high-level routines use this spelling for the label attribute to avoid conflicts with programming languages in which "label" is a reserved word.

The `DECburger` sample application uses toggle buttons in a radio box widget. Figure 5-2 illustrates this widget as it appears in the `DECburger` user interface.

Using the Label, Separator, and Button Widgets

5.5 Creating a Toggle Button Widget or Gadget

Figure 5-2 Radio Box with Toggle Button Gadgets in the DECburger Application



ZK-0160A-GE

In Example 5-3, the DECburger application creates the radio box widget and the three toggle button gadgets that implement the items it contains. Note that the only attribute explicitly set in the toggle button gadget definitions is the text they will contain. DECburger allows the radio box widget to determine the size of the toggle button gadgets.

Using the Label, Separator, and Button Widgets

5.5 Creating a Toggle Button Widget or Gadget

Example 5-3 Creating the Radio Box Widget in the DECburger Application

```
.
.
.
object
  burger_doneness_box : radio_box {
    arguments {
      x = 10;
      y = 22;
      orientation = DwtOrientationVertical;
      border_width = 0;
    };
    controls {
      ❶ toggle_button burger_rare;
        toggle_button burger_medium;
        toggle_button burger_well;
    };
  };

object
  ❷ burger_rare : toggle_button {
    arguments {
      label_label = k_rare_label_text;
    };
    callbacks {
      value_changed = procedure toggle_proc (k_burger_rare);
      create = procedure create_proc (k_burger_rare);
    };
  };

object
  burger_medium : toggle_button {
    arguments {
      label_label = k_medium_label_text;
      toggle_value = on;
    };
    callbacks {
      value_changed = procedure toggle_proc (k_burger_medium);
      create = procedure create_proc (k_burger_medium);
    };
  };

object
  burger_well : toggle_button {
    arguments {
      label_label = k_well_done_label_text;
    };
    callbacks {
      value_changed = procedure toggle_proc (k_burger_well);
      create = procedure create_proc (k_burger_well);
    };
  };
.
.
.
```

- ❶ The controls section of the radio box widget object declaration lists the three toggle button gadgets that are its children.

Using the Label, Separator, and Button Widgets

5.5 Creating a Toggle Button Widget or Gadget

- ② After defining the radio box widget, the DECburger UIL module defines each of the three toggle button gadgets it contains. Note that DECburger does not have to use the gadget qualifier with the `toggle_button` object type because, at the beginning of the UIL module, DECburger declares toggle button gadgets as the default of the `toggle_button` object type. See Section 3.2.4 for more information about specifying default object types.

For each toggle button gadget, DECburger passes the text string the gadget will contain as the value of the `label` attribute (called `label_label` in UIL). DECburger accepts defaults for all other toggle button gadget attributes. The radio box widget determines the sizing and positioning of the toggle button gadgets that are its children.

5.5.1 Specifying the State of a Toggle Button Widget or Gadget

The toggle button widget and gadget both maintain their current state in their `value` attribute. You can set the current state of a toggle button widget or gadget when you create it by setting this attribute on or off. The *VMS DECwindows Toolkit Routines Reference Manual* lists the constants used to indicate these values.

DECburger sets the initial value of one of the toggle button gadgets used in the radio box widget. In this way, DECburger specifies the default choice for the radio box widget. Example 5-4 shows the UIL object declaration of the toggle button in which the `value` attribute is set to `on`. (Note that, in UIL, this attribute is named `toggle_value`.)

Example 5-4 Setting the Initial State of a Toggle Button

```
object
  burger_medium : toggle_button {
    arguments {
      label_label = k_medium_label_text;
      toggle_value = on;
    };
    callbacks {
      value_changed = procedure toggle_proc (k_burger_medium);
      create = procedure create_proc (k_burger_medium);
    };
  };
};
```

After the toggle button widget or gadget has been created, you can read the current state or set the current state using the `GET VALUES` and `SET VALUES` intrinsic routines. Alternately, you can use the following support routines provided by the XUI Toolkit for use with toggle button widgets and gadgets:

- `TOGGLE BUTTON GET STATE` routine
- `TOGGLE BUTTON SET STATE` routine

The `TOGGLE BUTTON GET STATE` support routine retrieves the current value of the toggle button widget or gadget. The routine takes as its only argument the identifier of the toggle button widget or gadget whose state you want to read.

Using the Label, Separator, and Button Widgets

5.5 Creating a Toggle Button Widget or Gadget

The TOGGLE BUTTON SET STATE support routine allows you to set the current value of the toggle button widget or gadget. This routine takes the following arguments:

- The widget identifier of the toggle button widget or gadget
- The value you want the toggle button widget or gadget to have
- A Boolean variable that determines whether the toggle button widget or gadget notifies your application that its value has changed

The DECburger sample application uses the TOGGLE BUTTON SET STATE support routine to set the state of one of the toggle button gadgets in the radio box widget when a user chooses to reset the user interface.

5.5.2 Customizing a Toggle Button Widget

The toggle button widget supports all the attributes supported by the label widget. (For information about customizing a label widget, see Section 5.2.1.) In addition to supporting these attributes, the toggle button widget allows you to customize the following:

- The appearance of the indicator
- The pixmaps used to indicate on and off states when the widget is sensitive
- The pixmaps used to indicate on and off states when the widget is insensitive

5.5.2.1 Specifying the Appearance of the Indicator

Use the **shape**, **spacing**, and **indicator** attributes to specify the appearance of the indicator and its presence in the toggle button widget.

To specify whether the indicator is square or oval, use the **shape** attribute. The *VMS DECwindows Toolkit Routines Reference Manual* lists the constants used to specify these values.

To specify the amount of space between the indicator and the start of the label (if it is a text label) in the toggle button widget, use the **spacing** attribute. Specify this value in pixels.

To specify whether the toggle button widget includes an indicator, use the **indicator** attribute. Set this attribute to true to include an indicator in a toggle button widget. The *XUI Style Guide* recommends that toggle button widgets include an indicator.

If you set the **visible_when_off** attribute to true, the indicator will not be visible in the toggle button widget when it is in its off state.

5.5.2.2 Specifying On and Off Pixmaps

To specify the pixmap label that appears in a toggle button widget, pass the identifier of the pixmap in the **pixmap_on** and **pixmap_off** attributes. You can specify two separate pixmaps that graphically represent the toggle button in its on and off states.

Using the Label, Separator, and Button Widgets

5.5 Creating a Toggle Button Widget or Gadget

To specify the pixmap label that will appear in a toggle button widget when it is insensitive to user input, pass the identifier of the pixmap in the `insensitive_pixmap_on` and `insensitive_pixmap_off` attributes.

5.5.3 Customizing a Toggle Button Gadget

Toggle button gadgets support all attributes supported by the label gadget. For information about customizing a label gadget, see Section 5.2.2.

In addition, with the toggle button gadget, you can customize the shape of the indicator (see Section 5.5.2.1).

5.5.4 Associating Callback Routines with a Toggle Button Widget or Gadget

When its value changes, a toggle button widget or gadget notifies an application using the callback mechanism. The value changes when a user selects the toggle button widget or gadget by moving the pointer cursor onto it and clicking MB1. Your application can also change the value of the toggle button widget or gadget using the `TOGGLE BUTTON SET STATE` support routine or the intrinsic routine `SET VALUES`.

In addition, the toggle button widget performs callbacks when a user moves the pointer cursor onto it and holds down MB1. This user interaction arms the toggle button widget. The toggle button widget also performs a callback when a user moves the pointer cursor off of it without releasing MB1. This user interaction disarms the toggle button widget. The toggle button gadget does not support these callbacks.

The toggle button widget and gadget both perform callbacks when the user presses the Help key while simultaneously clicking MB1 inside the toggle button widget or gadget.

When the toggle button widget or gadget performs a callback, it returns callback data to your application. In this callback data, the toggle button widget or gadget returns its current value, along with other data. For complete information about the data returned in a callback by the toggle button widget or gadget, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a toggle button widget or gadget, pass a callback routine list to one of the callback attributes they support. Table 5-10 lists the callback attributes and the conditions that trigger these callbacks.

Using the Label, Separator, and Button Widgets

5.5 Creating a Toggle Button Widget or Gadget

Table 5–10 Toggle Button Widget and Gadget Callbacks

Callback Attribute	Description
value_changed	A user has clicked MB1 on the toggle button widget or gadget, causing it to change value, or your application has assigned a value to the value attribute using the SET VALUES intrinsic routine or the support routine TOGGLE BUTTON SET STATE.
arm_callback	A user has moved the pointer cursor onto the toggle button widget and is holding down MB1 (widget only).
disarm_callback	A user has moved the pointer cursor off the toggle button widget without releasing MB1 (widget only).
help_callback	A user has pressed the Help key while the pointer cursor is in the toggle button widget or gadget.

All the toggle button gadgets in the DECburger sample application use the same callback routine, called *toggle_proc*, shown in Example 5–5. In the callback routine, DECburger assigns the value returned in the callback data to a position in an array, called *toggle_array*. DECburger uses the array to store the current state of all its toggle buttons. In the callback routine, DECburger determines which toggle button gadget performed the callback by checking the tag field of the callback data.

Example 5–5 Toggle Button Callback Procedure in the DECburger Application

```
.  
. .  
static void toggle_proc(w, tag, toggle)  
    Widget w;  
    int *tag;  
    DwtTogglebuttonCallbackStruct *toggle;  
{  
    toggle_array[*tag - k_burger_min] = toggle->value;  
}  
. .  
.
```

5.6 Working with Compound Strings

All the text labels used in XUI Toolkit widgets are **compound strings**. For example, to specify the text in the **label** attribute of the label, push button, or toggle button widget (or gadget), you must pass the address of a compound string. (The simple text widget is the only XUI Toolkit widget that does not accept compound strings. See Chapter 9 for more information.)

Using the Label, Separator, and Button Widgets

5.6 Working with Compound Strings

A compound string is a Digital Document Interchange Format (DDIF) data type that describes a text string not only by the characters it contains but also by other aspects, such as the character set and writing direction used to display the text on a workstation screen. A compound string can be made up of multiple segments. You can specify a different character set, writing direction, or other attribute for each different segment of a compound string. (For an illustration of a compound string containing multiple segments, see Figure 9-1 in Section 9.1.)

The XUI Toolkit includes a set of routines that enable you to perform the following tasks on compound strings:

- Create a compound string
- Create a compound string made up of multiple segments
- Manipulate a compound string
- Retrieve information about the compound string
- Specify fonts

Table 5-11 lists all the compound string routines in the XUI Toolkit.

Table 5-11 Compound String Routines

Routine	Description
Creating a Compound String	
CS STRING	Creates a compound string, allowing you to specify all aspects of the string including character set and writing direction.
LATIN1 STRING	Creates a compound string that uses the ISO Latin1 character set and the left-to-right writing direction.
STRING	Creates a compound string, allowing you to specify character set and writing direction.
Manipulating a Compound String	
CS BYTE CMP	Compares two compound strings to determine if they are identical.
CS CAT	Appends a copy of one compound string to the end of another compound string.
CS COPY	Copies a compound string.
CS EMPTY	Determines if the compound string contains any text segments.
CS LEN	Returns the number of bytes in a compound string.

(continued on next page)

Using the Label, Separator, and Button Widgets

5.6 Working with Compound Strings

Table 5–11 (Cont.) Compound String Routines

Routine	Description
Retrieving Information About a Compound String	
GET NEXT SEGMENT	Returns information about a segment of a compound string.
INIT GET SEGMENT	Initializes a compound string context.
STRING FREE CONTEXT	Frees a compound string context.
STRING INIT CONTEXT	Initializes a compound string context.
Specifying Fonts	
ADD FONT LIST	Adds an entry to a font list.
CREATE FONT LIST	Creates a new font list.

5.6.1 Creating a Compound String

To create a compound string, pass a text string to one of the following compound string creation routines. (You can also use the UIL built-in function `COMPOUND_STRING` to create compound strings in a UIL module. For more information about this built-in function, see Section 3.2.7.4.)

- `CS STRING` routine
- `STRING` routine
- `LATIN1 STRING` routine

The `CS STRING` routine provides access to all the aspects of a compound string that you can specify, including character set and writing direction. The `STRING` and `LATIN1 STRING` routines are convenience routines that use default values for certain aspects of a compound string. Using the `STRING` routine, you can only specify the character set and writing direction. The `STRING` routine uses default values for all other aspects of a compound string. The `LATIN1 STRING` routine uses default values for all of the aspects of a compound string. The `LATIN1 STRING` routine creates a compound string that uses the ISO Latin1 character set and the left-to-right writing direction. Most English language applications can use the `LATIN1 STRING` convenience routine.

The compound string routines take a standard text string as an argument and create a compound string version of the text. Example 5–6 shows an excerpt from the Hello World! sample application in which the text contained in the label widget is defined. The example uses the `LATIN1 STRING` conversion routine to convert the text into a compound string.

Using the Label, Separator, and Button Widgets

5.6 Working with Compound Strings

Example 5-6 Creating a Compound String

```
.  
. .  
XtSetArg (arglist[0],DwtNlabel,  
    DwtLatin1String("Press button once\nto change label;\ntwice to exit."));  
label = DwtLabelCreate( helloworldmain, "label", arglist, 1 );  
. .  
.
```

Note that the compound string routines allocate memory. Remember to free the memory obtained by the compound string routines when the compound string is no longer needed. Use the **FREE** intrinsic routine to free the memory associated with a compound string.

5.6.2 Creating Compound Strings with Multiple Segments

To create a compound string with multiple segments, create each segment as a separate compound string and then concatenate the strings using the **CS CAT** routine. Use any of the routines described in Section 5.6.1 to create the segments.

Example 5-7 illustrates how to create a compound string with multiple segments.

Example 5-7 Creating a Compound String with Multiple Segments

```
.  
. .  
①#include <cda$def.h>  
. .  
②DwtCompString cstring1 = DwtLatin1String("Compound string text widget");  
③DwtCompString cstring2 = DwtString("Compound string text widget",  
    CDA$K_ISO_LATIN1,  
    DwtDirectionLeftDown );  
④DwtCompString mixed_string = DwtCStrcat( cstring1, cstring2 );  
. .  
.
```

- ① The Compound Document Architecture (CDA) symbol definition file, named *cda\$def.h*, enables the example to use the CDA constants to specify character sets.
- ② The **LATIN1 STRING** compound string routine creates a compound string that uses the ISO Latin1 character set and the left-to-right writing direction.
- ③ The **STRING** compound string routine allows you to specify the character set and writing direction. In the example, the ISO Latin1 character set and the right-to-left writing direction are specified.

Using the Label, Separator, and Button Widgets

5.6 Working with Compound Strings

- ④ The CS CAT compound string routine concatenates the two compound strings to create one compound string containing multiple segments.

5.6.3 Manipulating a Compound String

To compare, copy, determine the length, or determine the text content of a compound string, use one of the following compound string manipulation routines:

- CS BYTE CMP routine
- CS COPY routine
- CS LEN routine
- CS EMPTY routine

To compare two compound strings, use the CS BYTE CMP routine. This routine compares not only the text content of the compound strings, but also the character set and writing direction. The routine returns zero (0) if both compound strings are identical.

To copy a compound string, use the CS COPY routine. This routine makes a byte-for-byte copy of the specified compound string.

To determine the length of a compound string, use the CS LEN routine. The value returned by this routine includes all the components of the compound string, not just the length of the text component.

To determine if a compound string contains any text, use the CS EMPTY routine. The routine returns true (1) if the compound string does not contain text.

5.6.4 Retrieving Information About a Compound String

You can use compound string routines to determine the following information about a compound string:

- Text content
- Character set
- Writing direction

To obtain this information, perform the following steps:

- 1 Obtain an initialized compound string context for the compound string.

The **compound string context** is a data structure that contains information about a particular compound string. You pass a compound string to the STRING INIT CONTEXT routine with the address of a compound string context. The routine fills the compound string context with information about the compound string that you specified.

Using the Label, Separator, and Button Widgets

5.6 Working with Compound Strings

- 2 Extract information about the string from the compound string context.

Use the `GET NEXT SEGMENT` routine to determine the text content, character set, and writing direction of a compound string. If the compound string is made up of more than one segment, the `GET NEXT SEGMENT` routine returns information about the first segment and returns a status value indicating that there are additional segments. The following table lists all of the possible status values returned by the `GET NEXT SEGMENT` routine.

Status	Meaning
<code>DwtEndCS</code>	End of compound string has been reached.
<code>DwtFail</code>	Context is not valid.
<code>DwtSuccess</code>	Normal completion.
<code>DwtTruncate (VAX only)</code>	Text string was truncated to fit in the buffer described by the static descriptor.

- 3 Free the compound string context.

Use the `STRING FREE CONTEXT` routine to free the compound string context obtained by the `STRING INIT CONTEXT` routine.

You can also use the `INIT GET SEGMENT` routine to obtain an initialized compound string context; however, the `STRING INIT CONTEXT` routine is recommended because it offers better performance. If you use the `INIT GET SEGMENT` routine, you do not need to free the compound string context.

Example 5-8 shows how to use the `GET NEXT SEGMENT` routine to extract the text content from the first segment of a compound string. Note that extracting the text content from each segment of a compound string and concatenating the text to create a single text string may not always produce meaningful results.

Example 5-8 Extracting the Text Content from a Compound String

```

    .
    .
    DwtCompString comp_string = DwtLatin1String("My compound string");
    ❶ DwtCompStringContext context;
    char *result;
    long status, charset, direction, lang, rend;
    .
    .
    ❷ status = DwtStringInitContext( &context, comp_string );
    if( status != DwtSuccess )
    {
        printf("Cannot Initialize Compound String Context.");
    }
    else
    {
    ❸ status = DwtGetNextSegment( &context, &result, &charset,
                                &direction, &lang, &rend );
    }
    ❹ DwtStringFreeContext( &context );
    .
    .

```

- ❶ The compound string context will hold information about a particular compound string.
- ❷ The STRING INIT CONTEXT routine initializes the compound string context with information about the compound string named *comp_string*.
- ❸ The GET NEXT SEGMENT routine extracts the information about this compound string from the compound string context. The text content of the compound string is returned as a null-terminated array of text characters in the *result* argument.
- ❹ The STRING FREE CONTEXT routine frees the compound string context obtained earlier by the call to the STRING INIT CONTEXT routine.

5.6.5 Specifying Fonts

To specify the font you want used to display text, you must create a **font list** by using the compound string routine CREATE FONT LIST. A font list is an internal data structure that associates font names with character set identifiers. The XUI Toolkit specifies fonts in font lists because compound strings can employ more than one character set and, consequently, might require more than one font to display these character sets.

To add an additional font specification to an existing font list, use the ADD FONT LIST routine.

Using the Label, Separator, and Button Widgets

5.6 Working with Compound Strings

The program in Example 5–9 creates a font list and uses it to specify the font used in a simple text widget. (Use system default fonts whenever possible to ensure that your application appears well integrated in the VMS DECwindows environment.)

Example 5–9 Specifying a Font

```
#include <stdio>
#include <decw$include/DwtAppl.h>
❶#include <cda$def.h>

Widget toplevel, main_db, text_w;

int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Arg arglist[15];
    int ac = 0;
    ❷ XFontStruct *font;
    ❸ DwtFontList font_list;

    toplevel = XtInitialize("Font Example","exampleclass",NULL, 0, &argc, argv);

    ac = 0;
    XtSetArg( arglist[ac], XtNallowShellResize, TRUE ); ac++;
    XtSetArg( arglist[ac], XtNx, 150 ); ac++;
    XtSetArg( arglist[ac], XtNy, 150 ); ac++;
    XtSetValues( toplevel, arglist, ac );

    ac = 0;
    XtSetArg( arglist[ac], DwtNmarginHeight, 15 ); ac++;

    main_db = DwtDialogBoxCreate( toplevel, "MAINWIN", arglist, ac );

    ❹ font = XLoadQueryFont( XtDisplay( toplevel ),
                            "--Courier-BOLD-R-Normal--*-120--*-M--*-");

    ❺ font_list = DwtCreateFontList( font, CDA$K_ISO_LATIN1 );

    ac = 0;
    ❻ XtSetArg( arglist[ac], DwtNfont, font_list ); ac++;
    XtSetArg( arglist[ac], DwtNvalue, "Sample text" ); ac++;
    XtSetArg( arglist[ac], DwtNx, 20 ); ac++;
    XtSetArg( arglist[ac], DwtNy, 20 ); ac++;
    XtSetArg( arglist[ac], DwtNrows, 1 ); ac++;
    XtSetArg( arglist[ac], DwtNcols, 25 ); ac++;

    text_w = DwtSTextCreate( main_db, "textwidget", arglist, ac );

    ❽ XtFree( font_list );

    XtManageChild( text_w );
    XtManageChild( main_db );
    XtRealizeWidget( toplevel );
    XtMainLoop();
}
```

❶ The Compound Document Architecture (CDA) symbol definition file, named *cda\$def.h*, enables the example to use the CDA constants to specify character sets.

❷ The variable *font* is declared as a pointer to an X font structure.

Using the Label, Separator, and Button Widgets

5.6 Working with Compound Strings

- ③ The variable *font_list* is declared as a font list.
- ④ The `LOAD QUERY FONT Xlib` routine returns a pointer to the specified font. This routine returns the address of an X font structure. If the specified font cannot be loaded, the routine returns a null pointer.
- ⑤ The `CREATE FONT LIST` routine creates a font list. In the example, the first argument to the `CREATE FONT LIST` routine specifies the X font structure returned by the `LOAD QUERY FONT` routine. The second argument to this routine is a constant that identifies the character set.
- ⑥ The font list is used to specify the font the simple text widget will use to display text.
- ⑦ After using the font list, free the memory associated with the font list.

5.7 Defining Accelerators for Button Widgets and Gadgets

The primary mode of access to functions associated with push button widgets and gadgets and toggle button widgets and gadgets is by clicking a mouse button. However, you can also provide access to these same functions using the keyboard. This alternate mode of access is called an **accelerator**.

Defining an accelerator is a three-step process:

- 1 Create an event specification that defines the key or combination of keys used as the accelerator.
- 2 Add the definition of the accelerator to the accelerator table of the widget or gadget.
- 3 Install the event specification in the translation table of another widget higher in the application widget hierarchy that can accept the input focus.

You can optionally include a text label in the widget or gadget to provide a visual cue to users that the widget has an accelerator.

5.7.1 Defining the Accelerator Key or Key Combination

You specify the key or combination of keys that will be the accelerator in a translation table **event specification**. An event specification is a text string that contains the event name (enclosed in angle brackets), the name of the keyboard key, and any modifier keys, such as the control key. The modifier keys precede the event name. Terminate the event specification with a colon. Enclose the entire event specification with quotation marks.

As an example, to define the `Ctrl/B` key combination as an accelerator for a push button, you create the following event specification:

```
"Ctrl<KeyPress>b:"
```

Using the Label, Separator, and Button Widgets

5.7 Defining Accelerators for Button Widgets and Gadgets

In the example, the control key is the modifier key, specified by the abbreviation `Ctrl`. Following the modifier key is the event name enclosed in angle brackets. For an accelerator, the event is the pressing of a keyboard key, specified by the event name `KeyPress`. Following the event name, you specify the keyboard key that will be the accelerator. In the example, this is the letter `b`. The colon terminates the event specification. The entire event specification is enclosed with quotation marks. (For more information about translation table syntax, see Section D.9.4.)

If you are defining the user interface of your application in a UIL module, use the UIL built-in function `TRANSLATION TABLE` to create an accelerator event specification. For more information, see the *VMS DECwindows User Interface Language Reference Manual*.

5.7.2 Adding an Accelerator to a Widget or Gadget

To add an accelerator definition to a widget or gadget, assign the event specification as the value of the `button_accelerator` attribute. Example 5-10 adds an accelerator to a push button widget. Note in the example how the entire event specification is passed as the value of the `button_accelerator` attribute.

Example 5-10 Adding an Accelerator to a Push Button Widget or Gadget

```
.
.
XtSetArg( arglist[0], DwtNx, 10 );
XtSetArg( arglist[1], DwtNy, 40 );
XtSetArg( arglist[2], DwtNactivateCallback, callback_arg );
XtSetArg( arglist[3], DwtNlabel, DwtLatin1String("Hello\nWorld!") );
XtSetArg( arglist[4], DwtNacceleratorText, DwtLatin1String("^b") );
XtSetArg( arglist[5], DwtNbuttonAccelerator, "Ctrl<KeyPress>b:" );
button = DwtPushButtonCreate( helloworldmain, "button", arglist, 6 );
.
.
```

5.7.3 Installing an Accelerator in an Application

After creating the widget or gadget with the accelerator, you must add the accelerator to the translation table of a widget that is higher in the application widget hierarchy and that can accept input focus. The push button widget or gadget with the accelerator cannot accept input focus and so cannot receive keyboard input. Therefore, you must choose a widget in the application widget hierarchy that can accept input focus and install the accelerator on that widget. The main window widget, the dialog box widget, the attached dialog box widget, and the simple text widget are the only XUI Toolkit widgets that accept input focus.

To install an accelerator on a widget, use either the `INSTALL ACCELERATORS` or the `INSTALL ALL ACCELERATORS` intrinsic routine, as in the following example:

```
XtInstallAllAccelerators( main_win, main_win );
```

Using the Label, Separator, and Button Widgets

5.7 Defining Accelerators for Button Widgets and Gadgets

The `INSTALL ACCELERATORS` and the `INSTALL ALL ACCELERATORS` routines both accept the same arguments:

- The widget identifier of the **destination widget**
- The widget identifier of the **source widget**

The destination widget argument is the identifier of the widget on which you want the accelerators installed. The source widget argument is the identifier of the widget that contains the accelerators. For the `INSTALL ACCELERATORS` routine, the source widget is a single widget. For the `INSTALL ALL ACCELERATORS` routine, the source widget is a widget hierarchy. The `INSTALL ALL ACCELERATORS` routine searches for accelerators in the widget specified as the argument as well as all of the widgets below it in the widget hierarchy.

5.7.4 Specifying an Accelerator Label

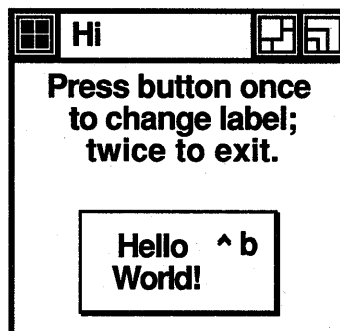
When you define an accelerator for a widget or gadget, you can optionally include a text representation of the accelerator in the widget or gadget. The text representation of the accelerator makes the user of the application aware of the accelerator for the widget or gadget.

To include an accelerator label, pass the label, in the form of a compound string, to the widget or gadget using the `accelerator_text` attribute, as in the following example:

```
XtSetArg(arglist[4], DwtNacceleratorText, DwtLatin1String("^b"));
```

The accelerator label you specify appears in the widget or gadget to the right of the text label. Figure 5-3 shows the appearance of the Hello World! application with the accelerator label "`^b`".

Figure 5-3 Hello World! Application with an Accelerator



ZK-0202A-GE

Using the Label, Separator, and Button Widgets

5.7 Defining Accelerators for Button Widgets and Gadgets

5.7.5 Adding an Accelerator to the Hello World! Sample Application

Example 5–11 modifies the Hello World! application to accept an accelerator. This version of the Hello World! application adds a main window widget to the application widget hierarchy of the Hello World! application to enable the application to accept input focus. The widget must be able to accept keyboard input to make use of accelerators. The example explicitly sets the width and height of the main window widget to 0, causing the main window widget to size itself to fit its children.

Example 5–11 Adding an Accelerator to the Hello World! Application

```
#include <stdio>
#include <decw$include/DwtAppl.h>

static void helloworld_button_activate ();
static DwtCallback callback_arg[2];

int main (argc, argv)
    unsigned int argc;
    char **argv;
{
    Widget toplevel, helloworldmain, button, label, main_win;
    Arg arglist[10];

    toplevel = XtInitialize("Hi", "helloworldclass", NULL, 0, &argc, argv);

    XtSetArg(arglist[0], XtNallowShellResize, TRUE);
    XtSetValues (toplevel, arglist, 1);

    XtSetArg(arglist[0], DwtNacceptFocus, TRUE);
    XtSetArg(arglist[1], DwtNwidth, 0);
    XtSetArg(arglist[2], DwtNheight, 0);
    ❶ main_win = DwtMainWindowCreate (toplevel, "main", arglist, 3);
    XtManageChild (main_win);

    helloworldmain = DwtDialogBoxCreate(main_win, "d_box", arglist, 0);
    XtSetArg(arglist[0], DwtNlabel,
        DwtLatin1String ("Press button once\nto change label;\ntwice to exit.));
    label = DwtLabelCreate( helloworldmain, "label", arglist, 1 );
    XtManageChild( label );

    callback_arg[0].proc = helloworld_button_activate;
    callback_arg[0].tag = 0;
    callback_arg[1].proc = NULL;

    XtSetArg(arglist[0], DwtNx, 10);
    XtSetArg(arglist[1], DwtNy, 40);
    XtSetArg(arglist[2], DwtNactivateCallback, callback_arg);
    XtSetArg(arglist[3], DwtNlabel, DwtLatin1String("Hello\nWorld"));
    ❷ XtSetArg(arglist[4], DwtNacceleratorText, DwtLatin1String("^b"));
    ❸ XtSetArg(arglist[5], DwtNbuttonAccelerator, "Ctrl<KeyPress>b:");

    button = DwtPushButtonCreate( helloworldmain, "button", arglist, 6 );
    XtManageChild(button);

    XtManageChild(helloworldmain);

    XtRealizeWidget (toplevel);
```

(continued on next page)

Using the Label, Separator, and Button Widgets

5.7 Defining Accelerators for Button Widgets and Gadgets

Example 5-11 (Cont.) Adding an Accelerator to the Hello World! Application

```
④ XtInstallAllAccelerators( main_win, main_win );
    XtMainLoop();
}

static void helloworld_button_activate (widget, tag, callback_data)
Widget widget;
char *tag;
DwtAnyCallbackStruct *callback_data;
{
    Arg arglist[2];
    static int call_count = 0;

    call_count += 1;

    switch (call_count) {
        case 1:
            XtSetArg(arglist[0], DwtNlabel, DwtLatin1String("Goodbye\nWorld"));
            XtSetArg(arglist[1], DwtNx, 6);
            XtSetValues (widget, arglist, 2);
            break;

        case 2:
            exit(0);
            break;
    }
}
```

- ① This version of the Hello World! application adds a main window widget at the top of its application widget hierarchy. The main window widget can accept input focus, which an application using accelerators requires. In the argument list used to set the attributes of the main window widget, the example sets the **accept_focus** attribute to true.
- ② In this statement, the example program defines the accelerator label that will appear to the right of the push button label. This information tells the user what accelerator key works with this push button. Note that, as with any other text label intended to appear on the display, the accelerator text must be converted into a compound string. The example converts the text string "^b" into a compound string using the LATIN1 STRING routine.
- ③ The example assigns the event specification as the value of the **button_accelerator** attribute in the argument list used to create the push button widget. The information is passed as a text string. (Note that the event specification does not have to be converted into a compound string.) The sample program defines the Ctrl/B key sequence as the accelerator.
- ④ After the program realizes the entire widget hierarchy, including the widget with the accelerator, the example installs the accelerator in the translation table of an application widget that accepts input focus. The example installs the accelerator on the main window widget, *main_win*. In the INSTALL ALL ACCELERATORS routine, both the

Using the Label, Separator, and Button Widgets

5.7 Defining Accelerators for Button Widgets and Gadgets

source and the destination arguments are the same widget. Used as the source argument, *main_win* represents the entire Hello World! application widget hierarchy. As the destination, *main_win* represents the main window widget on which the accelerators are installed.

6

Creating Menu Widgets

This chapter includes the following:

- An overview of the menu widgets provided by the XUI Toolkit
- A detailed description of how to include a menu widget in your application

6.1 Overview of Menu Widgets

Menu widgets allow you to present users with a list of choices. Users can select an option or activate an application function by clicking a mouse button on a menu item. You would typically use a menu widget to present a list of choices that perform actions. To present long lists of choices, use a list box widget. The list box widget allows users to scroll through long lists of choices. Chapter 8 describes the list box widget.

6.2 Menu Widgets in the XUI Toolkit

A menu widget is a rectangular container for menu items. The XUI Toolkit includes six menu widgets. All of the menu widgets are fundamentally the same; that is, they all are rectangular containers. The menu widgets differ in the type of user interaction they provide. The following are menu widgets in the XUI Toolkit:

- Work area menu widget
- Menu bar widget
- Option menu widget
- Radio box widget
- Pull-down menu widget
- Pop-up menu widget

The work area menu widget is the simplest menu widget. As with the other widgets in the XUI Toolkit, the work area menu widget is the child of a given parent widget and is clipped by that parent. You make work area menus appear on a display and remove them from a display by adding and removing them from their parent's list of managed widgets. The menu bar, option menu, and radio box widgets are all specially configured work area menu widgets.

The pull-down and pop-up menu widgets are **spring-loaded** menu widgets. Spring-loaded menu widgets appear on the display only when a user presses a mouse button. They disappear when the user releases the mouse button. The pull-down menu widget and the pop-up menu widget are not clipped by their parent.

Creating Menu Widgets

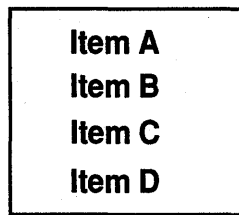
6.2 Menu Widgets in the XUI Toolkit

Pull-down menu widgets are spring-loaded on MB1 or MB2, depending on what type of widget is the parent of the pull-down menu widget. If the parent is a menu bar, work area menu, or option menu widget, the pull-down menu widget is spring-loaded on MB1. If the parent is a pop-up menu widget, the pull-down menu widget is spring-loaded on MB2, since pop-up menu widgets are themselves spring-loaded on MB2. The parent of a pull-down menu widget must be another menu widget.

Pop-up menu widgets are spring-loaded on MB2. Pop-up menu widgets appear on the display wherever the user has positioned the pointer cursor.

Figure 6-1 shows a work area menu widget. For an illustration of other menu widgets, see the illustration of the DECburger application user interface in Figure 1-4. (DECburger does not include a pop-up menu widget. For an illustration of a pop-up menu widget, see Figure 6-9.)

Figure 6-1 Menu Widget



ZK-0208A-GE

6.2.1 Creating Menu Items

You build a menu by creating a menu widget with a group of child widgets or gadgets that implement the menu items. Each item in a menu is a widget or a gadget. The menu widget displays menu items in the order you create them. You can dynamically change the contents of a menu widget at run time by adding and removing the widgets and gadgets that implement menu items from the menu widget's list of managed children.

Menu items can be active or inactive. Active menu items are sensitive to user interaction using a pointing device. Inactive menu items are insensitive to user interaction.

Use the following XUI Toolkit widgets and gadgets to implement menu items:

- Label
- Separator
- Push button

Creating Menu Widgets

6.2 Menu Widgets in the XUI Toolkit

- Toggle button
- Pull-down menu entry

Note: To improve the performance of your application, use gadgets instead of widgets to implement menu items. For example, a pull-down menu widget containing menu items that are gadgets will appear on the display faster than a pull-down menu widget containing menu items that are widgets.

Use label and separator widgets and gadgets to create inactive menu items. Inactive menu items provide descriptive information to the user or organize menu items into logical groups.

Use the push button and toggle button widgets and gadgets to create active menu items. Active menu items gather input from the user or activate application functions. Use the push button widget to implement menu items that carry out actions. Use the toggle button widget to implement menu items that require state information, such as on or off. Section 5.4 describes how to create push button widgets and gadgets; Section 5.5 describes how to create toggle button widgets and gadgets.

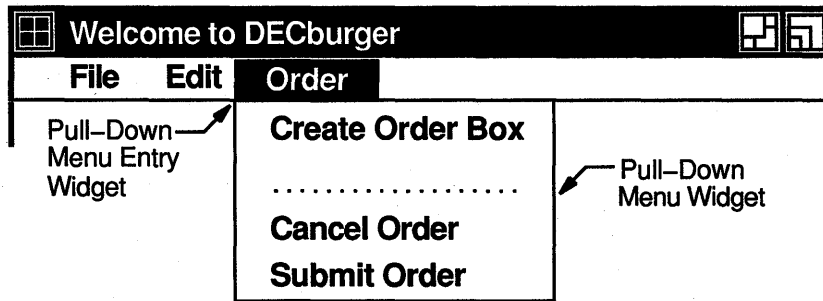
Use the pull-down menu entry widget or gadget to create an active menu item that makes a pull-down menu widget appear on the display. When a user selects the pull-down menu entry widget or gadget, the pull-down menu widget you associated with it appears on the display. This is the only way a user can access a pull-down menu.

Note the distinction between the pull-down menu widget and the pull-down menu entry widget (or gadget). A pull-down menu widget, like all menu widgets, is the rectangular container. A pull-down menu entry widget is a push button-like widget through which users access a pull-down menu widget. For example, you could create a menu bar widget containing three menu items. Each menu item is a pull-down menu entry widget. When a user selects an item in the menu bar widget, the pull-down menu widget associated with the pull-down menu entry widget appears on the display immediately below the menu item (or immediately to the right of the item for vertically oriented menus). Figure 6-2 illustrates these widgets.

Creating Menu Widgets

6.2 Menu Widgets in the XUI Toolkit

Figure 6-2 Relationship of Pull-Down Menu Widget and Pull-Down Menu Entry Widget or Gadget



ZK-0446A-GE

You can specify the text string or pixmap that the pull-down menu entry widget will contain. The pull-down menu entry gadget does not support pixmaps. Pull-down menu entry widgets or gadgets can also contain a **hotspot**. A hotspot is a graphical image that is sensitive to user input using a mouse button. Pull-down menu entry widgets and gadgets contain hotspots when they are used to create nested pull-down menu widgets. (For more information about nesting menus, see Section 6.2.2.) In the pull-down menu entry widget, you can specify the pixmap used for the hotspot. The pull-down menu entry gadget does not allow you to specify the hotspot pixmap.

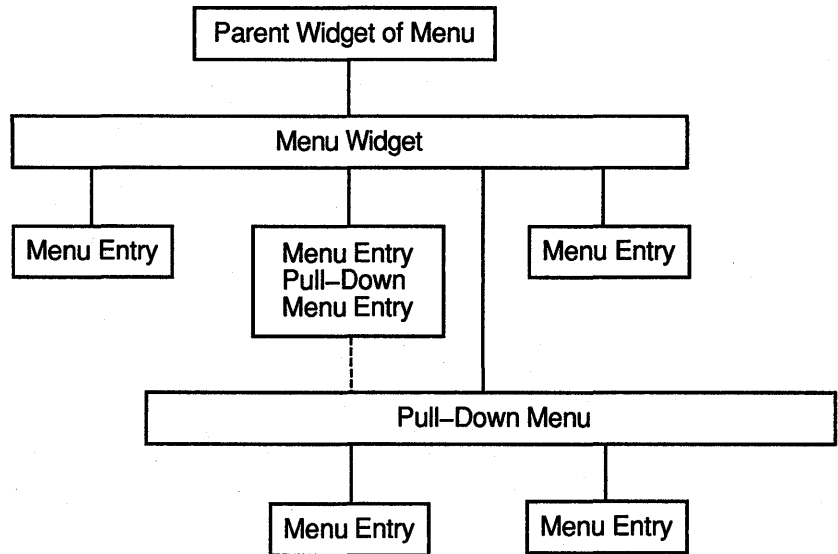
6.2.2 Nesting Menu Widgets

By including a pull-down menu entry widget or gadget in a pull-down menu widget, you can create a cascade of nested menus (also called **submenus**). Figure 6-3 illustrates the application widget hierarchy of a pull-down menu widget containing a nested pull-down menu widget.

Creating Menu Widgets

6.2 Menu Widgets in the XUI Toolkit

Figure 6-3 Widget Hierarchy of Nested Pull-Down Menu Widgets



ZK-0205A-GE

You can nest an unlimited number of menu widgets. However, the *XUI Style Guide* recommends using no more than four levels of nested pull-down menu widgets.

A pull-down menu widget activated from within another pull-down menu, option menu, or pop-up menu widget appears on the display to the immediate right of the pull-down menu entry widget that triggers it. For this reason, these pull-down menu widgets are sometimes called pull-right menu widgets. Pull-right menu widgets are functionally identical to pull-down menu widgets.

6.3 Creating a Work Area Menu Widget

To create a work area menu widget, perform the following steps:

- 1 Create the work area menu widget.

Use any of the widget creation mechanisms listed in Table 6-1. The choice of mechanism depends on the attributes of the widget you need to access.

DECLIT AA VAX MG21B

VMS DECwindows guide to
application programming

Creating Menu Widgets

6.3 Creating a Work Area Menu Widget

Table 6–1 Work Area Menu Widget Creation Mechanisms

High-level routine	Use the MENU routine to create a work area menu widget. Specify the type of menu in the format argument.
Low-level routine	Use the MENU CREATE routine to create a work area menu widget.
UIL object type	Use the work_area_menu object type to define a work area menu widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the work area menu widget according to this definition.

2 Create the children of the work area menu widget.

Use any of the widget creation mechanisms to create the widgets that you want to appear as items in the menu. The child widgets appear in the menu widget in the same order that you create them.

3 Manage the children of the work area menu widget.

Use the intrinsic routine MANAGE CHILD to manage a single child or the MANAGE CHILDREN routine to manage a group of children at the same time.

4 Manage the work area menu widget.

Use the intrinsic routine MANAGE CHILD to manage the widget.

After you complete these steps, if the parent of the work area menu widget has been realized, the work area menu widget and all of its children will appear on the display.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available using a high-level routine, use the SET VALUES intrinsic routine after the widget has been created.) Table 6–2 lists the attributes you can set if you use the high-level routine MENU to create a work area menu widget. Pass the values of these attributes as arguments to the routine.

Table 6–2 Attributes Accessible Using the High-Level Routine MENU

x	Specifies the x-coordinate of the upper left corner
y	Specifies the y-coordinate of the upper left corner
format	Specifies the type of menu: pull-down, pop-up, or work area
orientation	Specifies whether the menu has a horizontal or vertical orientation
map_callback	Specifies the address of a callback routine list
entry_callback	Specifies the address of a callback routine list
help_callback	Specifies the address of a callback routine list

Creating Menu Widgets

6.3 Creating a Work Area Menu Widget

Example 6-1 creates a typical work area menu widget. The example creates a work area menu widget with three push button widgets as its children. The push button widgets are the menu items.

Example 6-1 Building a Work Area Menu

```
Widget toplevel, parent_widget, menu;
WidgetList menu_items[3];

static void button1_callback();
static void button2_callback();
static void button3_callback();

static DwtCallback callback_list[2];

build_menu()
{
    Arg arglist[5];
    int count = 0;

    XtSetArg( arglist[0], DwtNmarginWidth, 20);

    ❶ menu = DwtMenuCreate( parent_widget, "menu", arglist, 1 );
    callback_list[0].proc = button1_callback;
    callback_list[0].tag = 0;
    callback_list[1].proc = NULL;

    XtSetArg( arglist[0], DwtNactivateCallback, callback_list );
    XtSetArg( arglist[1], DwtNlabel,
              DwtLatin1String("Menu Item One") );

    ❷ menu_items[count++] = DwtPushButtonCreate( menu, "button1", arglist, 2);
    callback_list[0].proc = button2_callback;
    callback_list[0].tag = 0;
    callback_list[1].proc = NULL;

    XtSetArg( arglist[0], DwtNactivateCallback, callback_list );
    XtSetArg( arglist[1], DwtNlabel,
              DwtLatin1String("Menu Item Two") );

    menu_items[count++] = DwtPushButtonCreate( menu, "button2", arglist, 2);
    callback_list[0].proc = button3_callback;
    callback_list[0].tag = 0;
    callback_list[1].proc = NULL;

    XtSetArg( arglist[0], DwtNactivateCallback, callback_list );
    XtSetArg( arglist[1], DwtNlabel,
              DwtLatin1String("Menu Item Three") );

    menu_items[count++] = DwtPushButtonCreate( menu, "button3", arglist, 2);

    ❸ XtManageChildren( menu_items, count );
}
```

(continued on next page)

Creating Menu Widgets

6.3 Creating a Work Area Menu Widget

Example 6–1 (Cont.) Building a Work Area Menu

```
⑤ XtManageChild( menu );  
}  
.  
.  
.
```

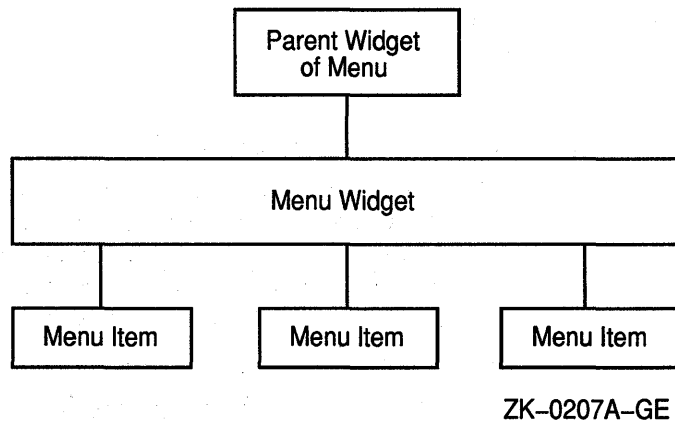
- ① The example declares variables to hold the widget identifiers for the widgets in the application widget hierarchy. The variable *menu* will hold the widget identifier of the work area menu widget. The variable *menu_items* is an array of widget identifiers that will hold the identifiers for the widgets that are the menu items.
- ② The example creates the work area menu widget using the low-level routine `MENU CREATE`. In the argument list, the example specifies a margin width of 20 pixels in the **margin_width** attribute. The argument list is passed to the widget creation routine along with a count of the number of attributes set in the argument list.
- ③ After creating the work area menu widget, the sample program creates the widgets that will be menu items. In the example, all the menu items are push button widgets. Each push button widget is a child of the work area menu widget. The sample program creates the push button widgets using the low-level routine `PUSH BUTTON CREATE`. For each push button widget, the sample program specifies a callback routine list, *callback_list*, and the text of the label the push button widget will contain. The widget identifier returned by each call to the `PUSH BUTTON CREATE` routine is stored in the array of widget identifiers, *menu_items*.
- ④ The example manages all the children of the work area menu widget in a single call to the intrinsic routine `MANAGE CHILDREN`. All the widgets managed in this call must be children of the same parent. The sample program passes the address of the array of widget identifiers and the number of widgets in the array as arguments to the routine.
- ⑤ The example manages the work area menu widget using the `MANAGE CHILD` intrinsic routine.

Figure 6–4 illustrates the application widget hierarchy of the menu created in Example 6–1.

Creating Menu Widgets

6.3 Creating a Work Area Menu Widget

Figure 6-4 Widget Hierarchy of a Work Area Menu



6.3.1 Customizing a Work Area Menu Widget

The attributes of the work area menu widget enable you to customize the following aspects of its appearance and functioning:

- Size
- Arrangement of menu items
- Margins and spacing between menu items
- Alignment of menu items
- Radio button exclusivity
- Type of widget that can be a menu item

By default, the menu widget can change many of the visible attributes of its child widgets to make the menu items appear uniform. For example, the menu widget can set the border size, align labels, change margin settings, and change the shape and visibility of a toggle button indicator. However, you can specify that the menu widget leave these attributes unchanged by setting the `change_vis_atts` attribute to false (this attribute is true by default).

6.3.1.1 Specifying the Size of a Work Area Menu Widget

You can specify the size of a work area menu widget using the common widget attributes `width` and `height`. Specify each dimension in pixels. By default, work area menu widgets size themselves to accommodate their children.

Work area menu widgets will grow to fit additional children to the degree the parent of the menu widget allows. When resized, work area menu widgets recalculate the layout of their children. If the work area menu widget cannot grow to accommodate its new children, the children are clipped.

Creating Menu Widgets

6.3 Creating a Work Area Menu Widget

6.3.1.2 Specifying the Arrangement of Menu Items

Use the **menu_packing**, **menu_num_columns**, and **orientation** attributes to specify the arrangement of the child widgets in a work area menu widget.

Use the **menu_packing** attribute to specify that the work area menu widget arrange its children in columns. If you do this, you can also specify the number of columns in the **menu_num_columns** attribute.

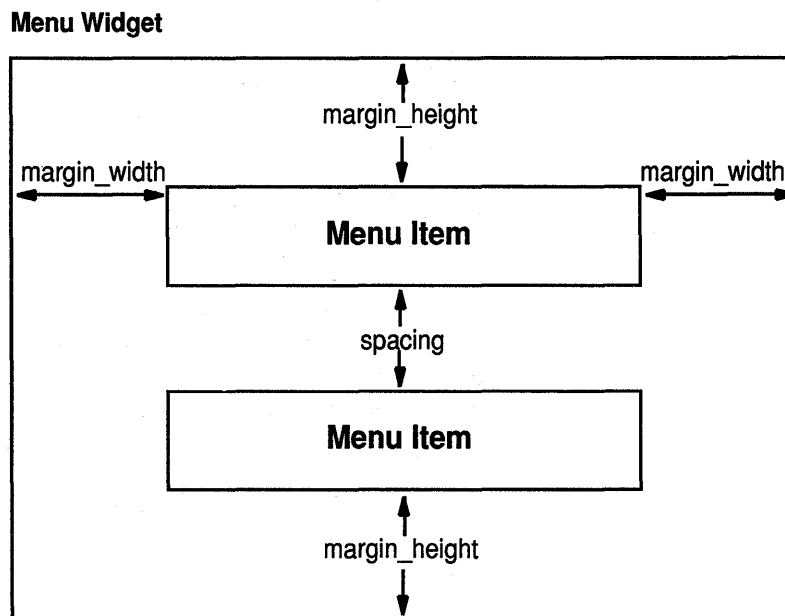
By default, work area menu widgets attempt to fit all their children as efficiently as possible by wrapping them when necessary. Wrapping means the work area menu widget starts a new column or row if the number of children would cause the work area menu widget to grow beyond the limitations set on the work area menu widget by its parent. You can disable this default behavior using the **menu_packing** attribute. In this case, you can determine the individual position of each child widget by setting their **x** and **y** attributes.

6.3.1.3 Specifying Margins and Spacing

Use the **margin_width**, **margin_height**, and **spacing** attributes to determine the amount of space surrounding each child of a work area menu widget. Specify these attributes in pixels.

Figure 6-5 illustrates the margins in a work area menu widget.

Figure 6-5 Laying Out Menu Items



ZK-0200A-GE

You can also control the margins of the widgets that implement the menu items by using the **adjust_margin** attribute. If you set this attribute to true, the work area menu widget sets the internal margins of the

Creating Menu Widgets

6.3 Creating a Work Area Menu Widget

widgets that are the menu items so that the text they contain is aligned. (Section 5.2.1.3 describes the margins of the label, separator, push button and toggle button widgets and gadgets.)

6.3.1.4 Determining Menu Item Alignment

Use the **menu_alignment** and **entry_alignment** attributes to control how the text in each child widget of the work area menu widget lines up. If you set **menu_alignment** to true, the work area menu widget aligns the text contained in each child widget of the work area menu widget. Choose the type of alignment in the **entry_alignment** attribute. The text can be centered, aligned to the right margin, or aligned to the left margin of the child widget. The XUI Toolkit defines constants that indicate each of these values. See the *VMS DECwindows Toolkit Routines Reference Manual* for these constants.

You can also specify whether the active area of the menu item should extend to the full width and height of the menu or whether it should follow the true length of the menu item. If you set the **menu_extend_last_row** attribute to true, the menu widget enlarges the active area of menu items with shorter labels to match the length of the longest menu item. Likewise, the height of the active area of the shortest menu item is extended to match the height of the tallest menu item.

6.3.1.5 Specifying Radio Button Exclusivity

Use the **menu_radio** attribute to specify that only one item in a work area menu widget can be selected at a time. This restriction is called radio button exclusivity. If you set this attribute to true, you can also specify that one item in the work area menu widget must always be selected by setting the **menu_always_one** attribute to true.

A radio box widget is a work area menu with the **menu_radio** attribute set to true by default.

6.3.1.6 Restricting Menu Items to Classes of Widgets

Use the **menu_is_homogeneous** and the **menu_entry_class** attribute to restrict the type of widgets that can be children of a work area menu widget. By setting the **menu_is_homogeneous** attribute to true, you specify that the work area menu widget accept only one class of widget as children. Specify the class of widgets that is allowed as children in the **menu_entry_class** attribute by class name. Table 6-3 lists the widget class names of the XUI Toolkit widgets commonly used as menu items.

Creating Menu Widgets

6.3 Creating a Work Area Menu Widget

Table 6–3 XUI Toolkit Widget and Gadget Class Names

Class Name	Widgets and Gadgets
labelwidgetclass	Label widgets
separatorwidgetclass	Separator widgets
pushbuttonwidgetclass	Push button widgets
togglebuttonwidgetclass	Toggle button widgets
labelgadgetclass	Label gadgets
separatorgadgetclass	Separator gadgets
pushbuttongadgetclass	Push button gadgets
togglebuttongadgetclass	Toggle button gadgets
pulldownwidgetclass	Pull-down menu entry widgets

Note that when you restrict menu items to a certain class, widgets that are subclasses of that widget will be excluded. For example, if you restrict menu items to only the label widget, the push button and toggle button widgets, which are subclasses of the label widget, will be excluded.

6.3.2 Associating Callback Routines with a Work Area Menu Widget

The input capabilities of a work area menu widget are provided mainly by the child widgets it contains. For example, in a work area menu containing push button widgets, it is the push button widgets that perform callbacks when activated by a user. However, you can specify that all the callbacks associated with the child widgets be redirected to a common callback routine.

To associate a common callback routine with a work area menu widget, pass a callback routine list to the work area menu widget in the **entry_callback** attribute. If you do not specify a callback routine in this argument, each child widget executes its own callback routine.

You can also associate a help callback routine with a work area menu widget. To do this, pass a callback list to the widget in the **help_callback** attribute. The application executes this routine when a user presses the Help key while positioning the pointer cursor in an inactive area of the work area menu widget and pressing MB1.

6.4 Creating a Pull-Down Menu Widget

To create a pull-down menu widget, perform the following steps:

- 1 Create the pull-down menu widget.

Use any of the widget creation mechanisms listed in Table 6–4. The choice of mechanism depends on the attributes of the pull-down menu you need to access.

Creating Menu Widgets

6.4 Creating a Pull-Down Menu Widget

Table 6–4 Pull-Down Menu Widget Creation Mechanisms

High-level routine	Use the MENU routine to create a pull-down menu widget. Specify the type of menu in the format argument.
Low-level routine	Use the MENU PULLDOWN CREATE routine to create a pull-down menu widget.
UIL object type	Use the pulldown_menu object type to define a pull-down menu widget in a UIL module. At run time, the DRM routine FETCH WIDGET will create the pull-down menu widget according to this definition.

2 Create the children of the pull-down menu widget.

Use any of the widget creation mechanisms to create the widgets that you want to appear as items in the pull-down menu widget. (For information about creating label, separator, push button, or toggle button widgets or gadgets, see Chapter 5.) The children appear in the pull-down menu widget in the same order that you create them.

3 Create a pull-down menu entry widget or gadget.

Use any of the mechanisms listed in Table 6–5 to create a pull-down menu entry widget or gadget. The choice of mechanism depends on the attributes of the widget or gadget to which you want to assign values. Pass the identifier of the pull-down menu widget you want associated with the pull-down menu entry widget or gadget in the **sub_menu_id** attribute of the pull-down menu entry widget.

Table 6–5 Pull-Down Menu Entry Widget and Gadget Creation Mechanisms

Mechanism	Widget	Gadget
High-level routine	Use the PULL DOWN MENU ENTRY routine to create a pull-down menu entry widget.	There is no high-level gadget creation routine.
Low-level routine	Use the PULL DOWN MENU ENTRY CREATE routine to create a pull-down menu entry widget.	Use the PULL DOWN MENU ENTRY GADGET CREATE routine to create a pull-down menu entry gadget.
UIL object type	Use the pulldown_entry object type to define a pull-down menu entry widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.	Use the pulldown_entry object type with the gadget qualifier.

4 Manage the children of the pull-down menu widget.

Use the intrinsic routine **MANAGE CHILD** to manage a single child or the intrinsic routine **MANAGE CHILDREN** to manage a group of children at the same time.

5 Manage the pull-down menu entry widget or gadget.

Use the intrinsic routine **MANAGE CHILD** to manage the widget or gadget.

Creating Menu Widgets

6.4 Creating a Pull-Down Menu Widget

Note that you do not manage the pull-down menu widget.

After you complete these steps, if the parent of the pull-down menu widget has been realized, the pull-down menu entry widget or gadget will appear on the display. The pull-down menu widget and all of its managed children do not appear on the display until a user activates the pull-down menu entry widget or gadget by pressing MB1.

Low-level and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available using a high-level routine, use the SET VALUES intrinsic routine after the widget has been created.) Table 6-6 lists the attributes you can set if you use the high-level routine MENU to create a pull-down menu widget. Pass the values of these attributes as arguments to the routine.

Table 6-6 Attributes Accessible Using the High-Level Routine MENU

x	Specifies the x-coordinate of the upper left corner.
y	Specifies the y-coordinate of the upper left corner.
format	Specifies the type of menu: pull-down, pop-up, or work area.
orientation	Specifies whether the menu has a horizontal or vertical orientation.
map_callback	Specifies the address of a callback routine list.
entry_callback	Specifies the address of a callback routine list.
help_callback	Specifies the address of a callback routine list.

Example 6-2 in Section 6.5 illustrates how the pull-down menu widgets used in the menu bar widget of the DECburger sample application are created.

6.4.1 Customizing the Appearance of a Pull-Down Menu Widget

The pull-down menu widget supports the same set of attributes as the work area menu widget. For information about customizing a work area menu widget, see Section 6.3.1.

6.4.2 Associating Callback Routines with a Pull-Down Menu Widget

The pull-down menu widget supports the same callback attributes as the work area menu widget. These callback attributes are described in Section 6.3.2.

In addition, with the pull-down menu widget, you can associate callback routines that get executed when the pull-down menu widget is about to appear on the display (be mapped) or has disappeared from the display (been unmapped). To associate a callback routine with these callbacks, pass a callback routine list to the **map_callback** or **unmap_callback** attributes.

Creating Menu Widgets

6.4 Creating a Pull-Down Menu Widget

For example, you could write a map callback routine that creates the children of the pull-down menu widget only when it is about to be mapped. In this way, you perform this processing only when necessary, saving on application startup time.

The pull-down menu entry widget and gadget, in addition to causing a pull-down menu widget to appear on the display, perform callbacks to your application when activated. Using the **pulling_callback** attribute, you can associate a callback routine with a pull-down menu entry widget or gadget that gets called immediately before the pull-down menu widget is mapped. For example, you could use this callback to defer creation of the pull-down menu widget until it is needed.

6.5 Creating a Menu Bar Widget

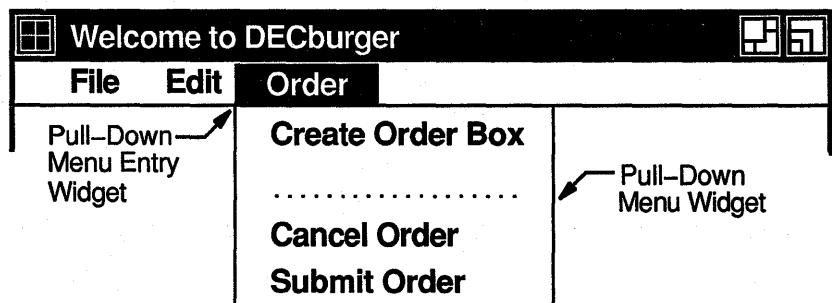
A menu bar widget can have only the following widgets or gadgets as children:

- Label widgets or gadgets
- Separator widgets or gadgets
- Pull-down menu entry widgets or gadgets
- Pull-down menu widgets

Use the label and separator widgets or gadgets to create inactive menu items. Use the pull-down menu entry widget or gadget, which causes pull-down menu widgets to appear on the display, to create active menu items. The pull-down menu widget is also a child of the menu bar widget; however, it does not appear as a visible item in the menu bar.

Figure 6-6 shows the menu bar from the DECburger sample application. In the illustration, the Order pull-down menu widget has been selected.

Figure 6-6 DECburger Menu Bar with a Pull-Down Menu Selected



ZK-0446A-GE

To create a menu bar widget, perform the following steps:

- 1 Create the menu bar widget.

Creating Menu Widgets

6.5 Creating a Menu Bar Widget

Use any of the widget creation mechanisms listed in Table 6–7. The choice of mechanism depends on which attributes of the menu bar widget you need to access.

Table 6–7 Menu Bar Widget Creation Mechanisms

High-level routine	Use the MENU BAR routine to create a menu bar widget.
Low-level routine	Use the MENU BAR CREATE routine to create a menu bar widget.
UIL object type	Use the menu_bar object type to define a menu bar widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

- 2 Create the pull-down menu widgets associated with items in the menu bar widget.

Section 6.4 describes how to create a pull-down menu widget.

- 3 Create the children of the menu bar widget.

For information about creating label or separator widgets or gadgets, see Chapter 5. For information about creating a pull-down menu entry widget, see Section 6.4. The children appear in the menu bar widget in the same order that you create them.

Each pull-down menu entry widget that is a child of the menu bar widget must have an associated pull-down menu widget. Pass the widget identifier of the pull-down menu widget to the pull-down menu entry widget using the `sub_menu_id` attribute.

- 4 Manage the children of the menu bar widget.

Use the intrinsic routine `MANAGE CHILD` to manage a single child of the menu bar widget. Use the intrinsic routine `MANAGE CHILDREN` to manage a group of children of the menu bar widget at one time.

- 5 Manage the menu bar widget.

Use the intrinsic routine `MANAGE CHILD` to manage the menu bar widget.

After you complete these steps, if the parent of the menu bar widget has been realized, the menu bar widget will appear on the display. Note that you do not manage the pull-down menu widgets, even though they are children of the menu bar widget. These widgets get managed when a user activates a pull-down menu entry widget or gadget.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available using the high-level routine, use the `SET VALUES` intrinsic routine after the widget has been created.) Table 6–8 lists the attributes you can set if you use the high-level routine `MENU BAR` to create a menu bar widget. Pass the values of these attributes as arguments to the routine.

Creating Menu Widgets

6.5 Creating a Menu Bar Widget

Table 6-8 Attributes Accessible Using the High-Level Routine MENU BAR

entry_callback	Address of a callback routine list
help_callback	Address of a callback routine list

Example 6-2 is the section of the UIL module in which the DECburger menu bar widget is specified.

Example 6-2 Creating the Menu Bar Widget in the DECburger Application

```
.
.
.
①object
    s_menu_bar : menu_bar {
②        arguments {
            orientation = DWT$C_ORIENTATION_HORIZONTAL;
            spacing      = 15;
        };
        controls {
③            pulldown_entry file_menu_entry;
            pulldown_entry edit_menu_entry;
            pulldown_entry order_menu_entry;
        };
    };
④object
    file_menu_entry : pulldown_entry {
        arguments {
            label_label = k_file_label_text;
        };
        controls {
            pulldown_menu file_menu;
        };
    };
⑤object
    file_menu : pulldown_menu {
        arguments {
            label_label = k_file_label_text;
        };
        controls {
            push_button m_print_button;
            push_button m_quit_button;
        };
        callbacks {
            entry = procedure activate_proc (0);
        };
    };
```

(continued on next page)

Creating Menu Widgets

6.5 Creating a Menu Bar Widget

Example 6-2 (Cont.) Creating the Menu Bar Widget in the DECburger Application

```
⑥object
    m_print_button : push_button {
        arguments {
            label_label = k_print_dot_label_text;
        };
        callbacks {
            activate = procedure activate_proc (k_nyi);
        };
    };
⑦
.
.
.
```

- ① In this UIL object declaration, DECburger defines an object named *s_menu_bar* as a menu bar widget.
- ② In the argument section of the menu bar widget definition, DECburger assigns values to two menu bar widget attributes. The first attribute defines the orientation of the menu bar as horizontal. The second attribute defines the amount of space between the items in pixels.
- ③ In the controls section of the menu bar widget definition, DECburger specifies that the menu bar widget has three pull-down menu entry widgets as its children. DECburger names these widgets *file_menu_entry*, *edit_menu_entry*, and *order_menu_entry*.
- ④ In the UIL object declaration for the *file_menu_entry* pull-down menu entry widget, DECburger specifies the text that will appear in the pull-down menu entry widget as the value of the **label** attribute (called *label_label* in UIL).

As a convenience, UIL allows you to specify a pull-down menu widget as a child of a pull-down menu entry widget (in the controls section of the object declaration). The pull-down menu widget is actually a child of the menu bar widget. If you use high- or low-level routines to create a pull-down menu widget used as a child of a menu bar widget, specify the menu bar widget as the parent.

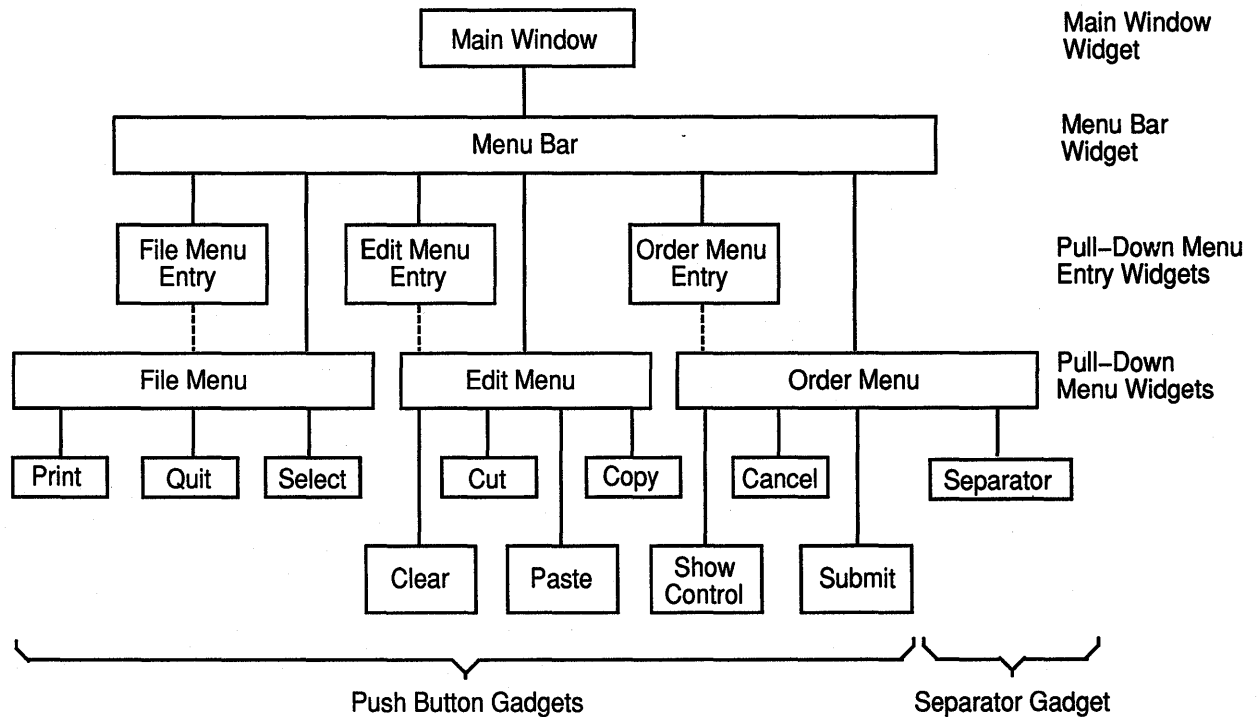
- ⑤ The next widget definition is for the pull-down menu widget that implements the File choice. This pull-down menu widget has two push button widgets as its children (in its controls list).
- ⑥ One of the children of the File pull-down menu is the Print push button widget. This definition defines the label and the callback routine used with the push button widget.
- ⑦ The UIL module goes on to create each child of the menu bar widget.

Figure 6-7 shows the widget hierarchy of the menu bar widget in the DECburger application.

Creating Menu Widgets

6.5 Creating a Menu Bar Widget

Figure 6-7 Widget Hierarchy of the DECburger Menu Bar Widget



ZK-0206A-GE

6.5.1 Customizing a Menu Bar Widget

The menu bar widget supports the same attributes as the work area menu widget. Use the attributes described in Section 6.3.1 to size, position, and customize aspects of the menu bar widget.

6.6 Creating an Option Menu Widget

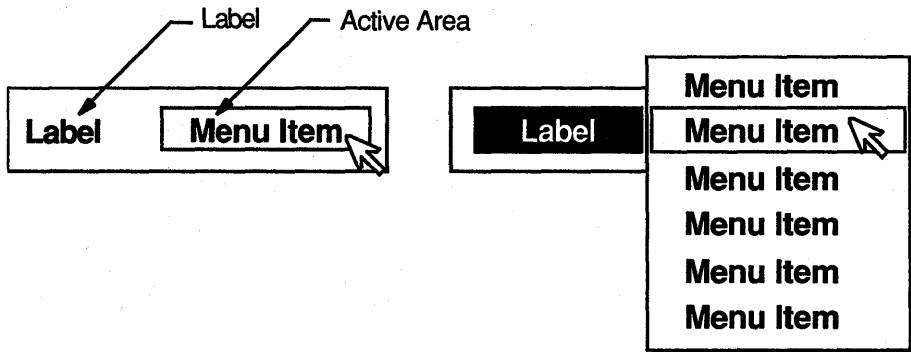
An option menu widget is a rectangular box that contains a descriptive text label and the current value of the menu. The current value of the option menu widget is the active area of an option menu. When a user presses MB1 in the active area of an option menu, a pull-down menu widget appears on the display. The pull-down menu widget contains the list of options.

Figure 6-8 illustrates an option menu widget and its components, both before and after the option menu widget is selected by a user.

Creating Menu Widgets

6.6 Creating an Option Menu Widget

Figure 6-8 Option Menu Widget



ZK-0442A-GE

An option menu widget can have only one child: a pull-down menu widget. When used with an option menu widget, a pull-down menu widget does not require an associated pull-down menu entry widget. Instead, you pass the widget identifier of the pull-down menu widget to the option menu widget using the **sub_menu_id** attribute.

To create an option menu widget, perform the following steps:

- 1 Create the pull-down menu widget.
This is the pull-down menu widget that the option menu widget will invoke. Section 6.4 describes how to create a pull-down menu widget.
- 2 Create the option menu widget.
Use any of the widget creation mechanisms listed in Table 6-9. The choice of mechanism depends on the attributes of the option menu you need to access. Section 6.6.1 describes the attributes supported by the work area menu widget.

Table 6-9 Option Menu Widget Creation Mechanisms

High-level routine	Use the OPTION MENU routine to create an option menu widget.
Low-level routine	Use the OPTION MENU CREATE routine to create an option menu widget.
UIL object type	Use the option_menu object type to define an option menu widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

When you create the option menu widget, you must pass it the widget identifier of the pull-down menu widget in the **sub_menu_id** attribute.

- 3 Manage the option menu widget.
Use the intrinsic routine **MANAGE CHILD**.

Creating Menu Widgets

6.6 Creating an Option Menu Widget

After you complete these steps, if the parent of the option menu widget has been realized, the option menu widget will appear on the display. The pull-down menu widget associated with the option menu widget only appears on the display when a user activates the option menu widget by pressing MB1.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes that are not available using the high-level routine, use the SET VALUES intrinsic routine after the widget has been created.) Table 6-10 lists the attributes you can set if you use the high-level routine OPTION MENU to create an option menu widget. Pass the values of these attributes as arguments to the routine.

Table 6-10 Attributes Accessible Using the High-Level Routine OPTION MENU

x	Specifies the x-coordinate of the upper left corner of the widget.
y	Specifies the y-coordinate of the upper left corner of the widget.
labl ¹	Specifies the text of the descriptive label.
orientation	Specifies whether the menu has a horizontal or vertical orientation.
entry_callback	Specifies the address of a callback routine list.
help_callback	Specifies the address of a callback routine list.

¹The high-level routines use this spelling for the label attribute to avoid conflicts with programming languages in which "label" is a reserved word.

Example 6-3 shows how the option menu widget used in the DECBurger application is created in the DECBurger UIL module.

Creating Menu Widgets

6.6 Creating an Option Menu Widget

Example 6-3 Creating the Option Menu Widget in the DECburger Application

```
.
.
.
①object
  fries_option_menu : option_menu {
    arguments {
      x = 130;
      y = 22;
      label_label = k_size_label_text;
      menu_history = push_button medium_fries;
    };
    controls {
      pulldown_menu fries_menu;
    };
  };

②object
  fries_menu : pulldown_menu {
    controls {
      push_button    tiny_fries;
      push_button    small_fries;
      push_button    medium_fries;
      push_button    large_fries;
      push_button    huge_fries;
    };
  };

③object
  tiny_fries : push_button {
    arguments {
      label_label = k_tiny_label_text;
    };
    callbacks {
      activate = procedure activate_proc (k_fries_tiny);
    };
  };
.
.
.
```

- ① DECburger defines the option menu widget in this UIL object declaration. DECburger positions the option menu widget within the parent dialog box by assigning values to the `x` and `y` attributes. In addition, DECburger specifies the label the option menu will contain as the value of the `label_label` attribute (called `label_label` in UIL). DECburger defines the initial value of the option menu widget by specifying the widget identifier of a child of the pull-down menu widget in the `menu_history` attribute. In the example, the push button widget named `medium_fries` is the initial value of the option menu widget.
- ② In the controls list section of the option menu widget declaration, DECburger defines the children of the option menu widget. For an option menu this is a pull-down menu widget. Note that in UIL, the pull-down menu widget appears in the controls list section of the object declaration. Using the high- or low-level routines, you pass the widget

Creating Menu Widgets

6.6 Creating an Option Menu Widget

identifier of the pull-down menu widget to the option menu widget in its **sub_menu_id** attribute.

- After defining the option menu widget, DECBurger defines the pull-down menu widget. The controls list of the pull-down menu widget declaration lists the children of the pull-down menu widget. These children will be the items in the pull-down menu widget.

The DECBurger UIL module goes on to define each of the five push button widgets that are children of the pull-down menu widget.

6.6.1 Customizing an Option Menu Widget

The option menu widget supports the same attributes as the work area menu widget. Use the attributes described in Section 6.3.1 to size, position, and customize aspects of the option menu widget.

In addition, the option menu widget supports other attributes that enable you to specify the initial value of the option menu widget and the content of the descriptive label it contains.

6.6.1.1 Specifying the Initial Value of an Option Menu Widget

The **menu_history** attribute contains the widget identifier of the child of the menu widget that was last selected. Use the **menu_history** attribute to specify the initial value of the option menu widget. For the option menu widget, the selected item is actually a child of the pull-down menu widget that implements the option menu widget's list of choices. All menu widgets support the **menu_history** attribute; however, the option menu widget also displays the value of this attribute in its active area.

Example 6-4 shows how DECBurger creates the default selection of the option menu widget it uses in its interface. DECBurger passes the widget identifier of the push button widget child of the pull-down menu widget as the value of the **menu_history** attribute.

Example 6-4 Creating an Option Menu Widget with an Item Selected

```
.
.
.
object
  fries_option_menu : option_menu {
    arguments {
      x = 130;
      y = 22;
      label_label = k_size_label_text;
      menu_history = push_button medium_fries;
    };
    controls {
      pulldown_menu fries_menu;
    };
  };
};
```

Creating Menu Widgets

6.6 Creating an Option Menu Widget

6.6.1.2 Specifying the Label in an Option Menu Widget

Use the `label` attribute to specify the descriptive text contained in an option menu widget. The option menu widget is the only menu widget that supports a label attribute. Other menu widgets can have label widgets as children, but only the option menu widget supports a label as an attribute. Specify this label as a compound string.

6.7 Creating a Pop-Up Menu Widget

You create a pop-up menu widget as you would any other menu widget. Create the pop-up menu widget, then create the widgets that will be items in the menu as its children. Pop-up menu widgets differ from other menu widgets in how you make them appear on the display. To make a pop-up menu widget appear on the display, you must modify the action table and the translation table of the pop-up menu widget's parent.

To create a pop-up menu widget, perform the following steps:

- 1 Create an action procedure that displays the pop-up menu widget.

To create a pop-up menu widget, you must create an action procedure that manages the pop-up menu widget when a user of the application presses MB2. Section 6.7.1 describes how to create an action procedure.

- 2 Create a translation table entry for the new action procedure.

A translation table maps an event to the name of an action procedure. Section 6.7.2 describes how to create a translation table entry.

- 3 Create an action table entry for the new action procedure.

An action table maps the name of an action procedure to its address. Section 6.7.2 describes how to create an action table entry.

- 4 Create the parent of the pop-up menu widget.

You must create the parent of a pop-up menu widget before you create the pop-up menu widget itself. Use any of the three widget creation mechanisms.

- 5 Manage the parent of the pop-up menu widget.

Use the intrinsic routine `MANAGE CHILD` to manage the widget.

- 6 Add the new action procedure to the translation table of the parent widget of the pop-up menu widget.

You must add the pop-up action procedure to the action table of the parent widget. When the user presses MB2 in the parent widget, the parent widget will activate the pop-up action procedure. Section 6.7.2 describes this procedure.

- 7 Realize the parent of the pop-up menu widget.

You must realize the parent of the pop-up menu widget before you create the pop-up menu widget.

Creating Menu Widgets

6.7 Creating a Pop-Up Menu Widget

8 Create the pop-up menu widget.

Use any of the widget creation mechanisms listed in Table 6–11. The choice of mechanism depends on the attributes of the pop-up menu widget you need to access.

Table 6–11 Pop-Up Menu Widget Creation Mechanisms

High-level routine	Use the MENU routine to create a pop-up menu widget. Specify the type in the format argument.
Low-level routine	Use the MENU POPUP CREATE routine to create a pop-up menu widget.
UIL object type	Use the popup_menu object type to define a pop-up menu widget in a UIL module. At run time, the DRM routine FETCH WIDGET will create the widget according to this definition.

9 Create the children of the pop-up menu widget.

Use any of the widget creation mechanisms to create the widgets that you want to appear as items in the pop-up menu widget. The widgets are children of the pop-up menu widget. The children appear in the menu in the order that you create them.

10 Manage the children of the pop-up menu widget.

Use the intrinsic routine MANAGE CHILD to manage a single child of the pop-up menu widget. Use the intrinsic routine MANAGE CHILDREN to manage more than one child of the pop-up menu widget in a single call.

When a user running the application moves the pointer cursor into the parent widget of the pop-up menu widget and presses MB2, the pop-up menu widget will appear on the display. Note that you manage the pop-up menu widget in the action procedure you create.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available using a high-level routine, use the SET VALUES intrinsic routine after the widget has been created.) Table 6–12 lists the attributes you can set if you use the high-level routine MENU to create a pop-up menu widget. Pass the values of these attributes as arguments to the routine.

Table 6–12 Attributes Accessible Using the High-Level Routine MENU

x	Specifies the x-coordinate of the upper left corner of the widget.
y	Specifies the y-coordinate of the upper left corner of the widget.

(continued on next page)

Creating Menu Widgets

6.7 Creating a Pop-Up Menu Widget

Table 6-12 (Cont.) Attributes Accessible Using the High-Level Routine MENU

format	Specifies the type of menu: pull-down, pop-up, or work area.
orientation	Specifies whether the menu has a horizontal or vertical orientation.
map_callback	Specifies the address of a callback routine list.
entry_callback	Specifies the address of a callback routine list.
help_callback	Specifies the address of a callback routine list.

6.7.1 Creating an Action Procedure

An action procedure is a procedure that is executed when a particular event occurs in a widget. All widgets contain action procedures. For example, when a user activates a push button widget by moving the pointer cursor into the push button and pressing MB1, the push button widget executes an action procedure.

To cause a pop-up menu widget to pop up on a display, you must create an action procedure and add it to the set of action procedures known by the parent of the pop-up menu widget. The action procedure you write must perform the following two functions:

- Positioning the pop-up menu widget where the user of the application has moved the pointer cursor
- Managing the pop-up menu widget

An action procedure must have the following four arguments:

- Widget in which the event occurred
- Event that occurred
- Parameters used by the action procedure
- Number of parameters passed to the action procedure

To position the pop-up menu widget where the user has placed the pointer cursor, use the MENU POSITION routine. This routine takes the following two arguments:

- The identifier of the menu widget to be positioned
- A pointer to the event data structure returned by the widget

The event data structure contains information on where the event occurred on the display. If you do not explicitly position the pop-up menu widget, it appears in the upper left corner of the display. Note that the pop-up menu widget must be realized before you call the MENU POSITION routine.

Use the intrinsic routine MANAGE CHILD to manage the pop-up menu widget. Since its parent has already been realized, the pop-up menu widget will appear on the display. Pop-up menu widgets disappear from the display automatically when the user of the application releases MB2.

Creating Menu Widgets

6.7 Creating a Pop-Up Menu Widget

Example 6-5 is a sample action procedure that pops up a pop-up menu widget.

Example 6-5 Action Procedure to Pop Up a Pop-Up Menu Widget

```
static void pop_up( widget, event, params, num_params )
    Widget    widget;
    XButtonPressedEvent *event;
    char      **params;
    int       num_params;
{
    if( !XtIsRealized( popup_menu )
        XtRealizeWidget( popup_menu );

    DwtMenuPosition( popup_menu, event );

    XtManageChild( popup_menu );
}
```

6.7.2 Adding an Action Procedure to a Widget

To add a new action procedure to a widget, you must add entries for the action procedure to the translation table and the action table of the widget. The widgets in the XUI Toolkit map an event to the name of an action procedure in their translation table and map the name of the action procedure to the address of the action procedure in their action table.

A translation table is a text string containing a list of translations. The translations are separated from each other by the newline character (\n). Each translation pairs an event identifier, terminated by a colon, with the name of an action procedure. Following is a sample translation table entry that associates the MB2 press event with the action procedure illustrated in Example 6-5. (See Section D.9.4 for more information about creating translation table entries.)

```
static char popup_translation_table[] = "<Btn2Down>: pop_up()";
```

An action table is an array of data structures that pair action procedure names with their addresses. The XUI Toolkit defines this data structure (XtActionRec). Following is a sample action table entry that associates the name of the action procedure illustrated in Example 6-5 with its address:

```
static XtActionRec our_action_table[] =
{
    { "pop_up", (caddr_t)pop_up }
}
```

Before you can add the new translation table entry to a widget, you must convert the entry from its ASCII format to the binary format used by the XUI Toolkit. Use the intrinsic routine PARSE TRANSLATION TABLE to perform this step. This routine returns the parsed translation table defined as the data type XtTranslations.

Creating Menu Widgets

6.7 Creating a Pop-Up Menu Widget

After converting the translation table entry, you can add the new translation to the existing translation table entry by using the intrinsic routine `OVERWRITE TRANSLATIONS`. This routine adds the new translation to the translation table of the widget without destroying the other translations in the table.

To add the new action table entry to an existing action table, use the intrinsic routine `ADD ACTIONS`.

The sample program in Example 6-6 creates a pop-up menu widget.

Example 6-6 Creating a Pop-Up Menu Widget

```
#include <stdio>
#include <decw$include/DwtAppl.h>

static Widget toplevel, main_widget, popup_menu, label;
static WidgetList menu_items[5];

❶ static void pop_up();

❷ static char popup_translation_table[] = "<Btn2Down>: pop_up()";

❸ static XtActionsRec our_action_table[] =
{
    {"pop_up", (XtActionProc)pop_up},
};

int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Arg arglist[5];

    /***** Set up the User Interface *****/
    toplevel = XtInitialize("Popup Demo","demo",NULL, 0, &argc, argv);
    XtSetArg (arglist[0], XtNallowShellResize, TRUE);
    XtSetValues (toplevel, arglist, 1);

    XtSetArg (arglist[0], DwtNwidth, 300 );
    XtSetArg (arglist[1], DwtNheight, 300 );

    ❹ main_widget = DwtDialogBoxCreate( toplevel, "MAINWIN", arglist, 4 );
    XtSetArg (arglist[0], DwtNmarginLeft, 75 );
    XtSetArg (arglist[1], DwtNlabel,
        DwtLatin1String("Move the pointer\nanywhere in this box\nand press MB2") );

    ❺ label = DwtLabelCreate( main_widget, "label", arglist, 2 );

    ❻ XtManageChild( label );
    XtManageChild( main_widget );
```

(continued on next page)

Creating Menu Widgets

6.7 Creating a Pop-Up Menu Widget

Example 6-6 (Cont.) Creating a Pop-Up Menu Widget

```
⑦ handle_mb2_press( main_widget );
⑧ XtRealizeWidget( toplevel );
⑨ build_popup_menu();
/***** Main Input Loop *****/
    XtMainLoop();
    return (0);
}
⑩ static void pop_up( widget, event, params, num_params )
    Widget widget;
    XButtonPressedEvent *event;
    char **params;
    int num_params;
{
    if( !XtIsRealized( popup_menu )
        XtRealizeWidget( popup_menu );
    DwtMenuPosition( popup_menu, event );
    XtManageChild( popup_menu );
}
⑪ handle_mb2_press( widget )
    Widget widget;
{
    Arg arglist[2];
    XtTranslations parsed_t_table;
    XtAddActions( our_action_table, 1 );
    parsed_t_table = XtParseTranslationTable( popup_translation_table );
    XtOverrideTranslations( widget, parsed_t_table );
}
⑫ build_popup_menu()
{
    Arg arglist[4];
    int count = 0;

    popup_menu = DwtMenuPopupCreate( main_widget, "button", NULL, 0);
    XtSetArg( arglist[0], DwtNlabel, DwtLatin1String("Menu Item A ") );
    menu_items[count++] = DwtPushButtonCreate( popup_menu, "button1", arglist, 1);
    XtSetArg( arglist[0], DwtNlabel, DwtLatin1String("Menu Item B ") );
    menu_items[count++] = DwtPushButtonCreate( popup_menu, "button2", arglist, 1);
    XtManageChildren( menu_items, count );
}
```

- ① This is a forward declaration of the action procedure that pops up the pop-up menu widget under the pointer cursor. ⑩ defines this action procedure.

Creating Menu Widgets

6.7 Creating a Pop-Up Menu Widget

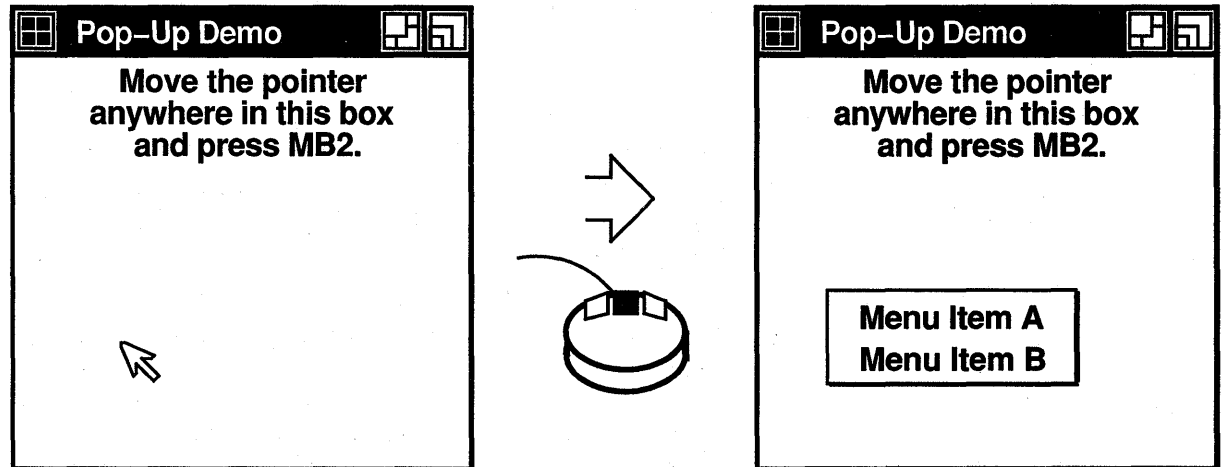
- ② This statement creates an entry in a translation table, *popup_translation_table*, that associates the event of a MB2 press with the name of the pop-up action procedure, *pop_up*.
- ③ In this statement, the sample program creates an entry in an action table, *our_action_table*, that associates the name of the action procedure with the address of the procedure. The sample program creates an action table entry for its pop-up action procedure.
- ④ The sample program creates a dialog box widget as the base of the application widget hierarchy by calling the low-level routine `DIALOG BOX CREATE`. This dialog box has as its children a label widget and the pop-up menu widget.
- ⑤ This statement creates the label widget with a call to the low-level routine `LABEL CREATE`. This label widget puts the instructional message "Move the pointer anywhere in this box and press MB2" inside the dialog box widget. By setting the left margin of the label widget, the label appears centered within the dialog box widget.
- ⑥ In these two calls to `MANAGE CHILD`, the sample program manages the label and the dialog box widget. Note that the other child of the main widget, the pop-up menu widget, does not get created or managed at this point in the application. This is accomplished after the dialog box widget has been realized.
- ⑦ The procedure *handle_mb2_press* adds the pop-up procedure to the action table of the dialog box widget. ⑩ details this routine.
- ⑧ In this call to the intrinsic routine `REALIZE WIDGET`, the sample program causes the dialog box widget and its managed child to appear on the display.
- ⑨ After realizing the widgets in the application widget hierarchy, the sample program calls the procedure that builds the pop-up menu widget. ⑫ describes this procedure.
- ⑩ This is the application-written routine that causes the pop-up menu widget to appear on the display. The routine pops up the pop-up menu widget wherever the user has positioned the pointer cursor within its parent. Section 6.7.1 describes this routine.
- ⑪ The *handle_mb2_press* routine performs all the processing necessary to add the pop-up routine to the action table of a widget so that it can manage the pop-up menu widget when a user presses MB2. Section 6.7.2 describes this procedure.
- ⑫ This routine creates the pop-up menu widget and its children. The child widgets will be items in the pop-up menu widget. Note that you only manage the children of the pop-up menu widget. The pop-up menu widget is managed in the action routine invoked when a user presses MB2 in the parent widget.

Figure 6-9 illustrates how the pop-up menu created in Example 6-6 appears on the display. The figure shows how a user can position the pointer cursor within the borders of the pop-up menu widget's parent and, by pressing MB2, make the pop-up menu widget appear on the display over the pointer cursor position.

Creating Menu Widgets

6.7 Creating a Pop-Up Menu Widget

Figure 6-9 Pop-Up Menu Widget



ZK-0203A-GE

6.7.3 Customizing a Pop-Up Menu Widget

The pop-up menu widget supports the same set of attributes as the work area menu widget. For information about customizing a work area menu widget, see Section 6.3.1.

6.7.4 Associating Callback Routines with a Pop-Up Menu Widget

The pop-up menu widget supports the same callbacks as the work area menu widget. For information about these callbacks, see Section 6.3.1.

In addition, with the pop-up menu widget, you can associate callback routines that get executed when the pop-up menu widget is about to appear on the display (be mapped) or has disappeared from the display (been unmapped). This notification enables an application to perform processing or perform some other action before the pop-up menu widget appears on the display or disappears from the display. To associate a callback routine with these callbacks, pass a callback list to the **map_callback** and **unmap_callback** attributes.

For example, you could write a map callback routine that creates the children of the pop-up menu widget only when the pop-up menu widget is about to be mapped. In this way, you perform this processing only when necessary, saving on application startup time.

7

Creating Dialog Box Widgets

This chapter provides the following:

- An overview of the dialog box widgets in the XUI Toolkit
- A detailed description of how to include a dialog box widget in your application

7.1 Overview of the Dialog Box Widget

A **dialog box widget** is a rectangular container for other widgets. You use a dialog box widget to solicit information from, and present messages to, users of your application. A dialog box widget can accept input focus to allow users to perform input using the keyboard.

The Hello World! sample application and the DECburger sample application both provide examples of dialog box widgets used as containers. The Hello World! application uses a dialog box widget to contain the label widget and the push button widget that implement the application function. The DECburger order entry box is a dialog box widget that contains dozens of widgets.

7.2 Dialog Box Widgets in the XUI Toolkit

A dialog box widget is a composite widget and, as such, its primary function is to act as a container for other widgets. The XUI Toolkit provides several dialog box widgets that fall into two general categories:

- Generic dialog box widgets
- Standard dialog box widgets

The generic dialog box widgets are simply empty rectangles. You decide what widgets they will contain to suit the needs of your application. The standard dialog box widgets are preconfigured with child widgets to perform certain commonly needed functions.

7.2.1 Generic Dialog Box Widgets

The XUI Toolkit provides two generic dialog box widgets:

- Dialog box widget
- Attached dialog box widget

Both widgets provide the same functional capabilities; they differ in how you specify the layout of their child widgets and how this positioning is maintained.

Creating Dialog Box Widgets

7.2 Dialog Box Widgets in the XUI Toolkit

7.2.1.1 Dialog Box Widget

In a dialog box widget, you create the layout of the child widgets by positioning each child within the dialog box widget by its x- and y-coordinates. This creates a fixed layout. If the dialog box widget is resized by a user or by the request of one of its children, the borders of the dialog box widget change, but the position of each child widget remains fixed. If the dialog box widget is made smaller, a child widget can be partially or completely clipped by the new boundaries of the parent.

In addition, the fixed layout of a dialog box widget creates font and language dependences. If you make the font larger or smaller, you risk upsetting the layout of the dialog box widget. Similarly, the text of a label may be significantly longer or shorter in different languages. In these cases, the child widgets may overlap, or labels may be clipped.

The Hello World! sample application provides an example of this behavior. In the Hello World! application, a push button widget containing the text string "Hello World!" appears centered in the dialog box when the user interface initially appears. When the user activates the push button widget, the text in the push button widget changes to "Good-bye World!". Because this text string is longer, the push button widget resizes to accommodate the new text. After being resized, the push button no longer appears centered in the dialog box widget. For this reason, when the Hello World! application changes the text, it also assigns a new value to the x-coordinate of the push button widget so that it will remain centered in the dialog box widget.

7.2.1.2 Attached Dialog Box Widget

To eliminate the limitations of the dialog box widget, the XUI Toolkit includes the attached dialog box widget. In an attached dialog box widget, you design the initial layout as you would with a normal dialog box widget. However, instead of defining the fixed position of each child widget by its x- and y-coordinates, you specify the position of the child widgets in relation to other child widgets or in relation to the attached dialog box widget.

You specify the position of a child widget in an attached dialog box widget by defining **attachments** between the child widget and its surroundings. An attachment is a special widget attribute you can use with any XUI Toolkit widget. The attachment defines the relationship between an edge of a child widget to an edge of the attached dialog box, to another child widget in the attached dialog box widget, or to a position within the attached dialog box widget. When the attached dialog box widget is resized, the child widgets can grow or shrink to maintain their original layout in the attached dialog box and to avoid being clipped. Section 7.5.1 describes how to define attachments.

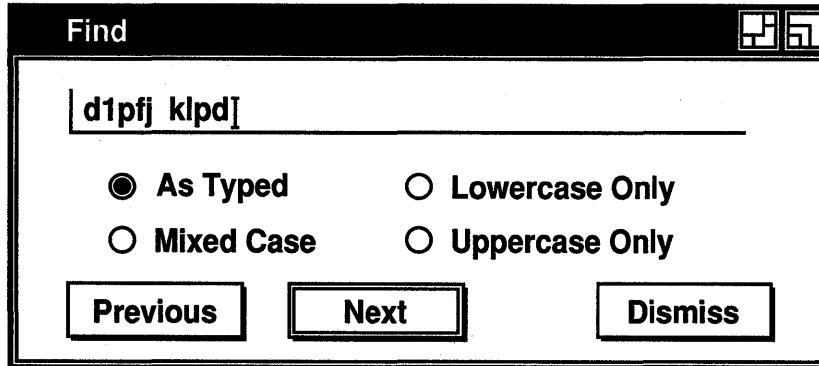
Figure 7-1 illustrates the resizing behavior of a dialog box widget and an attached dialog box widget. The resizing behavior of the child widgets depends on the types of attachments defined and their individual resizing characteristics.

Creating Dialog Box Widgets

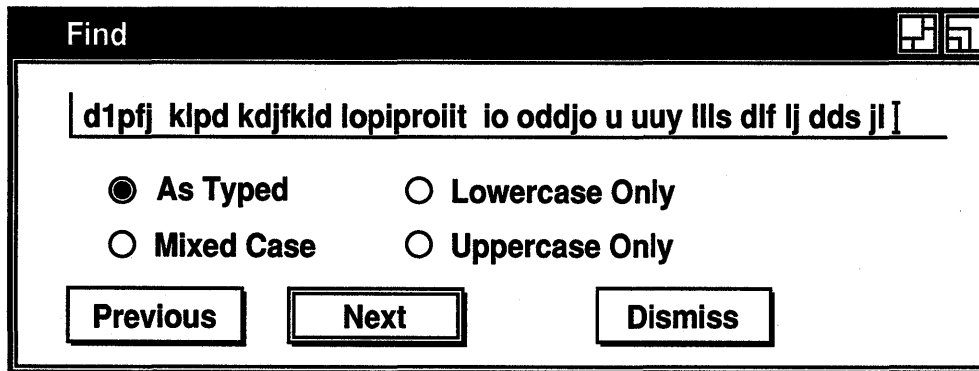
7.2 Dialog Box Widgets in the XUI Toolkit

Figure 7-1 Resizing a Dialog Box Widget

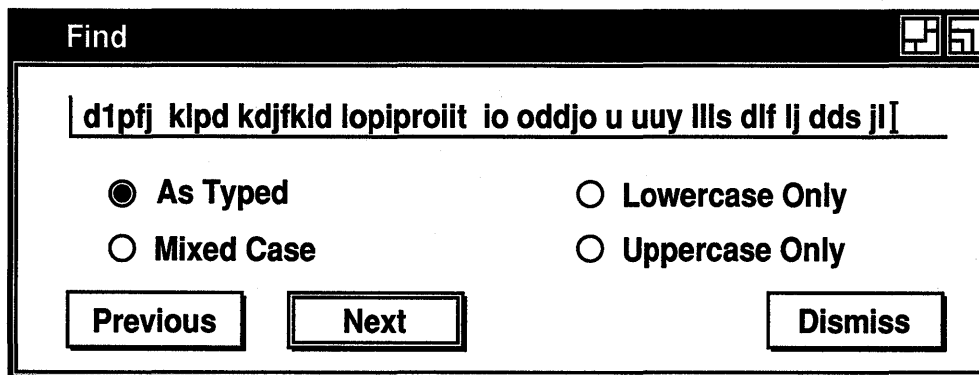
Initial Dialog Box



Resized Dialog Box



Resized Attached Dialog Box



ZK-0406A-GE

Creating Dialog Box Widgets

7.2 Dialog Box Widgets in the XUI Toolkit

7.2.2 Standard Dialog Box Widgets

The XUI Toolkit provides a set of standard dialog box widgets that perform commonly needed functions, such as presenting messages or selections. The standard dialog box widgets are preconfigured to contain the child widgets they need to implement their particular function. You do not have to build these dialog box widgets out of their component widgets.

Following are the standard dialog box widgets. The widgets listed under each of the two main standard dialog box widgets are variations of the standard dialog box widgets that perform specialized functions.

- Message box widget
 - Caution box widget
 - Work-in-progress box widget
- Selection box widget
 - File selection widget

7.2.2.1 Message Box Widget

A message box widget is a dialog box widget that contains a label widget and can optionally contain a push button widget. You specify the text of your message in the label widget. The push button widget allows the user to acknowledge the message. Use the message box widget to present any application-specific information to the user.

The XUI Toolkit provides two other versions of the message box widget that you can use for specific types of messages. Use the caution box widget to present a warning message to the user of your application. Use the work-in-progress box widget to notify the user of your application that processing is in progress.

7.2.2.2 Selection Box Widget

The selection box widget is a dialog box widget that contains a list box widget and a simple text widget, and can optionally contain several push button widgets. The list box presents the items of the selection.

The XUI Toolkit also provides a version of the selection widget, called the file selection widget, that is designed to be used with directories of files.

7.3 Styles of Dialog Box Widgets

The XUI Toolkit supports three styles of dialog box widgets:

- Work area
- Modal
- Modeless

Work area dialog box widgets are clipped by their parents. The dialog box widget in the Hello World! application is an example of a work area style dialog box widget.

Creating Dialog Box Widgets

7.3 Styles of Dialog Box Widgets

Modal and modeless dialog box widgets are pop-up widgets; they are not clipped by their parents. A modal dialog box widget causes an application to suspend all other processing until the user responds to the query presented by the dialog box widget. Modal dialog box widgets do not support resizing. Modeless dialog box widgets have title bars and can optionally be moved and resized by a user.

The dialog box widget in the DECburger sample application is an example of a modeless pop-up dialog box widget. All standard dialog box widgets are pop-up dialog box widgets.

7.4 Creating a Dialog Box Widget

The dialog box widget is a container for its child widgets. The child widgets may themselves have children. For example, a dialog box widget can contain a menu widget, which can contain many push button widgets.

To create a dialog box widget, perform the following steps:

1 Create the dialog box widget.

Use any of the widget creation mechanisms listed in Table 7-1. The choice of creation mechanism depends on the attributes of the dialog box you need to access to customize the dialog box widget.

Table 7-1 Dialog Box Widget Creation Mechanisms

High-level routine	Use the DIALOG BOX routine to create any style of dialog box widget. Indicate the style of the dialog box widget in the style argument.
Low-level routine	Use the DIALOG BOX CREATE routine to create a work area dialog box widget. To create a modal or modeless dialog box widget, use the DIALOG BOX POPUP CREATE routine. Indicate the style of the dialog box widget in the style attribute.
UIL object type	Use the UIL object type dialog_box to define a work area dialog box widget in a UIL module. To define a modal or modeless dialog box widget, use the UIL object type popup_dialog_box. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

2 Create the children of the dialog box widget.

Use any of the widget creation mechanisms to create the widgets you want to appear inside the dialog box widget. In this step, position the child widgets within the dialog box widget.

3 Manage the children of the dialog box widget.

Use the intrinsic routine MANAGE CHILD to manage a single child widget; use the MANAGE CHILDREN routine to manage a group of child widgets.

4 Manage the dialog box widget.

Use the intrinsic routine MANAGE CHILD to manage a single child widget.

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

After you complete these steps, if the parent of the dialog box widget has been realized, the dialog box widget and all its managed children will appear on the display.

Low-level widget creation routines and UIL provide access to the complete set of attributes at creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To assign values to those widget attributes not accessible, you must use the SET VALUES intrinsic routine after the widget has been created.) Table 7-2 lists the attributes you can set if you use the high-level routine DIALOG BOX to create a dialog box widget. Pass the values of these attributes as arguments to the routine.

Table 7-2 Attributes Accessible Using the High-Level Routine DIALOG BOX

default_position	A Boolean value that determines whether the x- and y-coordinates should be ignored in favor of default positioning.
x	Specifies the x-coordinate of the upper left corner of the widget.
y	Specifies the y-coordinate of the upper left corner of the widget.
title	Specifies the title displayed in the title bar (modeless dialog boxes only).
style	Specifies the style of dialog box: modal, modeless, or work area.
map_callback	Specifies the address of callback routine list.
help_callback	Specifies the address of callback routine list.

7.4.1 Specifying the Layout of Children in a Dialog Box Widget

To position the child widgets within a dialog box widget, specify the x- and y-coordinates for each child widget in their common widget attributes **x** and **y**. The origin of the coordinate system is the upper left corner of the dialog box widget. If you specify margins, the origin of the coordinate system is offset by the amount of the margin.

The *XUI Style Guide* provides recommendations for the aesthetic arrangement of child widgets in a dialog box widget.

As an example, the DECburger sample application creates its order entry box as a dialog box widget. The DECburger dialog box widget contains work area menu widgets, label widgets, separator widgets, a radio box widget, a scale widget, an option menu widget, a simple text widget, and a list box widget. Some of these child widgets (for example, the menu widgets) have child widgets of their own. The following sections detail the attributes of the dialog box widget using the DECburger order entry box as an example. Example 7-1 presents the section of the DECburger UIL module in which the dialog box widget is defined.

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

Example 7-1 Creating the Dialog Box Widget in the DECburger Application

① object

```
control_box : popup_dialog_box {
  arguments {
    ② title = k_decburger_title;
      style = DwtModeless;
      x = 300;
      y = 100;
      margin_width = 20;
      background_color = lightblue;
  };
  ③ controls {
    label        burger_label;
    label        fries_label;
    label        drink_label;

    separator    {arguments {
                  x = 110;
                  y = 10;
                  orientation = DwtOrientationVertical;
                  height = 180; }};

    separator    {arguments {
                  x = 205;
                  y = 10;
                  orientation = DwtOrientationVertical;
                  height = 180; }};

    work_area_menu  button_box;
    radio_box       burger_doneness_box;
    work_area_menu  burger_toppings_menu;
    scale           burger_quantity;
    option_menu     fries_option_menu;
    label           fries_quantity_label;
    simple_text     fries_quantity;
    list_box        drink_list_box;
    attached_dialog_box  drink_quantity;
  };
  callbacks {
    create = procedure create_proc (k_order_box);
  };
};
```

-
- ① In this UIL object declaration, DECburger defines the dialog box widget used to implement its order entry box. DECburger uses a modeless dialog box widget, so, in UIL, it must use the `popup_dialog_box` object type identifier.

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

- ② In the argument list, DECBurger configures the dialog box widget. In these attributes, DECBurger defines the text string that will appear in the title bar, defines the style of the dialog box widget as modeless, and positions the dialog box in relation to its parent using its x- and y-coordinates. The last two attributes determine the width of the border and the color of the dialog box widget.
- ③ In the controls section of the UIL object declaration, DECBurger defines the widgets that will be the children of the dialog box widget. These widgets implement the selections contained in the DECBurger order entry box.

After creating the dialog box widget, DECBurger creates all the widgets that will be children of the dialog box widget. DECBurger determines the layout of the dialog box widget by specifying the position of each child using the *x* and *y* attributes. If DECBurger did not specify the position, by default all the widgets would appear in the upper left corner of the dialog box widget overlapping each other. Example 7-2 presents a portion of the DECBurger UIL module in which the widgets in the Hamburgers section of the order entry box are defined.

Example 7-2 Creating the Children of the Dialog Box Widget in the DECBurger Application

```
.
.
.
①object
  burger_label : label {
    arguments {
      x = 25;
      y = 5;
      label_label = k_hamburgers_label_text;
    };
    callbacks {
      create = procedure create_proc (k_burger_label);
    };
  };
②object
  burger_doneness_box : radio_box {
    arguments {
      x = 10;
      y = 22;
      orientation = DwtOrientationVertical;
      border_width = 0;
    };
    controls {
      toggle_button    burger_rare;
      toggle_button    burger_medium;
      toggle_button    burger_well;
    };
  };
```

(continued on next page)

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

Example 7-2 (Cont.) Creating the Children of the Dialog Box Widget in the DECburger Application

```
.
.
.
③object
  burger_toppings_menu : work_area_menu {
    arguments {
      x = 55;
      y = 22;
      orientation = DwtOrientationVertical;
      border_width = 0;
    };
    controls {
      toggle_button    ketchup;
      toggle_button    mustard;
      toggle_button    pickle;
      toggle_button    onion;
      toggle_button    mayo;
    };
  };
.
.
.
```

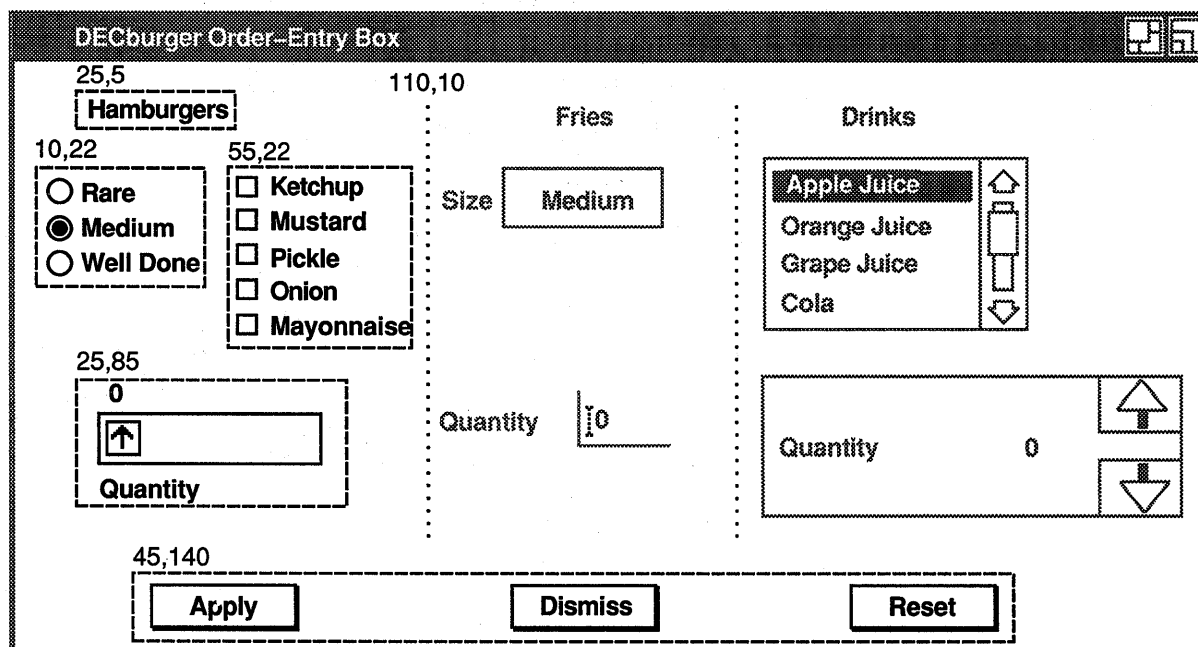
-
- ① In this object declaration, DECburger defines the label gadget that contains the text string Hamburgers. Note how DECburger specifies the position of the gadget within the dialog box widget by assigning values to the **x** and **y** attributes.
 - ② DECburger defines the radio box widget child of the dialog box widget. Once again, DECburger specifies the position of the widget using **x** and **y** attributes. (The object declarations of the children of the radio box widget are left out of this example because they are not children of the dialog box widget.)
 - ③ DECburger uses a work area menu widget to present the choice of toppings. DECburger positions this widget by assigning values to the **x** and **y** attributes.

Figure 7-2 shows how this layout appears in the DECburger user interface. Note that in the example, the borders of the widgets are included to show the upper left corners of the widgets with their associated **x**- and **y**-coordinates.

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

Figure 7-2 Layout of the DECburger Dialog Box Widget



ZK-0407A-GE

7.4.2 Customizing the Dialog Box Widget

The attributes of the dialog box widget enable you to customize the following aspects of its appearance and functioning:

- Initial size and resizing behavior
- Initial position
- Unit of measure used in the dialog box widget
- Translations used by simple text widgets in a dialog box widget
- Accelerators used by push button widgets in a dialog box widget
- Management of input focus

7.4.2.1 Sizing and Resizing a Dialog Box Widget

Use the common widget attributes **width** and **height** to size a dialog box widget. You specify these dimensions in pixels. By default, a dialog box widget sizes itself to fit all its child widgets.

When calculating its size, the dialog box widget includes margin settings. Use the **margin_width** attribute to specify a horizontal margin in pixels. Use the **margin_height** attribute to specify a vertical margin in pixels. You can only set the bottom and the left margins in a dialog box widget.

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

By default, a dialog box widget grows to accommodate additional children or the growth of an existing child widget, but you can control this behavior using the **resize** attribute. The XUI Toolkit defines three types of resizing behavior:

- Grow to accommodate child widgets (the default)
- Shrink when a child widget reduces its size
- Remain initial size

The *VMS DECwindows Toolkit Routines Reference Manual* lists the constants used to specify this resizing behavior.

A modeless dialog box widget can optionally contain a resize icon in its title bar. Only modeless dialog box widgets contain title bars.

When a dialog box widget resizes, its child widgets may overlap. You can control this behavior by using the **child_overlap** attribute. If you do not want child widgets to overlap, set the **child_overlap** attribute to false.

7.4.2.2 Positioning a Dialog Box Widget

To position a dialog box widget in relation to its parent, specify its x- and y-coordinates in the common widget attributes **x** and **y**. If you set the **default_position** true, the parent widget ignores any x- and y-coordinate values and centers the dialog box widget within the boundaries of its parent.

Some parent widgets determine the position of a dialog box widget child. For example, a main window widget positions a dialog box widget child below its title bar.

By default, modal and modeless dialog box widgets (both pop-up widgets) appear centered in relation to their parent. Because modeless dialog box widgets allow users to move them on the display, any positioning specified only determines the initial position of the widget.

7.4.2.3 Selecting the Unit of Measure Used in a Dialog Box Widget

The unit of measure in a dialog box widget is either the pixel or the font unit. For horizontal dimensions, a font unit is one-fourth the width of the default font. For vertical dimensions, a font unit is one-eighth the height of the font. Specify the unit of measure in the **units** attribute. By default, dialog box widgets use the font unit.

Any dialog box widget that displays text should use the font unit of measure. In this way, dimensions specified will remain valid if the font changes.

7.4.2.4 Defining Translations for Simple Text Widgets

Dialog box widgets provide special translation capabilities for the simple text widgets they contain. You can define keyboard translations in a translation table, parse the translation table, and then pass the parsed translation table to the dialog box widget. Section 6.7.2 details how to create a translation table and parse it. The **text_merge_translations** attribute accepts the identifier of the parsed translation table as its value and calls the intrinsic routine `OVERRIDE TRANSLATIONS` on all its children that are simple text widgets.

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

7.4.2.5 Assigning Accelerators to Child Widgets

Accelerators allow users of an application to activate a function associated with a push button or toggle button widget using a keyboard key. Dialog box widgets contain built-in accelerator key definitions that you can associate with the push button widgets contained in the dialog box widget. To make use of an accelerator, you need only pass the widget identifier of the push button widget to the dialog box widget in the **default_button** attribute.

To associate an accelerator with the Cancel button in a dialog box widget, pass the widget identifier of the push button widget implementing the cancel option to the dialog box widget in the **cancel_button** attribute.

7.4.2.6 Grabbing the Input Focus

By default, modal dialog box widgets take the input focus when they appear on the display because the rest of the application is disabled until the user responds to the modal dialog box widget. By default, modeless dialog box widgets do not take the input focus when they appear on the display. To change the default for either style dialog box widget, use the **take_focus** attribute.

The **auto_unmanage** attribute determines whether the dialog box widget automatically disappears when the user activates any of the push button widgets contained in the dialog box widget. If this attribute is set to false, you must explicitly remove the dialog box widget from its parent's list of managed children.

7.4.3 Associating Callback Routines with a Dialog Box Widget

The child widgets contained in a dialog box widget provide the primary input capabilities of dialog box widgets. For example, in a dialog box widget containing push button widgets, the push button widgets perform callbacks when activated by a user. However, the dialog box widget does support several callbacks with which you can associate callback routines.

To associate a callback routine with a dialog box widget that gets executed when the dialog box accepts the input focus, pass a callback routine list in the **focus_callback** attribute. Dialog box widgets take the input focus when a user clicks MB1 in an inactive area of the dialog box widget.

Pop-up dialog box widgets also execute a callback when they are about to appear on a display (be mapped) or have just disappeared from the display (been unmapped). You can use these callbacks to defer processing and reduce application startup time. For example, your application could delay creation of the children of the dialog box widget until it receives notification that the dialog box widget is about to be mapped.

To associate a callback routine with these callback reasons, pass a callback list in the **map_callback** and **unmap_callback** attributes.

Another callback supported by the dialog box widget is the **help_callback**. The dialog box widget executes this callback when a user presses the help key while simultaneously positioning the pointer cursor in an inactive area of the dialog box widget and pressing MB1. An inactive area of a

Creating Dialog Box Widgets

7.4 Creating a Dialog Box Widget

dialog box widget is the space surrounding all the children of the dialog box widget.

7.5 Creating an Attached Dialog Box Widget

Like a dialog box widget, an attached dialog box widget is a container for its children. The main difference between the two is how you specify the positioning of the child widgets.

To create an attached dialog box widget, perform the following steps:

1 Create the attached dialog box widget.

Use any of the widget creation mechanisms listed in Table 7-3. The choice of creation mechanism depends on the style of attached dialog box widget and the attributes you want to initialize to other than default values.

Table 7-3 Attached Dialog Box Widget Creation Mechanisms

High-level routine	Use the ATTACHED DIALOG BOX routine to create any style of attached dialog box widget. Indicate the style of the attached dialog box widget in the style argument.
Low-level routine	Use the ATTACHED DIALOG BOX CREATE routine to create a work area attached dialog box widget. To create a modal or modeless attached dialog box widget, use the ATTACHED DIALOG BOX POPUP CREATE routine. Indicate the style of the attached dialog box widget in the style attribute.
UIL object type	Use the UIL object type <code>attached_dialog_box</code> to define a work area style attached dialog box widget in a UIL module. To define a modal or modeless attached dialog box widget, use the UIL object type <code>popup_attached_db</code> . At run time, the DRM routine <code>FETCH WIDGET</code> creates the widget according to this definition.

2 Create the children of the attached dialog box widget.

In this step, position the child widgets within the attached dialog box widget by defining attachments. Section 7.5.1 describes how to define attachments.

3 Manage the children of the attached dialog box widget.

Use the intrinsic routine `MANAGE CHILD` to manage a single child. Use the intrinsic routine `MANAGE CHILDREN` to manage a group of children.

4 Manage the attached dialog box widget.

Use the intrinsic routine `MANAGE CHILD` to manage the attached dialog box widget.

After you complete these steps, if the parent of the attached dialog box widget has been realized, the attached dialog box widget and all its children will appear on the display.

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To assign values to those widget attributes not accessible, you must use the SET VALUES intrinsic routine after the widget has been created.) Table 7-4 lists the attributes you can set if you use the high-level routine ATTACHED DIALOG BOX to create an attached dialog box widget. Pass the values of these attributes as arguments to the routine.

Table 7-4 Attributes Accessible Using the High-Level Routine ATTACHED DIALOG BOX

default_position	A Boolean value that determines whether the x- and y-coordinates should be ignored in favor of default positioning.
x	Specifies the x-coordinate of the upper left corner of the widget.
y	Specifies the y-coordinate of the upper left corner of the widget.
title	Specifies the title displayed in the title bar (modeless dialog boxes only).
style	Specifies the style of dialog box: modal, modeless, or work area.
map_callback	Specifies the address of a callback routine list.
help_callback	Specifies the address of a callback routine list.

7.5.1 Defining Attachments in an Attached Dialog Box Widget

Position child widgets in an attached dialog box widget by defining attachments for any of the four edges of the child widget. Using the attachment attributes, you can attach an edge of the child widget to any of the following:

- An edge of the attached dialog box widget
- An edge of another child of the attached dialog box widget
- A position within the attached dialog box widget

Define attachments by assigning values to attachment attributes and passing these values to the children of the attached dialog box widget. Note that you define attachments in the child widgets, not in the attached dialog box widget that contains them.

The attachment attributes may alter the widget attributes that define position and size: the **x**, **y**, **width**, and **height** attributes. If you define an attachment attribute as well as one of these other sizing and positioning attributes and the values conflict, the attachment attribute overrides the normal widget attribute.

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

The XUI Toolkit defines four attachment attributes for each of the four edges of a child widget. Table 7–5 shows the set of attachment attributes used to specify the attachment of each edge.

Table 7–5 Attachment Attributes

Edge	Attachment Attribute	Function
top	adb_top_attachment	Type of attachment (see Table 7–6)
	adb_top_widget	Identifier of sibling widget to which edge is being attached
	adb_top_position	Numerator of fraction used in fractional positioning
	adb_top_offset	Amount of space between attached edges
bottom	adb_bottom_attachment	Type of attachment (see Table 7–6)
	adb_bottom_widget	Identifier of sibling widget to which edge is being attached
	adb_bottom_position	Numerator of fraction used in fractional positioning
	adb_bottom_offset	Amount of space between attached edges
right	adb_right_attachment	Type of attachment (see Table 7–6)
	adb_right_widget	Identifier of sibling widget to which edge is being attached
	adb_right_position	Numerator of fraction used in fractional positioning
	adb_right_offset	Amount of space between attached edges
left	adb_left_attachment	Type of attachment (see Table 7–6)
	adb_left_widget	Identifier of sibling widget to which edge is being attached
	adb_left_position	Numerator of fraction used in fractional positioning
	adb_left_offset	Amount of space between attached edges

Notice in Table 7–5 that each edge has the same four attributes. You need not always specify all four attributes for each edge; this depends on what type of attachment you define. The XUI Toolkit defines seven types of attachment. Table 7–6 lists the types of attachments with the constants you use to specify them.

Table 7–6 Attachment Constants for the Attached Dialog Box Widget

Attachment Type Constant	Function
DwtAttachAdb	Attach this edge to the same edge of the attached dialog box (the parent)
DwtAttachOppAdb	Attach this edge to the opposite edge of the attached dialog box (the parent)
DwtAttachWidget	Attach this edge to the opposite edge of the sibling
DwtAttachOppWidget	Attach this edge to the same edge of the sibling

(continued on next page)

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

Table 7–6 (Cont.) Attachment Constants for the Attached Dialog Box Widget

Attachment Type Constant	Function
DwtAttachPosition	Attach this edge to a point within the attached dialog box
DwtAttachSelf	Attach this edge to its current position in the attached dialog box
DwtAttachNone	Do not attach this edge

The following sections describe how you use the attachment attributes to create these types of attachments.

7.5.1.1 Attaching an Edge to the Attached Dialog Box

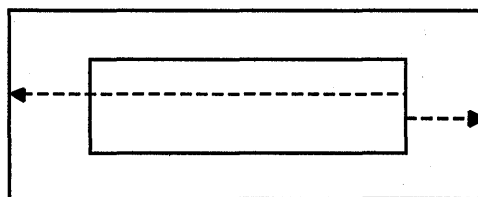
You can attach an edge of a child widget to either of the following two edges of the attached dialog box widget that is its parent:

- The same edge of the attached dialog box widget (**DwtAttachAdb**)
- The opposite edge of the attached dialog box widget (**DwtAttachOppAdb**)

Specify these types of attachment by passing the attachment type constant as the value of the attachment type attribute.

For example, set the **adb_right_attachment** attribute to **DwtAttachAdb** to attach the right edge of the child widget to the right edge of the attached dialog box widget. To attach the right edge of the child widget to the left edge of the attached dialog box widget, set the **adb_right_attachment** attribute to **DwtAttachOppAdb**. Figure 7–3 illustrates these two types of attachment. The shorter dotted line represents attachment of the right edge of the child widget to the same edge of the attached dialog box widget.

Figure 7–3 Attaching an Edge of a Child Widget to the Attached Dialog Box Widget



ZK-0443A-GE

When you specify this type of attachment, you can also specify the amount of space between the two edges. Specify this value in either pixels or font units in the attachment offset attribute. Use the unit of measure used in the attached dialog box widget. The default is font units. If you do not specify an offset, the borders of the two attached widgets abut each other.

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

You can also use an attached dialog box widget attribute to define default offsets; see Section 7.5.3 for more details.

Attachment to an edge of the attached dialog box widget is typically the first attachment you define when laying out an attached dialog box widget. For example, start by attaching the left edge of the widget in the upper left corner to the same edge of the attached dialog box widget. Then define attachments for the other widgets in the attached dialog box widget moving down and to the right inside the attached dialog box widget. The first attachment to the edge of the attached dialog box widget anchors the other attachments and helps avoid defining circular attachments.

7.5.1.2 Attaching an Edge to Another Child Widget

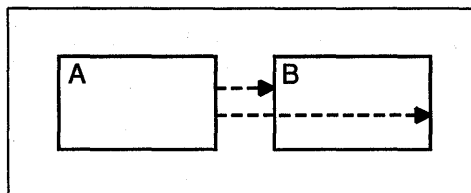
You can attach an edge of a child widget to either of the following two edges of another child widget in the attached dialog box widget:

- The opposite edge of the child widget (**DwtAttachWidget**)
- The same edge of the child widget (**DwtAttachOppWidget**)

Specify this type of attachment by passing the attachment type constants as the value of the attachment type attribute.

For example, set the **adb_right_attachment** attribute to **DwtAttachWidget** to attach the right edge of a child widget to the left edge of another child of the attached dialog box widget. To attach the right edge of a child widget to the right edge of another child of the attached dialog box widget, set the **adb_right_attachment** to **DwtAttachOppWidget**. Figure 7-4 illustrates these two types of attachments. In the figure, the shorter dotted line represents attachment of the right edge of child widget A to the left edge of the child widget B.

Figure 7-4 Attaching an Edge of a Child Widget to Another Child Widget in an Attached Dialog Box Widget



ZK-0444A-GE

When you specify this type of attachment, you must also specify the widget to which you are defining the attachment. Do this by passing the widget identifier of the child widget as the value of the attachment widget attribute. You can also specify the amount of space between the two widgets in the attachment offset attribute. Specify the amount in pixels or font units. The default is font units.

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

Note that while it is common practice to attach together two widgets in close proximity to each other, this is not a requirement. You can attach an edge of a child widget to an edge of any other child widget of the attached dialog box widget.

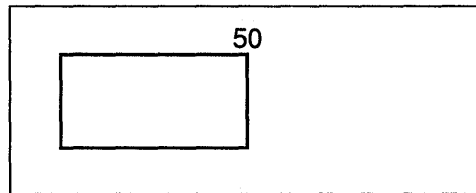
7.5.1.3 Attaching an Edge to a Position in the Attached Dialog Box Widget

You can attach an edge of a widget to a position in the attached dialog box widget. Instead of specifying the position by its x- and y-coordinates, specify the position as a fraction of the total dimension of the attached dialog box widget. This is called fractional positioning.

You specify this type of attachment by passing the attachment type constant **DwtAttachPosition** as the value of the attachment type attribute. You specify the position by supplying the numerator of the fractional position as the value of the attachment position attribute. The default denominator is 100, but you can change this default using an attached dialog box widget attribute, described in Section 7.5.3.

For example, the midpoint of the attached dialog box widget is one-half the distance between the two edges. To attach the right edge of a child widget to the midpoint of the attached dialog box widget, set the **adb_right_attachment** attribute to **DwtAttachPosition** and specify the numerator of the fractional position as 50 in the **adb_right_position** attribute. (The default denominator is 100.) Figure 7-5 illustrates this type of attachment.

Figure 7-5 Attaching an Edge to a Position in an Attached Dialog Box



ZK-0445A-GE

As with the other types of attachment, you can also specify an offset with fractional positioning. Specify the offset in pixels or font units in the attachment offset attribute. If you do not, the edge of the widget abuts the position specified. Note that the offset used is one-half the value you specify.

As a convenience, the XUI Toolkit includes an attachment type, **DwtAttachSelf**, that calculates the fractional position for you. This can be valuable when you do not know the size of the attached dialog box widget and therefore cannot calculate the fractional position yourself. Specify this attachment type by passing the attachment type constant **DwtAttachSelf** as the value of the attachment type attribute. Using this attachment type, you can explicitly position an edge of a child widget in the attached dialog box widget by its position and size attributes. The attached dialog box widget will calculate the relative position of the edge of the widget within the attached dialog box widget using these attributes.

7.5.1.4 Accepting Default Attachments

If you do not specify an attachment type or specify the attachment type **DwtAttachNone** for certain edges of a child widget, the attached dialog box widget will calculate the relative position of the edge according to its own defaults, depending on the setting of the **rubber_positioning** attribute. For more details about this topic, see Section 7.5.3.

7.5.2 Using Attachment Attributes

Maintaining the relationship between a simple text widget and a label widget is a common task that is well suited to an attached dialog box widget. If you specify fixed positions for these widgets, changes to the language or font size can disturb the layout. For example, the label widget could overlap the simple text widget. Using an attached dialog box widget, you can specify the relative positions of these two widgets so that they maintain their spatial relationship even if the language or font size changes.

Example 7-3 redefines the label and simple text widgets used in the DECburger user interface. In the example, the label and simple text widgets are contained in an attached dialog box widget, which is their parent. The attached dialog box widget is a child of the DECburger order entry dialog box widget. To maintain the original design of the DECburger interface, the border of the attached dialog box widget is set to zero width so it will not be visible in the user interface (see Figure 1-4).

Example 7-3 Positioning Children in an Attached Dialog Box Widget

```
①object
  fries_quantity_box : attached_dialog_box
  {
    arguments
    {
      x = 130;
      y = 100;
      border_width = 0;
    };
    controls
    {
      label fries_quantity_label;
      simple_text fries_quantity;
    };
  };
```

(continued on next page)

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

Example 7-3 (Cont.) Positioning Children in an Attached Dialog Box Widget

```
②object
  fries_quantity_label : label
  {
    arguments
    {
      adb_left_attachment = DwtAttachAdb;
      adb_left_offset = 0;
      adb_top_attachment = DwtAttachAdb;
      adb_top_offset = 0;
      label_label = k_quantity_label_text;
    };
  };

③object
  fries_quantity      :   simple_text
  {
    arguments
    {
      adb_left_attachment = DwtAttachWidget;
      adb_left_offset = k_label_stext_delta_x;
      adb_left_widget = label fries_quantity_label;

      adb_top_attachment = DwtAttachOppWidget;
      adb_top_offset = -1;
      adb_top_widget = label fries_quantity_label;

      max_length = 3;
      cols = 3;
      rows = 1;
      resize_width = false;
      resize_height = false;
      simple_text_value = k_0_label_text;
    };
  };
```

- ① In this UIL object declaration, the attached dialog box widget is defined. In the arguments section, the attached dialog box widget is positioned within its parent widget by specifying x- and y-coordinates. In the controls section of the object declaration, the label widget and the simple text widget are specified as children of the attached dialog box widget.
- ② This UIL object declaration defines the label widget. In the arguments section, the label attribute (called *label_label* in UIL) is set to the "Quantity" text string (defined as the constant *k_quantity_label_text* at the beginning of the UIL module).

In addition, the arguments section contains four attachment attributes. In these attachment attributes, the left edge of the label widget is attached to the right edge of the attached dialog box widget. (Note use of the attachment type **DwtAttachAdb**.) Similarly, the top edge of the label widget is attached to the same edge of the attached dialog box widget. (Note use of the attachment type constant **DwtAttachOppWidget**.) For both attachments, the edge of the child

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

widget abuts the edge of the attached dialog box because the offset is set to 0.

The label widget is upper-leftmost in the attached dialog box, so this attachment creates an anchor for other attachments.

- ③ The simple text widget is defined so that this horizontal orientation of the label and the simple text widgets is maintained no matter where the label widget may be moved. Note that the simple text widget in its arguments list makes reference to the label widget. Because of this dependency, you must list the simple text widget after the label widget in the controls list of the attached dialog box widget.

In the arguments section, the simple text widget is sized using simple text widget attributes that provide font independence. For more information about these attributes, see Section 9.2.2.1.

In addition, the arguments section contains six attachment attributes. The first three of these attributes define the attachment of the left edge of the simple text widget to the right edge of the label widget. (Note use of the **DwtAttachWidget** attachment type constant.) The widget identifier of the label widget is passed as the value of the **adb_left_widget** attribute, and the space between the two widgets is specified in the **adb_left_offset** attribute.

The second three attachment attributes define the attachment of the top edge of the simple text widget to the top edge of the label widget. (Note use of the **DwtAttachOppWidget** attachment type constant.) The widget identifier of the label widget is passed as the value of the **adb_left_widget** attribute, and the space between the two widgets is specified in the **adb_left_offset** attribute. (The example assigns a negative offset. Experimentation determined that this offset lines up the text in the label widget and the simple text widget, creating the best appearance.)

7.5.3 Customizing an Attached Dialog Box Widget

The attached dialog box widget supports all the attributes the dialog box widget supports. For information about customizing a dialog box widget, see Section 7.4.2.

In addition, the attached dialog box widget supports a unique set of attributes that enable you to customize the following aspects of its appearance and functioning:

- Default horizontal and vertical spacing between child widgets
- Default denominator used in fractional positioning
- Resizing behavior of child widgets

Creating Dialog Box Widgets

7.5 Creating an Attached Dialog Box Widget

7.5.3.1 Specifying the Default Spacing Between Child Widgets

Use the **default_vertical_offset** and **default_horizontal_offset** attributes to determine the amount of space between the edge of the child widget being attached and the edge or position to which it is being attached. Specify the offset in pixels or font units. The default is font units.

The **default_vertical_offset** attribute determines the offset for attachments of the top and bottom edges of a child widget. The **default_horizontal_offset** attribute determines the offset for attachments of the right and left edges of a child widget. By default, both offset attributes are set to 0.

7.5.3.2 Defining the Default Denominator Used in Fraction Positioning

Use the **fraction_base** attribute to specify the denominator used in fractional positioning. The default denominator is 100.

7.5.3.3 Controlling Resizing Behavior of Child Widgets

The **rubber_positioning** attribute determines how child widgets behave when the attached dialog box widget is resized. This attribute acts in coordination with several of the attachment attributes.

For example, when rubber positioning is set to true and both the left and right edges of a child widget have no explicit attachments defined, the attached dialog box attaches both edges to their initial positions (see Section 7.5.1.3). In this case, when the attached dialog box widget is resized, the child widget will stretch so that its new size encompasses the same percentage of the attached dialog box widget width.

If rubber positioning is set to false and both the left and right edges have no explicit attachments defined, the attached dialog box widget attaches only the left edge to the left edge of the attached dialog box widget. When the attached dialog box widget is resized, both edges maintain their original position in the attached dialog box widget.

Similarly, if rubber positioning is set to true and both the top and bottom edges of a child widget have no explicit attachments defined, the attached dialog box widget attaches both edges to their initial positions (see Section 7.5.1.3). If rubber positioning is set to false and both the top and bottom edges of a child widget have no explicit attachments defined, the attached dialog box widget attaches the top edge to the same edge of the attached dialog box widget.

7.5.4 Associating Callback Routines with an Attached Dialog Box Widget

The attached dialog box widget supports the same set of callback attributes as the dialog box widget. For information about associating a callback routine with a dialog box widget, see Section 7.4.3.

8

Creating a List Box Widget

This chapter provides the following:

- An overview of the list box widget in the XUI Toolkit
- A detailed description of how to create a list box widget for an application
- A description of the list box widget support routines provided in the XUI Toolkit

8.1

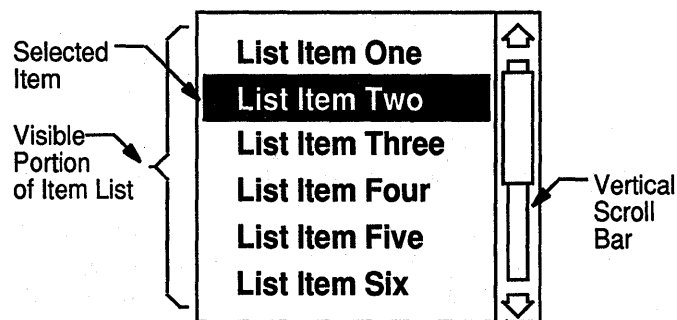
Overview of the List Box Widget

The list box widget is a rectangular window containing the visible portion of an **item list**. The list box widget is similar to the menu widgets described in Chapter 6 in that they both allow you to present the users of your application with a list of choices. As with a menu widget, a user can select an item in the list box widget by moving the pointer cursor onto the item and clicking MB1. The list box widget notifies your application when a list item has been selected using the callback mechanism.

Unlike the menu widgets, the list box widget includes a scroll bar widget that provides users with the ability to scroll through the list of choices. The list box widget is a window onto a portion of a larger list of choices. The user can scroll backwards and forwards to view the complete item list. This capability makes the list box widget preferable to the menu widgets for presenting many choices. Typically, applications use menu widgets to present lists of choices where each choice performs an action; they use list box widgets to present lists of choices where each choice is an option.

Figure 8–1 shows a list box widget and its components.

Figure 8–1 List Box Widget



ZK-0399A-GE

Creating a List Box Widget

8.2 Creating a List Box Widget

8.2

Creating a List Box Widget

While the list box widget is similar to a menu widget, you use a different procedure to create it. You build a menu widget as a widget hierarchy where you create the menu items as widget or gadget children of the parent menu widget. In a list box widget, you create the list items as an array of text strings. You pass the address of the array to the list box widget as the value of the list box widget **items** attribute.

Because the list items appear on the display, you must convert the text strings to compound strings before passing them to the list box widget. Use any of the compound string routines described in Section 5.6 to convert the list item text strings to compound strings.

To create a list box widget, perform the following steps:

1 Create the item list.

Create an array of compound strings with each string representing one list item. Section 8.2.1 describes this procedure.

2 Create the list box widget.

Use any of the widget creation mechanisms listed in Table 8–1. The choice of mechanism depends on the attributes of the list box widget you need to access.

Pass the address of the item list in the **items** attribute when you create the list box widget.

Table 8–1 List Box Widget Creation Mechanisms

High-level routine	Use the LIST BOX routine to create a list box widget.
Low-level routine	Use the LIST BOX CREATE routine to create a list box widget.
UIL object type	Use the list_box object type to define a list box widget in a UIL module. At run time, the DRM routine FETCH WIDGET will create the widget according to this description.

3 Manage the list box widget.

Use the intrinsic routine **MANAGE CHILD** to manage the list box widget. If you use UIL to define the user interface, this step is not necessary. By default, DRM manages the widgets it creates.

After you complete these steps, if the parent of the list box widget has been realized, the list box widget will appear on the display.

Low-level widget routines and UIL provide access to the complete set of widget attributes at creation time. High-level routines provide access only to a subset of these widget attributes at widget creation time. (To access attributes not available using a high-level routine, use the **SET VALUES** intrinsic routine after the widget has been created.) Table 8–2 lists the attributes available using the high-level routine **LIST BOX**. Pass values for these attributes as arguments to the routine.

Creating a List Box Widget

8.2 Creating a List Box Widget

Table 8–2 Attributes Accessible Using the High-Level Routine LIST BOX

x	Specifies the x-coordinate of the upper left corner of the widget.
y	Specifies the y-coordinate of the upper left corner of the widget.
items	Specifies the array of list items.
item_count	Specifies the number of items in the item list.
visible_item_count	Specifies the number of items that should appear in the list box widget window.
callback	Specifies the address of a callback routine list.
help_callback	Specifies the address of a callback routine list.
resize	Specifies whether the list box allows horizontal resizing.
horiz	Specifies whether the list box includes a horizontal scroll bar.

8.2.1 Creating an Item List

There are two ways you can create an item list:

- Create an array of compound strings using the capabilities of the programming language in which you are writing your application.
- Use the UIL built-in function `STRING TABLE` to create an array of text strings independent of the programming language used.

Note that all items in the list must be unique. A list box widget cannot contain two items containing the same text.

8.2.1.1 Creating an Item List as an Array of Compound Strings

An item list is an array of pointers to compound strings. Pass the address of the array to the list box widget as the value of the **items** attribute. When you pass an item list to a list box widget, you must also pass it the number of items in the item list. Use the **item_count** attribute to do this.

Example 8–1 creates the item list shown in Figure 8–1.

Creating a List Box Widget

8.2 Creating a List Box Widget

Example 8-1 Creating an Item List as an Array of Compound Strings

```
.
.
.
①static DwtCompString *list_items[] = NULL;
.
.
.
int item_count = 0;

②list_items[item_count++] = DwtLatin1String("List Item One");
list_items[item_count++] = DwtLatin1String("List Item Two");
list_items[item_count++] = DwtLatin1String("List Item Three");
list_items[item_count++] = DwtLatin1String("List Item Four");
list_items[item_count++] = DwtLatin1String("List Item Five");
list_items[item_count++] = DwtLatin1String("List Item Six");
list_items[item_count++] = DwtLatin1String("List Item Seven");
list_items[item_count++] = DwtLatin1String("List Item Eight");
list_items[item_count++] = DwtLatin1String("List Item Nine");
list_items[item_count++] = DwtLatin1String("List Item Ten");
list_items[item_count++] = DwtLatin1String("List Item Eleven");
list_items[item_count++] = DwtLatin1String("List Item Twelve");

③XtSetArg( arglist[0], DwtNitems, list_items );
XtSetArg( arglist[1], DwtNitemsCount, item_count );
XtSetArg( arglist[2], DwtNvisibleItemsCount, 6 );

④list_box = DwtListBoxCreate( main_widget, "list", arglist, 3 );

⑤XtManageChild( list_box );
.
.
.
```

-
- ① The example declares an array, named *list_items*, of pointers to compound strings.
 - ② In this group of assignment statements, the example creates an item list by assigning the address of a compound string to each element of the *list_items* array. The example uses the compound string routine LATIN1 STRING to convert the text strings to compound strings.
 - ③ After creating the array of list items, the example assigns the address of the array as the value of the **items** attribute. The example uses the SET ARG intrinsic routine to do this. The example also assigns the number of items in the list as the value of the **item_count** attribute.
 - ④ In this statement, the example creates the list box widget using the low-level widget creation routine LIST BOX CREATE. The example passes the argument list as a parameter to the low-level routine along with the count of the number of arguments in the argument list.
 - ⑤ The program fragment ends by managing the list box widget using the MANAGE CHILD intrinsic routine.

Creating a List Box Widget

8.2 Creating a List Box Widget

8.2.1.2 Creating an Item List Using the UIL STRING TABLE Function

UIL includes the built-in function `STRING TABLE`, which you can use to create an array of text strings to create an item list independent of the programming language you use. Using UIL, you can declare the item list in a UIL module as a named value and use the `STRING TABLE` function to create the value. You can use the value name to refer to the item list throughout the remainder of the UIL module.

To create an item list using the `UIL STRING TABLE` function, pass the function a list of text strings, delimiting each string with quotation marks or apostrophes. Separate the text strings with commas. UIL creates an array of list items out of the text strings, converting the text strings to compound strings automatically. You can then pass this value by name in the arguments list of the UIL object declaration. For more information about defining string tables in UIL, see Section 3.2.7.5.

The DECburger sample application uses a list box widget to present a selection of drink choices to the user. In Example 8–2, DECburger uses the `STRING TABLE` function to create the item list for the drink selection list box.

Example 8–2 Creating an Item List Using the UIL STRING TABLE Function

```
.
.
.
❶ k_drink_list_text      : string_table ('Apple Juice',
                                     'Orange Juice', 'Grape Juice',
                                     'Cola', 'Punch', 'Root beer',
                                     'Water', 'Ginger Ale', 'Milk',
                                     'Coffee', 'Tea');
k_drink_list_select     : string_table('Apple Juice');
.
.
object
  drink_list_box : list_box {
    arguments {
      x = 230;
      y = 22;
      visible_items_count = 4;
      ❷ items = k_drink_list_text;
      selected_items = k_drink_list_select;
      single_selection = true;
    };
    callbacks {
      single = procedure list_proc (k_drink_list);
    };
  };
.
.
.
```

- ❶ The DECburger sample application creates the item list for the drink selection list box widget using the `UIL STRING TABLE` function. The name of the item list is `k_drink_list_text`. (Note that, in the example, DECburger creates a second item list, named `k_drink_list_select`. This

Creating a List Box Widget

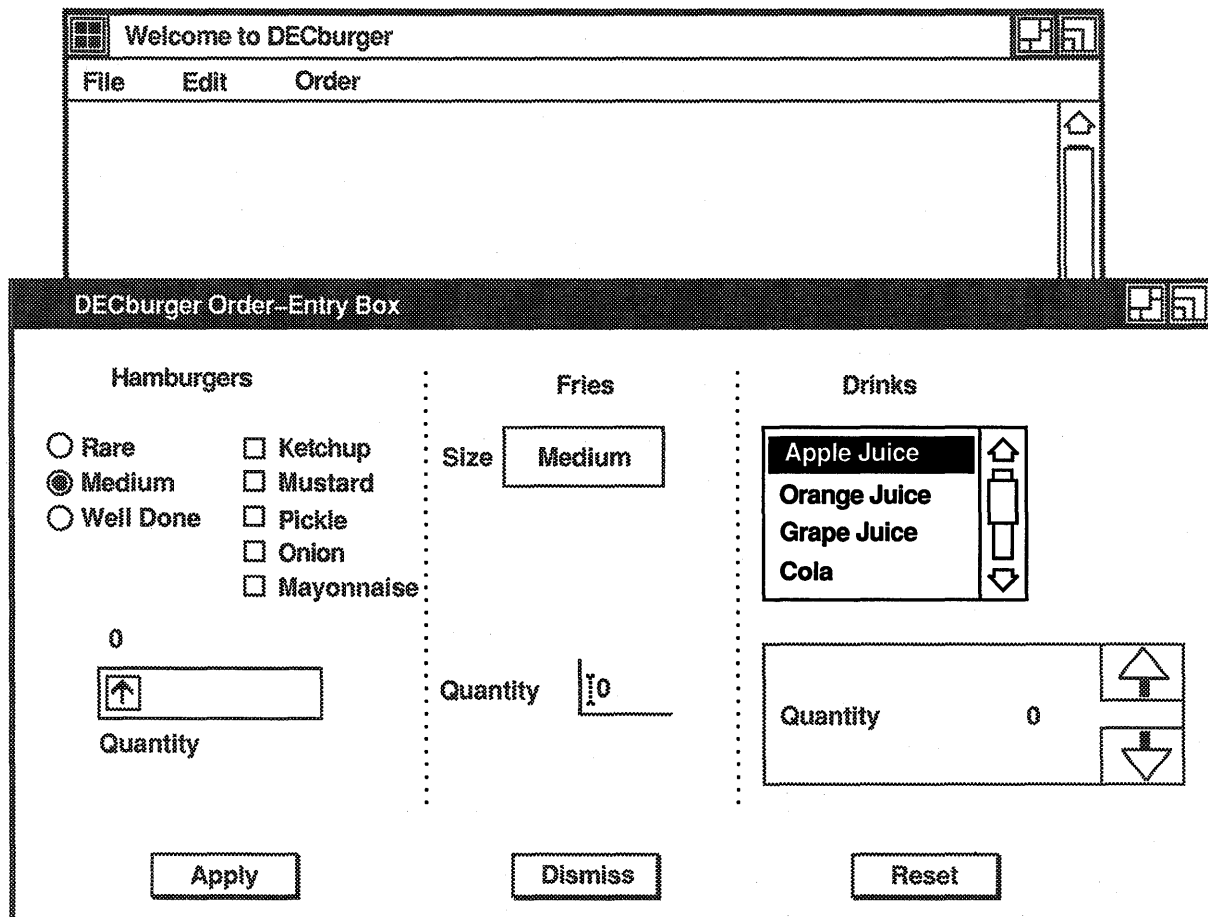
8.2 Creating a List Box Widget

item list is used to select a list item as the default choice in the drink list box. Section 8.2.2 describes this procedure.)

- ② DECburger passes the item list by name, *k_drink_list_text*, to the list box widget as the value of the **items** attribute.

Figure 8-2 shows the drink selection list box widget as it appears in the DECburger user interface.

Figure 8-2 List Box Widget Used in the DECburger User Interface



ZK-0404A-GE

8.2.2 Selecting and Canceling Selections of List Items

The user of an application can select an item from a list box widget by moving the pointer cursor onto the item and clicking MB1. When an item is selected, the list box widget does the following:

- Highlights the selected item

Creating a List Box Widget

8.2 Creating a List Box Widget

- Adds the item to the list of selected items maintained in the **selected_items** attribute
- Notifies the application, using the callback mechanism, that an item has been selected
- Updates the count of selected items maintained in the **selected_item_count** attribute

When a list box widget performs a callback to notify your application that a list item has been selected, the list box widget also informs your application whether the user selected the item with a single click on MB1 or a double click on MB1. The list box widget distinguishes between these two types of selections and informs your application which type caused the selection. Your application can perform different processing depending on how an item was selected. (Section 8.3.4 describes how you can associate callback routines with either type of selection.)

By default, the list box widget only allows one item to be selected at a time. When the user clicks MB1 on an item when another item is selected, the list box widget cancels the selection on the previous item. When a selection is canceled, the list box widget turns off the highlighting of the item and removes the item from the list of selected items.

You can configure the list box to allow selection of multiple items by setting the **single_selection** attribute to false. To select additional items, the user presses the shift key in conjunction with the click of MB1 on an item.

Example 8–3 shows how DECburger makes the apple juice option the default selection of the drink selection list box widget.

Example 8–3 Selecting an Item in an Item List

```

.
.
.
k_drink_list_text      : string_table ('Apple Juice',
                                     'Orange Juice', 'Grape Juice',
                                     'Cola', 'Punch', 'Root beer',
                                     'Water', 'Ginger Ale', 'Milk',
                                     'Coffee', 'Tea');
❶ k_drink_list_select  : string_table('Apple Juice');
.
.
object
  drink_list_box : list_box {
    arguments {
      x = 230;
      y = 22;
      visible_items_count = 4;
      items = k_drink_list_text;
❷   selected_items = k_drink_list_select;
      single_selection = true;
    };
    callbacks {
```

(continued on next page)

Creating a List Box Widget

8.2 Creating a List Box Widget

Example 8-3 (Cont.) Selecting an Item in an Item List

```
single = procedure list_proc (k_drink_list);  
};  
};  
.  
.  
.
```

- ❶ DECburger creates an item list, called *k_drink_list_select*, using the UIL built-in function `STRING TABLE`. This string table contains only one item: apple juice.
- ❷ DECburger passes this single-item string table as the value of the `selected_items` attribute in the object declaration of the list box widget. When the list box widget appears in the DECburger user interface, the apple juice item is highlighted.

8.3 List Box Widget Support Routines

The XUI Toolkit provides a set of support routines for use with the list box widget. These routines provide convenient ways to perform commonly needed tasks, such as adding items to an item list. Table 8-3 lists these support routines; the following sections describe how to use them.

Table 8-3 List Box Widget Support Routines

LIST BOX ADD ITEM	Adds an item to a list box widget.
LIST BOX DELETE ITEM	Deletes an item, identified by its content, from a list box widget.
LIST BOX DELETE POSITION	Deletes an item, identified by its position in the item list, from a list box widget.
LIST BOX SELECT ITEM	Highlights the item in the list box widget, if it is visible, and adds it to the list of selected items. When you use this routine to select an item, you can optionally specify that the list box widget use the callback mechanism to notify your application that an item has been selected.
LIST BOX DESELECT ITEM	Removes an item that had previously been selected from the list of selected items and turns off highlighting of the item.
LIST BOX DESELECT ALL ITEMS	Removes all selected items from the list of items in the list box widget, turning off highlighting of the visible items.
LIST BOX SET ITEM	Specifies which item, identified by its content, will appear at the top of the visible items displayed in the list box widget.

(continued on next page)

Creating a List Box Widget

8.3 List Box Widget Support Routines

Table 8–3 (Cont.) List Box Widget Support Routines

LIST BOX SET POS	Specifies which item, identified by its position in the list, will appear at the top of the visible items displayed in the list box widget.
LIST BOX SET HORIZ POS	Specifies a horizontal position in a list item where all text to the left of the position will not appear on the display. Specify the position in pixels. The text of the list item that is not currently visible can be viewed using the horizontal scroll bar, if the list box widget includes one.
LIST BOX ITEM EXISTS	Verifies that an item, identified by its content, is currently in the list contained in the list box widget.

8.3.1 Adding and Deleting List Items at Run Time

To add or delete an item from a list box widget after the list box widget has been created, use one of the following two methods:

- The SET VALUES intrinsic routine
- The list box widget support routines

The support routines offer several advantages over the SET VALUES routine:

- The support routines use fewer system resources than the SET VALUES routine and are, therefore, more efficient.
- The support routines renumber the positions of items in the item list to accommodate the change in the list.
- The support routines automatically update the item count.

Use the support routines if you need to add or delete a single item in an item list; use the SET VALUES intrinsic routine when you need to update the entire item list.

8.3.1.1 Using SET VALUES to Add or Delete List Items

To add or delete list items using the SET VALUES intrinsic routine, you must create a new item list, inserting or deleting list items as necessary. You then assign the address of the new item list to the **items** attribute of the list box widget using the SET VALUES intrinsic routine. Whenever you modify an item list, you must also update the item count maintained in the **item_count** attribute and the selected item count maintained in the **selected_item_count** attribute.

When using the SET VALUES intrinsic routine to dynamically add items to an item list, be careful to allocate memory for the items in the original list. Do not simply add items to the item list returned by the GET VALUES intrinsic routine. The GET VALUES intrinsic routine returns pointers to the list box widget's copies of the strings. When you pass a new item list as the value of the **items** attribute using the SET VALUES intrinsic routine, the list box widget releases the memory that it used to

Creating a List Box Widget

8.3 List Box Widget Support Routines

store the original item list and allocates new memory for the new item list. Thus, the pointers returned by the GET VALUES intrinsic routine for the original list will be meaningless.

8.3.1.2 Using a Support Routine to Add an Item to an Item List

To add a single item to an item list, use the LIST BOX ADD ITEM support routine. This routine takes the following three arguments:

- Widget identifier of the list box widget
- Compound string that is the new list item
- Position in which you want the new item to appear in the list

The list box widget identifies each item in an item list by its position number. The first item in the list is numbered 1 (not 0) with each subsequent item numbered sequentially. If you pass the position parameter as null, the list box widget adds the new item to the bottom of the item list.

The DECburger sample application uses the LIST BOX ADD ITEM support routine to add items to the total order list box widget. In Example 8-4, DECburger adds the item to the bottom of the list by specifying the position parameter as 0.

Example 8-4 Adding an Item to a List Box Widget

```
.  
. .  
{  
    sprintf(list_buffer, "%d ", fries_num);  
    list_txt = DwtLatin1String(list_buffer);  
  
    list_txt = DwtCStrcat(list_txt, current_fries);  
    list_txt = DwtCStrcat(list_txt, latin_space);  
  
    list_txt = DwtCStrcat(list_txt, name_vector[k_fries_index]);  
    DwtListBoxAddItem(widget_array[k_total_order], list_txt, 0);  
}  
. .  
.
```

8.3.1.3 Using a Support Routine to Delete an Item from an Item List

The XUI Toolkit provides two support routines for deleting a single item from an item list. The support routines allow you to specify the item to be deleted in two ways:

- By the text content of the item
- By the position of the item in the item list

To delete an item by specifying its content, use the LIST BOX DELETE ITEM support routine. Note that the item must be unique. This routine takes the following two arguments:

- Widget identifier of the list box
- Compound string identifying the list item to be deleted

Creating a List Box Widget

8.3 List Box Widget Support Routines

To delete an item by specifying its position, use the LIST BOX DELETE POSITION support routine. This routine takes the following two arguments:

- Widget identifier of the list box
- Position of the item in the item list to be deleted

To determine if a list box widget contains a particular item in its item list, use the support routine LIST BOX ITEM EXISTS. This routine takes the following two arguments:

- Widget identifier of the list box
- Compound string identifying the list item

8.3.2 **Selecting and Canceling the Selection of List Items at Run Time**

To select or cancel the selection of an item in a list box widget after the list box widget has been created, use one of the following two methods:

- The SET VALUES intrinsic routine
- The list box widget support routines

The support routines offer several advantages over the SET VALUES routine:

- The support routines use fewer system resources than the SET VALUES routine and are, therefore, more efficient.
- The support routines automatically update the count of selected items.

Use the support routines if you need to select or cancel the selection of a single item in an item list; use the SET VALUES intrinsic routine when you want to select more than one item.

8.3.2.1 **Using the SET VALUES Intrinsic Routine to Select List Items**

To select items in an item list using the SET VALUES intrinsic routine, first create a list of selected items. Then assign this list as the value of the **selected_items** attribute using the SET VALUES routine. Whenever you modify the list of selected items, you must update the value of the **selected_item_count** attribute.

Note that passing a list of selected items using the SET VALUES intrinsic routine overwrites the current list of selected items. This cancels the selection of items that had been selected but that are not in the new selected item list.

8.3.2.2 **Using a Support Routine to Select a List Item**

To select a single item in a list box widget, use the LIST BOX SELECT ITEM support routine. This routine takes the following three arguments:

- The widget identifier of the list box
- The compound string that identifies the list item

Creating a List Box Widget

8.3 List Box Widget Support Routines

- A Boolean value that indicates whether the list box widget should notify the application through the callback mechanism that the item has been selected

8.3.2.3 Canceling the Selection of Items in an Item List

You can cancel the selection of an item using the `SET VALUE` intrinsic routine or by using a list box widget support routine. The XUI Toolkit provides two support routines for canceling selections. Using the support routines you can:

- Cancel the selection of a single selected item in an item list
- Cancel the selection of all selected items in an item list

To cancel the selection of a single item from a list box widget, use the `LIST BOX DESELECT ITEM` support routine. This routine takes the following two arguments:

- The widget identifier of the list box
- The compound string identifying the item whose selection you want to cancel

To cancel the selection of all the items currently selected in a list box widget, use the `LIST BOX DESELECT ALL ITEMS` support routine. This routine accepts only the widget identifier of the list box widget as an argument.

8.3.3 Customizing the Appearance of a List Box Widget

The attributes of the list box widget enable you to customize the following aspects of its appearance:

- Size
- List of visible items
- Margins and spacing

8.3.3.1 Specifying the Size of a List Box Widget

You can specify the size of a list box widget in pixels using the common widget attributes `width` and `height`. However, as with all widgets that display text, using these attributes to specify the size of a list box widget makes the widget font-dependent. If the font size changes, fewer items will be visible in the list box widget, and some items may be clipped.

Instead of specifying the height of a list box in pixels, specify the number of items in the item list that should be visible at any time using the `visible_items_count` attribute. The list box widget calculates its height dimension, to the degree allowed by its parent, based on the number of items you specify to be visible and on the font size.

By default, the list box widget determines its width from the width of the longest list item. If you specify a width in the `width` attribute and set the `resize` attribute to false, the rightmost portion of list items that extend beyond the right border of the list box will not be visible. To enable a user

Creating a List Box Widget

8.3 List Box Widget Support Routines

to view this information, include a horizontal scroll bar in the list box widget. To include a horizontal scroll bar, set the **horiz** attribute to true.

In Example 8–5, DECburger specifies the height of the drink selection list box by specifying the number of visible items in the **visible_item_count** attribute. Of the total of 11 items in the item list, DECburger specifies that 4 items be visible.

Example 8–5 Specifying the Size of the DECburger List Box Widget

```
.
.
.
object
  drink_list_box : list_box {
    arguments {
      x = 230;
      y = 22;
      visible_items_count = 4;
      items = k_drink_list_text;
      selected_items = k_drink_list_select;
      single_selection = true;
    };
    callbacks {
      single = procedure list_proc (k_drink_list);
    };
  };
.
.
.
```

If your list is made up of long items that would make the list box wider than you want it to be, you can position the left border of the list box so that a specific amount of each list item will not be visible. The XUI Toolkit file selection widget provides an example of this attribute. The file selection widget positions the list of file specifications in a directory so that only the file name part of the file specification is visible. The device and directory parts of the file specifications, which appear at the left of a file specification, are not visible.

Specify the horizontal position in pixels using the list box widget support routine LIST BOX SET HORIZ POS. The user can view this information using the horizontal scroll bar widget, if one is included in the list box widget. The routine takes the following two arguments:

- The widget identifier of the list box
- The horizontal position in the list item where you want it to begin to appear on the display

Note that the position argument for this routine specifies a point, measured in pixels, along the horizontal axis of a list item. The other list box widget support routines use the term *position* to refer to the order of elements in the item list array.

Creating a List Box Widget

8.3 List Box Widget Support Routines

8.3.3.2 Specifying List Items to Be Visible

By default, the list box widget displays the list item at position 1 as the topmost visible item. However, you can specify that another item in the item list appear as the topmost item. The XUI Toolkit provides two list box widget support routines that you can use to specify which item appears as the topmost visible item in the list. In conjunction with the **visible_items_count** attribute, you can use these support routines to determine what list items are visible in the list box widget.

You can specify the topmost item in two ways:

- By the text of the list item
- By the position of the item in the item list

To specify the topmost item by text content, use the **LIST BOX SET ITEM** routine. This routine takes the following two arguments:

- The widget identifier of the list box
- The compound string that identifies the item

To specify the topmost item by position, use the **LIST BOX SET POS** routine. This routine takes the following two arguments:

- The widget identifier of the list box
- The position of the item in the list

Your choice of topmost item is limited by the number of items in the list and the number of visible items you have specified. For example, in a list box widget with 10 items, if you specify that 5 items should always be visible, you cannot specify item 8 as the topmost item. The list box widget will always display 5 items to the limit of the available items.

The list box widget always attempts to display the last selected item as close to the top of the list box as it can, depending on the number of items in the list and the size of the list box.

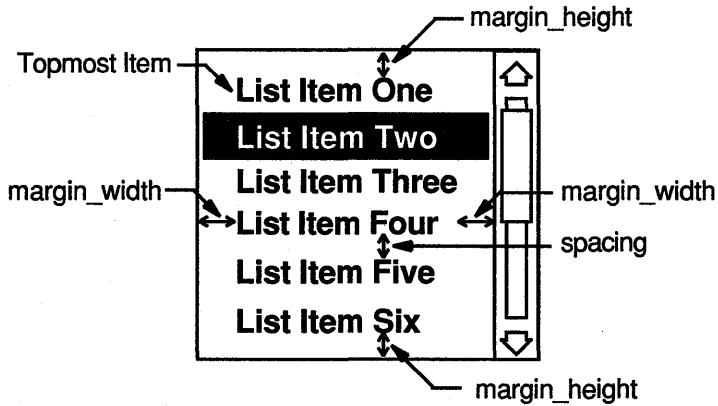
8.3.3.3 Specifying Margins and Spacing in a List Box Widget

You can determine the amount of space around the items in a list box widget by assigning values to the **margin_width**, **margin_height**, and **spacing** attributes. Specify these values in pixels. Figure 8-3 illustrates these margins.

Creating a List Box Widget

8.3 List Box Widget Support Routines

Figure 8-3 Margins and Spacing in a List Box Widget



ZK-0400A-GE

8.3.4 Associating Callbacks with a List Box Widget

When an item in an item list is selected, a list box widget notifies your application using the callback mechanism. A user can select an item by moving the pointer cursor onto the item and clicking MB1. Your program can select an item using a list box widget support routine or the SET VALUES intrinsic routine.

When the list box widget performs a callback, it returns callback data to the application. In this callback data, the list box widget identifies which list item has been selected. For complete information about the data returned in a callback by the list box widget, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a list box widget callback, pass a callback routine list to one of the four list box widget callback attributes. The list box widget supports four distinct callbacks, depending on the type of user interaction. Table 8-4 describes what conditions trigger these callbacks and the widget attributes you use to associate callback routines with them.

Table 8-4 List Box Widget Callbacks

Callback Attribute	Description
single_callback	A user has clicked MB1 on an item in the list box widget, causing it to be selected, or your application has selected the item using the support routine LIST BOX SELECT ITEM.

(continued on next page)

Creating a List Box Widget

8.3 List Box Widget Support Routines

Table 8–4 (Cont.) List Box Widget Callbacks

Callback Attribute	Description
extend_callback	A user has selected an item in the list box widget, and there is already at least one other item selected.
single_confirm_callback	A user has selected an item in the list box widget by double clicking MB1 on the item.
extend_confirm_callback	A user has double clicked MB1 on an item in the list box widget, and there is already at least one other item selected.

DECburger specifies that only one item can be selected at a time in the drink selection list box widget. In Example 8–6, DECburger associates a callback routine by assigning the symbolic reference for the callback routine as the value of the **single_callback** attribute. DECburger does not associate a callback with a double click on a list item.

Example 8–6 Associating a Callback Routine with a List Box Widget

```

.
.
.
object
  drink_list_box : list_box {
    arguments {
      x = 230;
      y = 22;
      visible_items_count = 4;
      items = k_drink_list_text;
      selected_items = k_drink_list_select;
      single_selection = true;
    };
    callbacks {
      single = procedure list_proc (k_drink_list);
    };
  };
.
.
.

```

Example 8–7 shows the callback routine used by the DECburger drink selection list box widget. The list box widget returns the text of the selected item in its callback data structure. The callback routine reads this data from the data structure and assigns it to the static variable *current_drink*. DECburger maintains the current drink selection in this variable.

Creating a List Box Widget

8.3 List Box Widget Support Routines

Example 8-7 Callback Routine DECburger Associates with the List Box Widget

```
.
.
.
static void list_proc(w, tag, list)
    Widget w;
    int *tag;
    DwtListBoxCallbackStruct *list;
{
    current_drink = list->item;
}
.
.
.
```

The XUI Toolkit file selection widget provides an example of how to use a double click callback. The file selection widget contains a list box widget in which it displays the list of files in a specified directory. When a user selects a file from the list with a single click, the list box widget notifies the file selection widget using the callback mechanism. In response, the file selection widget displays the file in its text field. This allows the user to edit the file name, if necessary.

When a user double clicks MB1 on a file name, the file selection widget displays the file name in its text field and also activates the default push button OK in the file selection widget. In this context, the file selection widget interprets the double click as confirmation of the operation.

9

Handling Text

This chapter provides the following:

- An overview of the text widgets in the XUI Toolkit
- A description of the support routines used with the text widgets

9.1

Overview of Text Widgets

The XUI Toolkit includes two widgets that you can use to provide your application with text editing capabilities:

- Simple text widget
- Compound string text widget

Both text widgets enable users of your application to enter text or edit existing text using the keyboard. The difference between the two widgets is that the simple text widget supports manipulation of a null-terminated array of characters; the compound string text widget supports compound strings.

In a compound string, you specify not only the characters in the text string, but also the character set and writing direction you want for displaying the text string on a workstation screen. All the XUI Toolkit widgets that contain text labels use compound strings to represent these labels. By using the compound string text widget, you enable users of your application to enter and edit text in the same character set and writing direction used throughout the user interface of your application.

The simple text and compound string text widgets have the same visual appearance (see Figure 9–1). In compliance with the *XUI Style Guide*, both widgets appear on the display by default as two perpendicular lines forming a right angle. These lines, known together as the **half-border**, mark off the text entry area. In the simple text widget, the half-border is made up of the left and bottom borders of the widget. The compound string text widget uses the left and bottom borders to form the half-border if the main writing direction of the compound string it contains is left-to-right. If the main writing direction is right-to-left, the compound string text widget uses the right and bottom borders to create the half-border. (For information about how the compound string text widget determines the main writing direction, see Section 9.2.2.7.)

The text entry area marked off by the half-border contains a text cursor that indicates where text will be inserted. When the widgets have input focus, the text cursor blinks and is displayed at full brightness. When the widgets do not have input focus, the text cursor appears dimmed and does not blink.

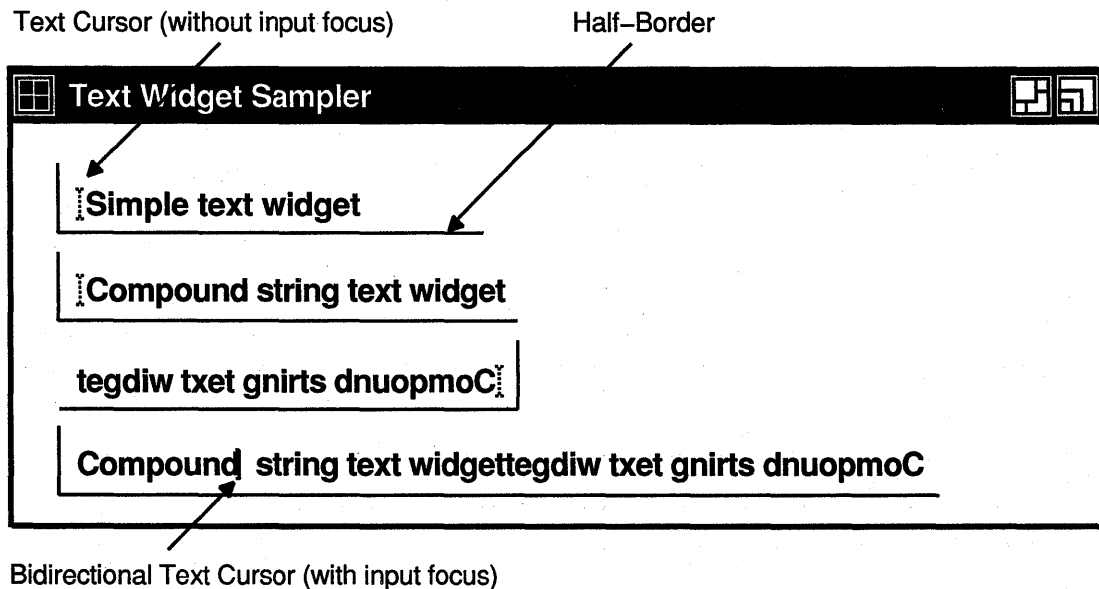
Handling Text

9.1 Overview of Text Widgets

The text cursor in the compound string text widget can also indicate the current **editing direction**. The editing direction is the direction in which characters can be inserted or deleted. Users of your application can switch between the left-to-right and right-to-left editing directions by pressing the toggle key (F17). Whenever a user changes the editing direction in a compound string text widget, the shape of the text cursor, called a **bidirectional text cursor**, can change to indicate the new editing direction. When the compound string text widget does not have the input focus, it contains a dimmed, standard text cursor. For information on how to create a compound string text widget with a bidirectional text cursor, see Section 9.2.2.7.

Figure 9-1 illustrates both the simple text and compound string text widgets. Note how the first compound string text widget, which contains a compound string using the ISO Latin1 character set and the left-to-right writing direction, appears identical to the simple text widget. The second compound string text widget, which contains a compound string using the right-to-left writing direction, illustrates a half-border made from the right and bottom borders of the widget. The third compound string text widget contains a compound string made up of two segments, each with a different writing direction. This compound string text widget, shown in the figure with input focus, also illustrates a bidirectional text cursor.

Figure 9-1 Text Widgets



ZK-1275A-GE

The widgets use the callback mechanism to notify your application when the text they contain changes. Note, however, that the widgets do not return the text in the callback. To retrieve the text, you must use the GET VALUES intrinsic routine or one of the support routines provided by the

Handling Text

9.1 Overview of Text Widgets

XUI Toolkit for use with the text widgets. (For more information about this topic, see Section 9.2.1.2.)

The XUI Toolkit includes support routines for many commonly performed tasks, such as specifying the text contained in the text widget. Table 9–1 lists these support routines by function; the following sections describe how to use them.

Table 9–1 Text Widget Support Routines

Simple Text Widget	Compound String Text Widget	Description
Manipulating the Text Content of the Widget		
S TEXT GET STRING	CS TEXT GET STRING	Retrieves the current text contents of the widget.
S TEXT SET STRING	CS TEXT SET STRING	Replaces the text contents of the widget with completely new text.
S TEXT REPLACE	CS TEXT REPLACE	Replaces the portion of the current text contents of the widget, specified by the start and end point positions, with new text.
Modifying Widget Behavior		
S TEXT GET EDITABLE	CS TEXT GET EDITABLE	Returns a Boolean value that indicates whether the user of the application can edit the current text contents of the widget. When this routine returns true, the user can edit the text; when it returns false, the user cannot edit the text.
S TEXT SET EDITABLE	CS TEXT SET EDITABLE	Sets the Boolean value that indicates whether the user can edit the current text contents of the widget. To allow editing, set this value to true.
S TEXT GET MAX LENGTH	CS TEXT GET MAX LENGTH	Returns the maximum length of text that the widget will allow a user to enter.
S TEXT SET MAX LENGTH	CS TEXT SET MAX LENGTH	Sets the maximum length of the text that the user can enter in the widget.
Handling Text Selections		
S TEXT GET SELECTION	CS TEXT GET SELECTION	Retrieves the text in the widget that has been selected using the selection mechanism. Selected text is highlighted on the display.

(continued on next page)

Handling Text

9.1 Overview of Text Widgets

Table 9–1 (Cont.) Text Widget Support Routines

Simple Text Widget	Compound String Text Widget	Description
Handling Text Selections		
S TEXT SET SELECTION	CS TEXT SET SELECTION	Sets the text in the widget or the portion of the text specified by the start and end point positions as the selection, and highlights the text on the screen.
S TEXT CLEAR SELECTION	CS TEXT CLEAR SELECTION	Cancels the selection of text in the widget and turns off highlighting of the text.

9.2

Creating Text Widgets

To create a simple or compound string text widget, perform the following steps:

- 1 Create the text widget using any of the widget creation mechanisms listed in Table 9–2.

Choose the mechanism that provides access to the widget attributes you need to set.

Table 9–2 Mechanisms for Creating Text Widgets

Mechanism	Simple Text Widget	Compound String Text Widget
High-level routine	Use the S TEXT routine to create a simple text widget.	Use the CS TEXT routine to create a compound string text widget.
Low-level routine	Use the S TEXT CREATE routine to create a simple text widget.	Use the CS TEXT CREATE routine to create a compound string text widget.
UIL object type	Use the UIL object type <code>simple_text</code> to define a simple text widget in a UIL module. At run time, the DRM routine <code>FETCH WIDGET</code> creates the widget according to this definition.	Use the UIL object type <code>cs_text</code> to define a compound string text widget in a UIL module. At run time, the DRM routine <code>FETCH WIDGET</code> creates the widget according to this definition.

- 2 Manage the text widget using the intrinsic routine `MANAGE CHILD`.

After you complete these steps, the text widget will appear on the display if its parent has been realized.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To assign values to attributes not available at widget creation time, use the `SET VALUES` intrinsic routine or support routine after the widget has been created.) Table 9–3 lists the attributes that you can set if you use the high-level routine `S TEXT` to create a simple text widget or if you use the `CS TEXT` routine to create a compound string widget. Pass the values of these attributes as arguments to the routine.

Handling Text

9.2 Creating Text Widgets

Table 9-3 Attributes Accessible Using the High-Level Routines S TEXT and CS TEXT

Attribute	Description
x	Specifies the x-coordinate of the upper left corner of the widget.
y	Specifies the y-coordinate of the upper left corner of the widget.
cols	Specifies the initial width of the widget measured in character spaces.
rows	Specifies the initial height of the widget measured by the height of a character.
value ¹	Specifies the text content of the widget.

¹In the high-level routine S TEXT, the argument to access this attribute is named **string_value**.

When you create a text widget, you can specify aspects of the initial appearance of the widget by assigning values to widget attributes. For example, the simple text widget used in the user interface of the DECburger sample application defines the initial text that appears in the widget, specifies the initial position of the widget by its x-coordinate and y-coordinate, specifies the width of the widget in pixels, and restricts the number of characters that a user can enter. Example 9-1 shows an excerpt from the DECburger UIL module in which the simple text widget is defined.

Example 9-1 Defining the Simple Text Widget in the DECburger Sample Application

```
.
.
.
object
  fries_quantity : simple_text {
    arguments {
      x = 165;
      y = 100;
      width = 30;
      max_length = 3;
      simple_text_value = k_0_label_text;
    };
    callbacks {
      create = procedure create_proc (k_fries_quantity);
    };
  };
.
.
.
```

In the example, the **value** attribute (called **simple_text_value** in UIL) receives the initial value 0, defined as the constant *k_0_label_text*. The DECburger UIL module uses constants to represent all text strings. These constants are defined at the beginning of the DECburger UIL module.

The following sections describe how to use the attributes of the text widgets and the text widget support routines.

Handling Text

9.2 Creating Text Widgets

9.2.1 --- Manipulating the Text Contents of the Text Widgets

The text widgets provide text entry and text editing capabilities in a user interface. To manipulate the text content of the text widgets at run time (after the widget has been created), you can use either of the following two methods:

- The SET VALUES or GET VALUES intrinsic routine
- The text widget support routines

The support routines offer several advantages over the SET VALUES or GET VALUES intrinsic routines:

- The support routines use fewer system resources and, therefore, are more efficient.
- The support routines do not require that you create an argument list.

--- 9.2.1.1 Placing Text in a Text Widget

To place text in a text widget after the widget has been created, you can use the SET VALUES intrinsic routine or a support routine.

To use the SET VALUES intrinsic routine, specify the address of the text string or compound string as the value of the **value** attribute in an argument list. Then pass this argument list to the SET VALUES intrinsic routine to assign the value to the widget attribute.

Using the text widget support routines, you can either modify the text the widget contains or replace the text entirely.

To replace all the text in a simple text widget, use the S TEXT SET STRING support routine. Use the CS TEXT SET STRING routine to replace all the text in a compound string text widget. Both support routines place the address of the new text in the **value** attribute.

To modify the text currently in the simple text widget, use the S TEXT REPLACE support routine. Use the CS TEXT REPLACE support routine to modify the text currently in the compound string text widget. Both routines take the following arguments:

- The identifier of the text widget
- The position in the text where the text to be replaced begins
- The position in the text where the text to be replaced ends
- The new text that you want to put in place of the existing text

Specify the position in the text as an offset from the beginning. Determine the offset by counting the characters, including spaces. The first character is numbered 0 (zero). Successive characters are numbered sequentially.

To insert text, specify the same position for both the start and the end points. If the start and end points are not specified as the same position, the text in the section defined by the start and end points is replaced by the new text.

9.2.1.2 Retrieving Text from a Text Widget

To retrieve the current text content of a text widget, you can use the GET VALUES intrinsic routine or a support routine.

To use the GET VALUES intrinsic routine, create a variable to hold the address of the text string or compound string and specify this variable as the value of the **value** attribute in an argument list. Then pass this argument list to the GET VALUES intrinsic routine. The GET VALUES intrinsic routine writes the address contained in the **value** attribute into the variable that you specified in the argument list.

Use the S TEXT GET STRING support routine to retrieve the current text content of the simple text widget. For the compound string text widget, use the CS TEXT GET STRING support routine to retrieve its text content. These support routines return the value of the **value** attribute.

Example 9-2 shows how the DECburger sample application uses the S TEXT GET STRING support routine to retrieve the current value of a simple text widget. The DECburger user interface uses a simple text widget to solicit information about quantity from the user. In the example, the S TEXT GET STRING support routine returns the text contained in the simple text widget. (The variable *fries_text* holds this string.) The example goes on to convert the text into an integer so that DECburger can manipulate the value.

Example 9-2 Using the S TEXT GET STRING Support Routine in the DECburger Sample Application

```
.  
. .  
fries_text = DwtSTextGetString(widget_array[k_fries_quantity]);  
fries_num = 0;  
sscanf(fries_text, "%d", &fries_num);  
. .  
.
```

9.2.1.3 Disabling Text Editing

By default, users can edit the text contained in the text widgets. However, you can disable text editing by setting the **editable** attribute to false (this attribute is true by default). To change this value after the widget has been created, use the SET VALUES intrinsic routine or a support routine.

To set the value of the **editable** attribute in the simple text widget, use the S TEXT SET EDITABLE support routine. In the compound string text widget, use the CS TEXT SET EDITABLE support routine to set this attribute.

To read the value of the **editable** attribute in the simple text widget, use the S TEXT GET EDITABLE support routine. In the compound string text widget, use the CS TEXT GET EDITABLE support routine to read this attribute.

Handling Text

9.2 Creating Text Widgets

9.2.1.4 Limiting the Length of the Text

You can specify the maximum amount of text that the user can enter in the text widgets by using the **max_len** attribute. To assign a value to this attribute at run time, use the SET VALUES intrinsic routine or the S TEXT SET MAX LENGTH support routine. To read the value of this attribute at run time, use the GET VALUES intrinsic routine or the S TEXT GET MAX LENGTH support routine. Use the CS TEXT GET MAX LENGTH and the CS TEXT SET MAX LENGTH support routines to read and set this attribute in the compound string text widget.

9.2.2 Customizing the Appearance of the Text Widgets

You can customize the following aspects of the appearance and function of the text widgets:

- Size
- Margins
- Resizing behavior
- Text cursor appearance
- Position of the insertion point
- Border visibility and color

The compound string text widget supports additional attributes that are not supported by the simple text widget. These additional attributes enable you to do the following:

- Identify the current writing direction
- Identify the current editing direction
- Specify the text cursor shape

Section 9.2.2.7 describes these attributes.

9.2.2.1 Specifying Size

To specify the dimensions of the text widgets, use the **cols** and **rows** attributes. These attributes specify the size of the widget in relation to the size of the characters they contain, which is determined by the fonts used to display the characters.

Use the **cols** attribute to specify the width of the text widgets. (Each character width is referred to as a column.) With this attribute, you can specify the width by the number of characters that the widget can contain horizontally.

Use the **rows** attribute to specify the height of the text widgets. The height of each row is determined by the height of a character. The overall height dimension of the simple text widget is determined by the number of rows that you specify in the **rows** attribute.

Handling Text

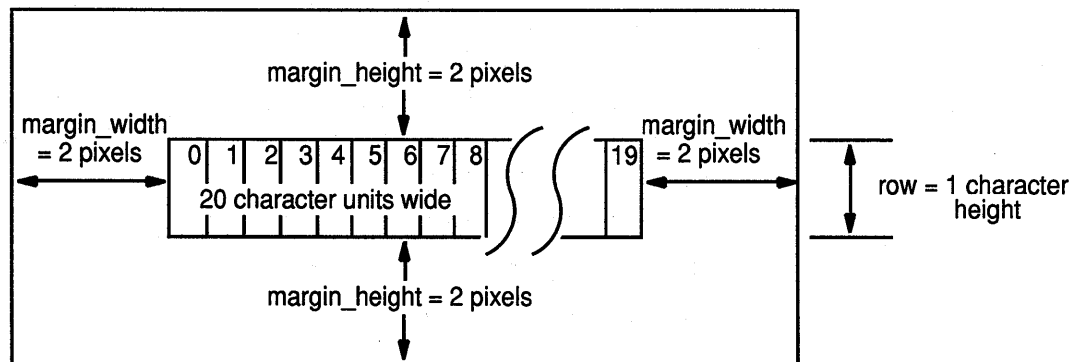
9.2 Creating Text Widgets

The exact measurement in pixels of these two dimensions depends on the font being used. In the XUI Toolkit, fonts are specified in font lists. (For more information about specifying fonts in font lists, see Section 5.6.5.) The simple text widget, which can only use a single font, uses the dimensional values from the first font specified in the font list as the unit of the **cols** and **rows** attributes. The compound string text widget, which can use as many fonts as are specified in the font list, uses the maximum dimensional values from all of the specified fonts as the unit of the **cols** and **rows** attributes.

While you can specify the size of the text widgets in pixels by using the common widget attributes **width** and **height**, this method is not recommended. Fixing the size of the widget in this way creates a dependency on the font. The size that you specify may work well with a particular font, but if the font size is increased, the text characters may no longer fit inside the widget.

Figure 9-2 illustrates where each of these attributes appears in the text widgets. As shown, the default values for these attributes are 20 columns wide and 1 row high.

Figure 9-2 Default Configuration of the Text Widgets



ZK-0398A-GE

9.2.2.2 Specifying Margins

You can specify the amount of space around the text entry area of the text widgets by using the **margin_width** and **margin_height** attributes.

Use the **margin_width** attribute to specify the amount of space between the border of the widget and the beginning of the array of characters. (The length of the text determines the amount of space between the end of the text and the border.) Specify this margin in pixels.

Use the **margin_height** attribute to specify the amount of space between the top and bottom borders of the widget and the top and bottom edges of the text entry area. Specify this margin in pixels.

Figure 9-2 illustrates where each of these attributes appears in the text widgets. As shown, the default values for these attributes are 2 pixels for each margin.

Handling Text

9.2 Creating Text Widgets

9.2.2.3 Controlling Resizing Behavior

Although you might specify the dimensions of the text widgets, if the user enters more text than will fit in the widget, the text widgets attempt to expand to fit the text. Using the attributes of the text widgets, you can control this behavior in the following ways:

- Setting the resize attributes to false
- Making the text wrap
- Including a scroll bar in the simple text widget

You can turn off the automatic resizing behavior of the text widgets by using the **resize_height**, **resize_width**, or **word_wrap** attributes.

To prevent the text widgets from increasing their height, set the **resize_height** attribute to false. To prevent the text widgets from increasing their width, set the **resize_width** attribute to false.

The **direction_rtol** attribute determines in which direction the text widgets expand. For example, if the writing direction is right-to-left, the widgets attempt to expand to the left, keeping the rightmost column fixed in its place.

If you fix the width of a text widget, you can specify that the text widget wrap words that would otherwise extend beyond the right edge of the widget onto the next line by setting the **word_wrap** attribute to true.

Another way to control resizing is by including a vertical scroll bar in the text widget. If you include a vertical scroll bar in a text widget, the widget will not resize its height to fit additional text. The scroll bar enables the user to scroll through text that is not currently visible.

To include a vertical scroll bar in a text widget, set the **scroll_vertical** attribute to true. By default, the vertical scroll bar appears on the right side of the widget, but you can make the scroll bar appear on the left side of the widget by setting the **scroll_left_side** attribute to true.

9.2.2.4 Controlling Text Cursor Appearance

When the text widget has input focus, its text cursor blinks. By assigning values to text widget attributes, you can specify the following:

- How fast the text cursor blinks
- Whether the text cursor is visible

Use the **blink_rate** attribute to specify how fast the text cursor should blink. Specify this value in milliseconds.

Use the **insertion_point_visible** attribute to determine whether or not the text cursor is visible in the widget. (The text cursor is visible when it is drawn in the foreground color.) Set this value to true if you want the text cursor to be visible.

Note that the **insertion_point_visible** argument specifies only whether the text cursor should be drawn in the foreground color. If the text cursor is positioned in a portion of the text that is not currently visible in the text widget, the text cursor will not be visible. To ensure that the text cursor

is always in the visible portion of the text widget, use the **auto_show_insert_position** attribute (described in Section 9.2.2.5).

In the compound string text widget, you can also specify whether the shape of the text cursor indicates the current editing direction. For more information about this topic, see Section 9.2.2.7.

9.2.2.5 Positioning the Insertion Point

Use the **insertion_position** attribute to position the text cursor within the text contents of the text widget. Specify the position of the insertion point as an offset from the beginning of the text string or compound string. Determine the offset by counting the number of characters in the string, including spaces. The first character in a string is numbered 0 (zero). Successive characters are numbered sequentially.

To specify that the insertion point should always be in the visible portion of the text widget, set the **auto_show_insert_position** to true. This causes the widget to scroll as the position of the insertion point changes, keeping the insertion point in the visible portion of the text.

9.2.2.6 Specifying Border Visibility and Color

According to the *XUI Style Guide*, the text widgets appear in a user interface by default as two perpendicular lines forming a right angle at the text entry area (see Figure 9–1). These lines are actually two of the borders of the widget. To create one of these widgets without a visible half-border, set the **half_border** attribute to false.

The **foreground** attribute determines the color of the text widget. Specify the pixel to be used as the value of this attribute.

9.2.2.7 Identifying the Current Writing and Editing Directions

You can identify the current writing and editing directions of the text contained in a compound string text widget by reading the value of the **text_path** and **editing_path** attributes. The simple text widget does not support these attributes.

The **text_path** attribute indicates the main writing direction of the text in the compound string text widget. The compound string text widget sets the value of the **text_path** attribute to the writing direction specified in the compound string that it contains when it is created. If this compound string has multiple segments, the compound string text widget uses the value of the first segment.

The **editing_path** attribute indicates the writing direction enabled for text entry and editing. For example, if the value of the **editing_path** attribute is left-to-right, the delete key deletes characters to the left of the insertion point. If the value is right-to-left, the delete key deletes the character to the right of the insertion point.

At widget creation time, the compound string text widget sets the value of the **editing_path** attribute to be the same as the value of the **text_path** attribute. However, the value of the **editing_path** attribute changes whenever a user changes the editing direction. (Users of your application can switch between the left-to-right and right-to-left editing directions by pressing the toggle key [F17].)

Handling Text

9.2 Creating Text Widgets

The compound string text widget can indicate the current editing direction by changing the shape of the cursor. To use this feature, set the **bidirectional_cursor** attribute to true. By default, the text cursor does not indicate editing direction.

9.2.3 Handling Text Selections

All applications running in the VMS DECwindows environment have access to a global selection facility. This facility allows users of applications to select portions of the display by moving the pointer cursor. Selected portions appear highlighted on the display. For more information about this facility, see the *XUI Style Guide*.

The text widgets support the selection mechanism. In these widgets, you can perform the following functions:

- Select text in the text widget
- Retrieve the selected text
- Cancel the current selection

9.2.3.1 Selecting Text

To select text in a simple text widget, use the S TEXT SET SELECTION support routine. Use the CS TEXT SET SELECTION support routine to select a portion of the text in a compound string text widget. Both routines take the following arguments:

- The widget identifier of the text widget
- The position in the text where you want to start the selection
- The position in the text where you want to end the selection
- The time of the event that led to the call to the selection

Section 9.2.1.1 describes how to determine positions in a text string or compound string.

You obtain the time stamp of the event that triggered the selection from the X Event data structure. (See Example 13-1 in Section 13.2 for an example of how to obtain the time stamp from the X Event data structure.)

If the currently selected text contains the insertion point, the selected text is deleted when new text is entered. You can specify that this selected text not be deleted by setting the **pending_delete** attribute to false. By default, this attribute is set to true.

9.2.3.2 Retrieving Selected Text

To retrieve the currently selected text in a simple text widget, use the S TEXT GET SELECTION support routine. Use the CS TEXT GET SELECTION support routine to retrieve the currently selected text in a compound string text widget. The selected text is returned as a text string by the simple text widget or as a compound string by the compound string text widget.

9.2.3.3 Canceling the Selection of Text

To cancel the selection of text in the simple text widget, use the S TEXT CLEAR SELECTION support routine. Use the CS TEXT CLEAR SELECTION support routine to cancel the selection of text in the compound string widget.

Both routines turn off the selected text highlighting.

9.2.4 Associating Callbacks with Text Widgets

When the text contained in a text widget changes, the widget uses the callback mechanism to notify your application. The text in the widget can change as the result of a user interaction, such as entering new text or editing existing text. Your program can also cause a callback by changing the text in a text widget using the SET VALUES intrinsic routine or a support routine.

In addition, the text widgets perform callbacks whenever they accept or lose input focus. To enable users to enter text by using the keyboard, the text widgets must have input focus. The text widgets get input focus when the user clicks MB1 anywhere within their borders.

The compound string text widget performs a callback if it cannot find in its font list the character set required to display a segment of text. (The simple text widget does not support this callback.) In this callback, the compound string text widget identifies the required character set for which there is no entry in the font list. The compound string text widget searches its font list a second time for the character set when the callback routine returns. If you update the compound string text widget's font list in the callback routine, the widget will find the character set in its font list and be able to display the text tagged with this character set. If you do not associate a callback routine with this callback reason, the compound string text widget does not perform the second search of the font list. The compound string text widget uses a checkerboard character in place of any character tagged with a character set it cannot find in the font list.

For complete information about the data returned in the callbacks performed by the text widgets, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a text widget, pass a callback routine list to one of the callback attributes. Table 9-4 lists the callback attributes and describes the conditions that trigger these callbacks.

Handling Text

9.2 Creating Text Widgets

Table 9-4 Text Widget Callbacks

Callback Attribute	Conditions for Callback
value_changed	The text contained in the text widget has changed. This callback can be triggered by a user interaction or because your application has changed the text in the widget using the SET VALUES intrinsic routine or one of the text widget support routines.
focus_callback	The text widget has accepted input focus.
lost_focus_callback	The text widget has lost input focus.
help_callback	A user has pressed the Help key while clicking MB1 in the text widget.
nofont_callback	The compound string text widget cannot find a character set in its font list that is needed to display the text in a compound string.

10 Using the Scale and the Scroll Bar Widgets

Both the scale and scroll bar widgets are graphical widgets that enable users to input data to your application using a pointing device, such as a mouse. This chapter provides the following:

- Overviews of the scale and scroll bar widgets in the XUI Toolkit
- A detailed description of how to use these widgets in an application user interface

10.1 Overview of the Scale Widget

The **scale widget** is a rectangular window containing a **scale** and a descriptive text label, called a **title**.

The scale contained in the scale widget is an elongated rectangle that represents a range of values. Users of your application can select a value from the range by moving an indicator, called a **slider**, inside the scale. Users can also select a value by moving the pointer cursor anywhere in the scale and clicking MB1. The position of the slider corresponds to the current value of the scale. Use the scale widget to solicit input from users of your application. Specify the range of values represented by the scale to ensure that the user can enter only legal values.

The title contained in the scale widget is descriptive text or graphics that labels the scale widget. In a scale widget with horizontal orientation, the title appears beneath the scale and aligned with the left end of the scale. In a scale widget with vertical orientation, the title appears to the left of the scale and aligned with the top of the scale.

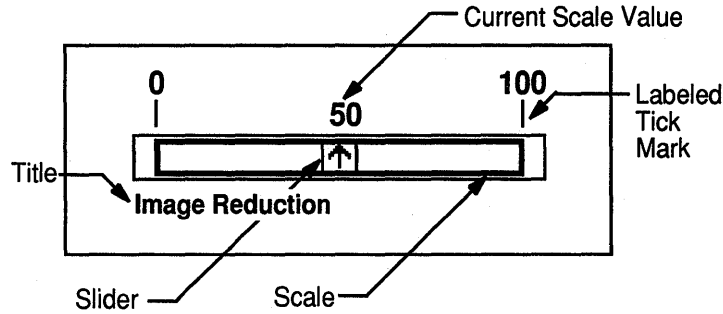
You can optionally mark off points along the range of the scale by including labeled **tick marks**. Section 10.2.2.6 describes how to add tick marks to a scale.

Figure 10–1 shows a scale widget and its components. In the figure, the border of the scale widget is visible. By default, the border of the scale widget does not appear on the display. However, you can make it visible by assigning a value to the **border_width** attribute.

Using the Scale and the Scroll Bar Widgets

10.1 Overview of the Scale Widget

Figure 10-1 Scale Widget



ZK-0402A-GE

10.2 Creating a Scale Widget

The main tasks involved in creating a scale widget are determining the range of values it represents and configuring its appearance. You can create children of a scale widget, but the children can only be used to implement tick mark labels. While you can use any widget as a tick mark label, typically you would use only a label widget.

To create a scale widget, perform the following steps:

1 Create the scale widget.

Use any of the widget creation mechanisms listed in Table 10-1. The choice of mechanism depends on the attributes you need to access.

Table 10-1 Scale Widget Creation Mechanisms

High-level routine	Use the SCALE routine to create a scale widget.
Low-level routine	Use the SCALE CREATE routine to create a scale widget.
UIL object type	Use the scale object type to define a scale widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

2 Create the children of the scale widget.

If you want to include labeled tick marks on your scale, you must create the labels as children of the scale widget. Section 10.2.2.6 provides more information about this procedure.

3 Manage the children of the scale widget.

Use the intrinsic routine MANAGE CHILD to manage a single child of the scale widget. Use MANAGE CHILDREN to manage a group of children.

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

4 Manage the scale widget.

Use the intrinsic routine `MANAGE CHILD` to manage the scale widget.

After you complete these steps, if the parent of the scale widget has been realized, the scale widget will appear on the display.

Low-level routines and `UIL` provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these widget attributes at widget creation time. (To access attributes not available with a high-level routine, use the `SET VALUES` intrinsic routine after the widget has been created.) The *VMS DECwindows Toolkit Routines Reference Manual* lists the complete set of attributes supported by the scale widget. Table 10–2 lists the attributes you can set if you use the high-level routine `SCALE` to create a scale widget. Pass the values of these attributes as arguments to the routine.

Table 10–2 Attributes Accessible Using the High-Level Routine `SCALE`

<code>x</code>	The x-coordinate of the upper left corner of the widget
<code>y</code>	The y-coordinate of the upper left corner of the widget
<code>width</code>	The horizontal dimension of the widget
<code>height</code>	The vertical dimension of the widget
<code>scale_width</code>	The horizontal dimension of the scale
<code>scale_height</code>	The vertical dimension of the scale
<code>title</code>	The text string used as title of the scale
<code>min_value</code>	The minimum value represented by the scale
<code>max_value</code>	The maximum value represented by the scale
<code>decimal_points</code>	The placement of decimal point in value labels
<code>value</code>	The current value of the scale
<code>orientation</code>	The orientation of the scale (horizontal or vertical)
<code>callback</code>	The address of a callback routine list
<code>drag_callback</code>	The address of a callback routine list
<code>help_callback</code>	The address of a callback routine list

10.2.1 Determining the Range of Values

You can determine the range of values represented by the scale in a scale widget. By default, the range is from 0 to 100. You can choose any integer value.

Specify the minimum value in the **`min_value`** attribute. This value will be the top of a vertical scale and the left of a horizontal scale. Specify the maximum value in the **`max_value`** attribute. This value is represented by the bottom or the right end of the scale.

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

The scale widget always maintains its current value in the **value** attribute. The position of the slider in the scale represents the current value. You can create the scale widget with a default value by assigning a value to this attribute. To change the value of the scale widget after it has been created, use the SET VALUES intrinsic routine.

Example 10–1 shows the scale widget definition from the DECburger UIL module. In it, the scale is configured with a minimum value of 0 and a maximum value of 10.

Example 10–1 Determining the Range of Values

```
object
  burger_quantity : scale {
    arguments {
      x = 25;
      y = 85;
      min_value = 0;
      max_value = 10;
      width = 70;
      border_width = 0;
      title = k_quantity_label_text;
    };
    callbacks {
      create = procedure create_proc (k_burger_quantity);
      value_changed = procedure scale_proc (k_burger_quantity);
    };
  };
};
```

10.2.2 Customizing the Appearance of a Scale Widget

The attributes of the scale widget enable you to customize the following aspects of its appearance:

- Size
- Orientation
- Title
- Appearance of the slider
- Representation of scale value
- Placement of tick marks along the scale

10.2.2.1 Specifying the Size of a Scale Widget

Specify the size of a scale widget using the common widget attributes **width** and **height**. Specify these dimensions in pixels.

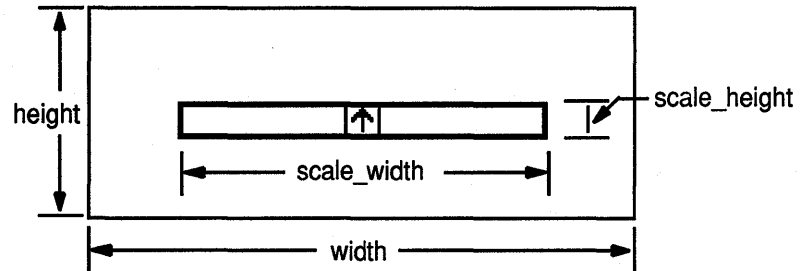
You can also specify the size of the scale contained in the scale widget. Specify the dimensions of the scale by assigning values to the **scale_width** and the **scale_height** attributes. Note that if you specify the size of the scale larger than the size of the scale widget that contains it, the scale will be clipped.

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

Figure 10-2 illustrates the dimensions these attributes affect.

Figure 10-2 Scale Widget Sizing Attributes



ZK-0401A-GE

You do not need to specify any of these dimension attributes. The scale widget can calculate its size to accommodate the space requirement of its children.

10.2.2.2 Specifying the Orientation of the Scale Widget

You can specify the orientation of the scale widget by setting the **orientation** attribute. Specify the value of this attribute using one of the constants listed in Table 10-3.

Table 10-3 Horizontal and Vertical Orientation Constants

Orientation	MIT C Binding	VAX Binding
Horizontal	DwtHorizontal	DWT\$C_HORIZONTAL
Vertical	DwtVertical	DWT\$C_VERTICAL

10.2.2.3 Specifying the Title of the Scale Widget

You can include a title, in the form of a compound string, in your scale widget to describe its function to users. To specify a title, create the title as a text string, convert the text string into a compound string, and pass the address of the compound string to the scale widget in the **title** attribute.

Example 10-2 shows how the title, width, position, and other aspects of the scale widget in the DECburger application are specified.

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

Example 10–2 Setting Appearance Attributes of the Scale Widget in the DECburger Sample Application

```
object
  burger_quantity : scale {
    arguments {
      x = 25;
      y = 85;
      min_value = 0;
      max_value = 10;
      ❶ width = 70;
      border_width = 0;
      ❷ title = k_quantity_label_text;
    };
    callbacks {
      create = procedure create_proc (k_burger_quantity);
      value_changed = procedure scale_proc (k_burger_quantity);
    };
  };
};
```

- ❶ DECburger specifies the horizontal dimension of the scale widget by assigning a value to the **width** attribute.
- ❷ DECburger specifies the text used as the title of the scale widget by assigning the constant *k_quantity_label_text* as the value of the **title** attribute. DECburger defines constants for all the text strings used in its interface at the beginning of the UIL module. This makes changing the text of a label easy to manage.

10.2.2.4 Specifying the Color of the Slider

You can specify the color used for the slider. When your application runs on a color workstation, the slider will appear in the color you specify. When your application runs on a workstation with a black and white monitor, the slider will appear as a shade of gray.

Specify the color by passing the color pixel in the **slider_pixel** attribute.

10.2.2.5 Representing the Value of the Scale

By default, the scale widget displays its current value above the scale over the slider position, if the orientation of the scale is horizontal. If the orientation of the scale widget is vertical, the widget displays its current value to the right of the scale, opposite the slider position. The value is displayed as a decimal number. You can specify that the current value not appear in the scale widget by setting the **show_value** attribute to false (the default is true).

The scale widget always maintains its value as an integer. However, you can specify that the value be presented on the display with a decimal point. Use the **decimal_points** attribute to specify the number of positions you want displayed to the right of the decimal point. Note that, to the scale widget, the value is still an integer.

For example, to represent a value with two places to the right of the decimal point, specify the **decimal_points** attribute as 2. If the scale range was 0 to 10,000, the scale widget would represent the current value as 0.00 to 100.00. The value returned by the scale widget is always an integer between the minimum and maximum values.

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

10.2.2.6 Adding Labeled Tick Marks to a Scale Widget

You can mark off points along the scale contained in a scale widget by adding labeled tick marks. You create the labeled tick marks by creating children of the scale widget. Each child represents a point along the scale. You only need to specify the tick mark labels (typically label widgets). The scale widget positions the children symmetrically along the scale and draws the actual tick marks at each position on the scale.

For example, if you create two widgets as children of the scale widget, the scale widget positions the children at either end of the scale. If you create three or more children, the scale widget spaces their positions evenly across the scale.

For scale widgets with vertical orientation, the tick marks appear on the right side of the scale; for scale widgets with horizontal orientation, the tick marks appear above the scale. See Figure 10–1 for an illustration.

Example 10–3 shows a code fragment that creates a scale widget with two labeled tick marks.

Example 10–3 Labeling Points Along a Scale in a Scale Widget

```
build_scale()
{
    WidgetList scale_marks[] = NULL;
    int count = 0;

    ❶ scale = DwtScale( parent_widget,
        "Image Reduction", /* will appear as title */
        0,0, /* x and y */
        0,0, /* width and height */
        0,0, /* scale width and height */
        DwtLatin1String("Sample Scale"), /* title */
        0,100, /* minimum and maximum values */
        0, /* no decimal points */
        50, /* default value */
        DwtHorizontal, /* orientation */
        callback_list, /* on change of value, do this */
        NULL,NULL); /* no drag or help callbacks */

    ❷ scale_marks[count++] = DwtLabel( scale,
        "label_min",
        0,0, /* x and y */
        DwtLatin1String("0"), /* label */
        NULL); /* no help callback */

    ❸ scale_marks[count++] = DwtLabel( scale,
        "label_max",
        0,0,
        DwtLatin1String("100"),
        NULL);

    ❹ XtManageChildren( scale_marks, count );

    ❺ XtManageChild( scale );
}
```

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

- ① In this statement, the example creates the scale widget using the high-level routine `SCALE`. The example configures the scale widget by passing attribute values as arguments to the routine. In the example, the orientation of the scale widget is specified as horizontal. The scale contained in the scale widget is specified with a minimum value of 0 and a maximum value of 100. The initial value of the scale is specified as 50.
- ② After creating the scale widget, the example creates the two label widgets that will be the labeled tick marks on the scale. In this statement, the example creates a label widget using the high-level routine `LABEL`. The example specifies the text string that will appear in the label: the text string "0". The scale widget positions its children in the order they are managed. This label will appear at the extreme left of the scale.
- ③ The example creates a second label widget as a child of the scale widget. This label widget will contain the text string "100". This label will appear at the extreme right of the scale.
- ④ The example manages both children of the scale widget using the intrinsic routine `MANAGE CHILDREN`.
- ⑤ The example then manages the scale widget itself using the intrinsic routine `MANAGE CHILD`.

10.2.3 Associating Callbacks with a Scale Widget

When the value of the scale is changed, the scale widget uses the callback mechanism to notify your application. The change can be due to a user interaction, such as moving the slider, or because your application changed the value using the `SET VALUES` intrinsic routine.

In addition, the scale widget also uses the callback mechanism to notify your application of the intermediate values of the scale when the slider is in motion.

When the scale widget performs a callback, it returns the new value in its callback data along with other data. For complete information about the data returned in a scale widget callback, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a scale widget, pass a callback routine list to one of the scale widget callback attributes. The scale widget supports two callbacks, depending on the type of user interaction. Table 10-4 lists the callback attributes and the conditions that trigger them.

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

Table 10-4 Scale Widget Callbacks

Callback Attribute	Description
drag_callback	A user has pressed MB1 on the slider and is moving the slider across the scale.
value_changed_callback	The value of the scale has changed because a user has moved the slider or clicked MB1 in the scale, or your application changed the contents of the value attribute using SET VALUES.

In Example 10-4, DECburger associates a callback routine with the **value_changed_callback** attribute of the scale widget it uses in its interface. DECburger does not use the **drag_callback** attribute.

Example 10-4 Associating a Callback Routine with a Scale Widget

```
object
  burger_quantity : scale {
    arguments {
      x = 25;
      y = 85;
      min_value = 0;
      max_value = 10;
      width = 70;
      border_width = 0;
      title = k_quantity_label_text;
    };
    callbacks {
      create = procedure create_proc (k_burger_quantity);
      value_changed = procedure scale_proc (k_burger_quantity);
    };
  };
```

Example 10-5 shows the callback routine DECburger uses with the scale widget. The callback routine assigns the value returned by the scale widget in the scale callback data structure to an array of quantity variables. This information is then used to complete the total order.

Using the Scale and the Scroll Bar Widgets

10.2 Creating a Scale Widget

Example 10–5 Scale Widget Callback Routine in the DECburger Application

```
static void scale_proc(w, tag, scale)
Widget w;
int *tag;
DwtScaleCallbackStruct *scale;
{
    quantity_vector[k_burger_index] = scale->value;
}
.
```

10.3 Overview of the Scroll Bar Widget

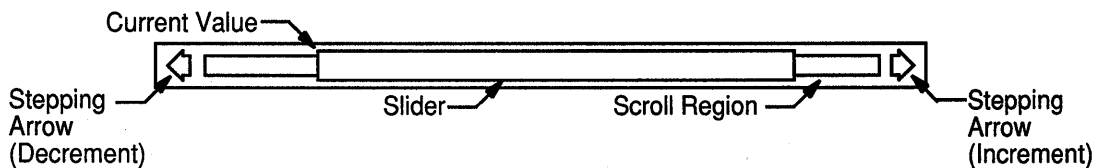
A scroll bar widget is a rectangular window containing an elongated rectangular area, called the **scroll region**, that is sensitive to user input using a mouse button. Overlaying the scroll region is the slider. The position of the slider indicates the current value of the scroll bar widget. (The top of the slider represents the current value in vertical scroll bar widgets; the left side of the slider represents the current value in horizontal scroll bar widgets.) The slider also provides, by its size, a visual representation of the portion of the work area that is visible.

In addition, a scroll bar includes two arrow-shaped, push buttons widgets, called **stepping arrows**, that appear at both ends of the scroll region. The stepping arrows allow the user to move in increments through the work area. You can determine the amount of increment that one MB1 click on a stepping arrow causes. For example, for a spreadsheet application, you might choose a row or column as the amount of increment or decrement.

The scroll bar widget includes additional input capabilities that allow users to request that a position in the work area be moved to the top or bottom of the visible portion of the work area.

Figure 10–3 shows a scroll bar widget and its components.

Figure 10–3 Scroll Bar Widget



ZK-0403A-GE

10.4 Creating a Scroll Bar Widget

The primary task involved in creating a scroll bar widget is determining the range of values it represents, determining the size of the slider, and creating the callback routines that will perform the actual scrolling operations.

The scroll bar widget is specially designed to allow users to indicate what portion of the work area they want to be visible. Note, however, that the scroll bar widget only returns data to your application; it does not perform the scrolling of the work area. Your application must interpret the data returned by the scroll bar widget and adjust the visible portion of the work area to satisfy the user request. Your application must also update the size of the slider when you change the visible portion of the work area. (The scroll window widget updates the size and position of the slider automatically. For more information, see Section 4.4.)

To create a scroll bar widget, perform the following steps:

1 Create the scroll bar widget.

Use any of the widget creation mechanisms listed in Table 10–5. The choice of mechanism depends on the attributes of the scroll bar widget you need to access.

Table 10–5 Scroll Bar Widget Creation Mechanisms

High-level routine	Use the SCROLL BAR routine to create a scroll bar widget.
Low-level routine	Use the SCROLL BAR CREATE routine to create a scroll bar widget.
UIL object type	Use the scroll_bar object type to define a scroll widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

2 Manage the scroll bar widget.

Use the intrinsic routine MANAGE CHILD to manage the scroll bar widget.

After completing these steps, if the parent of the scroll bar widget has been realized, the scroll bar widget will appear on the display.

Low-level routines and UIL provide access to the complete set of attributes at widget creation time. High-level routines provide access to only a subset of these attributes at widget creation time. (To access widget attributes not available using the high-level routine, use the SET VALUES intrinsic routine after the widget has been created.) The *VMS DECwindows Toolkit Routines Reference Manual* lists the complete set of widget attributes. Table 10–6 lists the attributes you can set if you use the high-level routine SCROLL BAR to create a scroll bar widget. Pass the values of these attributes as arguments to the routine.

Using the Scale and the Scroll Bar Widgets

10.4 Creating a Scroll Bar Widget

Table 10–6 Attributes Accessible Using the High-Level Routine SCROLL BAR

x	The x-coordinate of the upper left corner of the widget
y	The y-coordinate of the upper left corner of the widget
width	The horizontal dimension of the widget
height	The vertical dimension of the widget
inc	The amount of unit increment in application-defined units
page_inc	The amount of page increment in application-defined units
shown	The size of the slider proportional to the amount of the work area visible
value	The current value of the scroll bar indicated by the top or the left of the slider
min_value	The minimum value represented by the scroll bar
max_value	The maximum value represented by the scroll bar
orientation	The orientation of the scroll bar (horizontal or vertical)
callback	The address of a callback routine list
help_callback	The address of a callback routine list
unit_inc_callback	The address of a callback routine list
unit_dec_callback	The address of a callback routine list
page_inc_callback	The address of a callback routine list
page_dec_callback	The address of a callback routine list
to_top_callback	The address of a callback routine list
to_bottom_callback	The address of a callback routine list
drag_callback	The address of a callback routine list

10.4.1 Determining the Range of a Scroll Bar Widget

You determine the range of values represented by the scroll bar widget. Specify the minimum value in the **min_value** attribute. This value will be the top of a vertical scroll bar widget and the left of a horizontal scroll bar widget. Specify the maximum value in the **max_value** attribute. This value is represented by the bottom or the right end of the scroll bar widget. Specify these values as integers. Your application determines what units the integers represent.

The **value** attribute always contains the current value of the scroll bar widget. In a scroll bar widget with a vertical orientation, the top edge of the slider represents the current value. In a scroll bar widget with horizontal orientation, the left edge of the slider represents the current value. You can specify the current value of a scroll bar widget when you create it by assigning a value to the **value** attribute. To change the value of this attribute after the widget has been created, use the SET VALUES intrinsic routine.

Using the Scale and the Scroll Bar Widgets

10.4 Creating a Scroll Bar Widget

Example 10–6 sets the minimum value of the scroll bar widget to 0 in the **min_value** attribute and the maximum value to 1000 in the **max_value** attribute.

Example 10–6 Specifying the Range of Values in a Scroll Bar Widget

```
object
  sample_scroll : scroll_bar {
    arguments {
      min_value = 0;
      max_value = 1000;
      shown = 200;
    };
    callbacks {
      value_changed = procedure scroll_proc();
    };
  };
```

10.4.2 Specifying the Size of the Slider in a Scroll Bar Widget

The size of the slider overlaying the scroll region in a scroll bar widget should represent the portion of the work area currently visible in the scrolling window. Use the **shown** attribute to specify the size of the slider. Specify the size as an integer. (If you use the scroll window widget, described in Section 4.4, the scroll window widget will update the size of the slider for you.) Your application is responsible for keeping the size of the slider current.

For example, if the work area widget is 1000 pixels in height, and the window onto the work area widget is 200 pixels in height, only one-fifth of the work area is visible. Calculate the size of the slider to represent this proportion. After initial setup, the size of the slider will not change unless the widget implementing the window onto the work area is resized or the size of the work area changes.

10.4.3 Defining the Size of Increment and Decrement

In addition to the slider, the scroll bar widget supports the following three mechanisms through which users can indicate movement of the window on the work area:

- Unit increment or decrement
- Page increment or decrement
- Movement of a position in the work area to the top or bottom of the window

The two stepping arrows included in the scroll bar widget implement the unit stepping functions. Each time a user clicks MB1 on a stepping arrow, your application should adjust the visible portion of the work area accordingly. You define the unit of increment or decrement within your application. By default, each click of a stepping area translates into increment or decrement of 10 units, but you can specify another value using the **inc** attribute.

Using the Scale and the Scroll Bar Widgets

10.4 Creating a Scroll Bar Widget

For example, you could specify in your application that a unit is one pixel. If you accept the default value of the **inc** attribute, you would have to move the work area 10 pixels for each click on an increment or decrement stepping arrow.

The page increment and decrement functions are activated when a user clicks MB1 in the scroll region between the slider and the stepping arrows. The area between the top or left of the slider and the top or left stepping arrow invokes the page decrement function. The area between the bottom or right of the slider and the bottom or right stepping arrow invokes the page increment function. As with the unit stepping functions, you define the size of the page increment or decrement within your application and specify the number of pages incremented or decremented in the **page_inc** attribute.

The unit and page increment and decrement functions automatically move the slider to represent the new position in the work area.

Users can indicate that they want to move a position in the work area to the top of the window by clicking MB2 anywhere in the scroll bar widget. They can indicate they want to move a position in the work area to the bottom of the window by clicking MB3 anywhere in the scroll bar widget. Note that these functions do not automatically set the slider position.

10.4.4 Modifying the Action of the Stepping Arrows

You can use the two translations attributes to modify the events that the increment and decrement stepping arrows respond to. Use **translations1** to change the decrement stepping arrows. Use the **translations2** attribute to change the increment stepping arrow. Specify the new action in a translation table, parse it, and pass the parsed translation table as the value of the translations argument. See Section 5.7 for more details about defining translations.

10.4.5 Customizing the Appearance of the Scroll Bar Widget

You can specify the size of a scroll bar widget using the **width** and **height** attributes. However, you should let the parent of the scroll bar widget determine its size. By default, the vertical scroll bar widget takes the height of its parent (minus 17 pixels) and a width of 17 pixels. A horizontal scroll bar widget takes the width of its parent (minus 17 pixels) and a height of 17 pixels.

As with the scale widget, you specify the orientation of the scroll bar in the **orientation** attribute. See Section 10.2.2.2 for more information about this topic. By default, the vertical scroll widget bar appears on the right side of its parent; the horizontal scroll bar widget appears on the bottom of its parent.

Using the Scale and the Scroll Bar Widgets

10.4 Creating a Scroll Bar Widget

10.4.6 Associating Callbacks with a Scroll Bar Widget

When the value of the scroll bar widget is changed, the widget uses the callback mechanism to notify your application. The change can be due to user interaction, such as moving the slider or activating a stepping arrow. You can also change the value of the scroll bar widget using the SET VALUES intrinsic routine.

When the scroll bar widget performs a callback, it returns data to your application in a callback data structure. In this data structure, the scroll bar widget returns the new value, along with other data. For complete information about the data returned in a scroll bar widget callback, see the *VMS DECwindows Toolkit Routines Reference Manual*.

To associate a callback routine with a scroll bar widget, pass a callback routine list to one of the scroll bar widget callback attributes. The scroll bar widget supports eight callbacks associated with the many types of user interaction it supports. In addition, the scroll bar widget supports the help callback. The *XUI Style Guide* describes the types of user interaction supported by the scroll bar widget. Table 10-7 summarizes these interactions and lists the scroll bar widget attributes you use to associate callback routines with them.

Table 10-7 Scroll Widget Callbacks

Callback Attribute	Description
unit_inc_callback	A user has clicked MB1 in the top stepping arrow, for vertical scroll bar widgets, or the left stepping arrow, for horizontal scroll bar widgets.
unit_dec_callback	A user has clicked MB1 in the bottom stepping arrow, for vertical scroll bar widgets, or the right stepping arrow, for horizontal scroll bar widgets.
page_inc_callback	A user has clicked MB1 in the scroll region above the slider.
page_dec_callback	A user has clicked MB1 in the scroll region below the slider.
to_top_callback	A user has clicked MB2 somewhere in the scroll bar widget, indicating the point in the work area that should be moved to the top of the window.
to_bottom_callback	A user has clicked MB3 somewhere in the scroll bar widget, indicating the point in the work area that should be moved to the bottom of the window.
drag_callback	A user has pressed MB1 on the slider and is moving the slider across the scroll bar widget.
value_changed_callback	The value of the scroll bar widget has changed because of a user interaction or because your application changed the contents of the value attribute using SET VALUES.

11 Using the Color Mixing Widget

This chapter provides the following:

- An overview of the color mixing widget in the XUI Toolkit
- A description of the support routines for the color mixing widget

In addition, the chapter describes how to modify the color mixing widget to support various color models.

11.1 Overview of the Color Mixing Widget

The color mixing widget enables users of your application to define colors according to either the HLS (Hue, Lightness, Saturation) color model or the RGB (Red, Green, Blue) color model. In addition, the color mixing widget provides users with immediate feedback, displaying each new color as it is defined. When the user activates the color mixing widget by pressing one of its push buttons, the color mixing widget performs a callback to your application, returning the RGB values of the newly defined color (as well as the initial color values).

Note that the color mixing widget returns only RGB values—your application is responsible for obtaining the color resources necessary to display the color. The color mixing widget uses RGB values to specify colors, regardless of the color model being supported, because the X Window System, Version 11, uses the RGB color model to specify colors. In the X Window System, Version 11, you specify the intensity of red, green, or blue as a value between 0 and 65,535.

By default, the color mixing widget supports the HLS and RGB color models (described in Section 11.1.1). However, you can customize the color mixing widget to support other color models (see Section 11.3).

Use the color mixing widget to provide users of your application with the ability to customize the colors used in your application. For example, if your application includes graphics, such as a pie chart, allow users to define the colors used in the pie chart by including the color mixing widget as an item in a customization menu.

11.1.1 Color Models

Color models are abstractions that enable unambiguous color specification. The color mixing widget supports the HLS and RGB color models.

In the HLS color model, a color is specified by three characteristics: hue, lightness, and saturation. Hue is color. Lightness describes the intensity of the color, that is, the amount of the color. Saturation describes the purity of the color or how much the color is diluted by white.

Using the Color Mixing Widget

11.1 Overview of the Color Mixing Widget

HLS expresses hue as a continuous spectrum of values arranged in a circular pattern. Red appears at 0 degrees (and again at 360 degrees), magenta is at 60 degrees, blue is at 120 degrees, cyan is at 180 degrees, green is at 240 degrees, and yellow is at 300 degrees. HLS expresses the lightness or intensity of a color as a percentage between 0 and 100 percent. One hundred percent lightness creates white light; zero percent lightness creates black. One oddity of the HLS color model is that full intensity colors are specified at 50 percent lightness. HLS expresses the saturation or purity of a color also as a percentage between 0 and 100 percent. One hundred percent saturation is a pure color. A zero-saturated color is a shade of gray, determined by the value of lightness.

In the RGB color model, a color is specified as a mixture of different intensities of red, green, and blue. In the X Window System, Version 11, you specify the intensity of red, green, or blue as a value between 0 and 65,535. Zero is the lowest intensity. Black is defined as a zero-intensity value for all three colors; white is 100 percent intensity for all three colors.

The color mixing widgets shown in Figure 11-1 and Figure 11-2 (in Section 11.1.2) illustrate how the color "sky blue" is specified in each color model. (The X Window System, Version 11, specifies a number of "named" colors, such as sky blue, by their RGB values. For a complete list of the colors named by the X Window System, see the *VMS DECwindows Xlib Programming Volume*.) In the HLS color model, sky blue is specified as 199 on the Hue scale, 49 percent lightness, and 60 percent saturation. In the RGB color model, sky blue is specified as a mixture of red at 12,800 intensity, green at 39,168 intensity, and blue at 52,224 intensity. Figure 11-2 illustrates how the scales in the color mixer subwidget express these X11 RGB values as percentages.

11.1.2 Components of the Color Mixing Widget

The color mixing widget is a pop-up dialog box that is preconfigured to contain the child widgets, called subwidgets, it needs to implement its functions. (When an XUI Toolkit widget contains other widgets, the widgets it contains are called subwidgets.) The color mixing widget contains the following subwidgets:

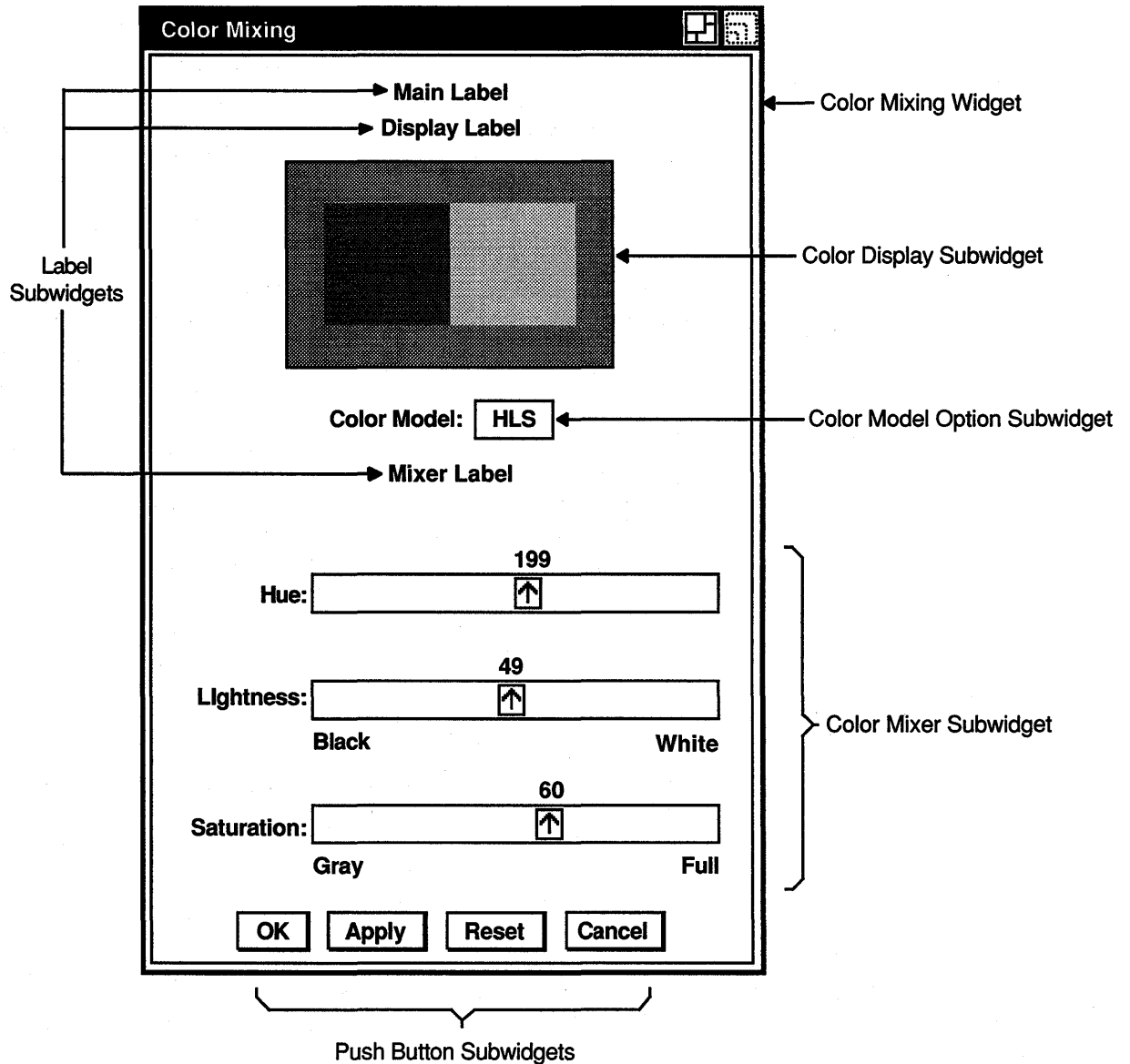
- Color display subwidget—displays the original color and the new color
- Color model option menu subwidget—implements choice of color model
- Color mixer subwidget—provides graphic tools with which users can define new colors
- Push button subwidgets—activate color mixing widget functions
- Label subwidgets—provide descriptive information
- Work area subwidget—supplies additional functions defined by application (optional)

Using the Color Mixing Widget

11.1 Overview of the Color Mixing Widget

Figure 11–1 shows these components in a color mixing widget with the HLS color model selected.

Figure 11–1 Components of the Color Mixing Widget (HLS Color Model)



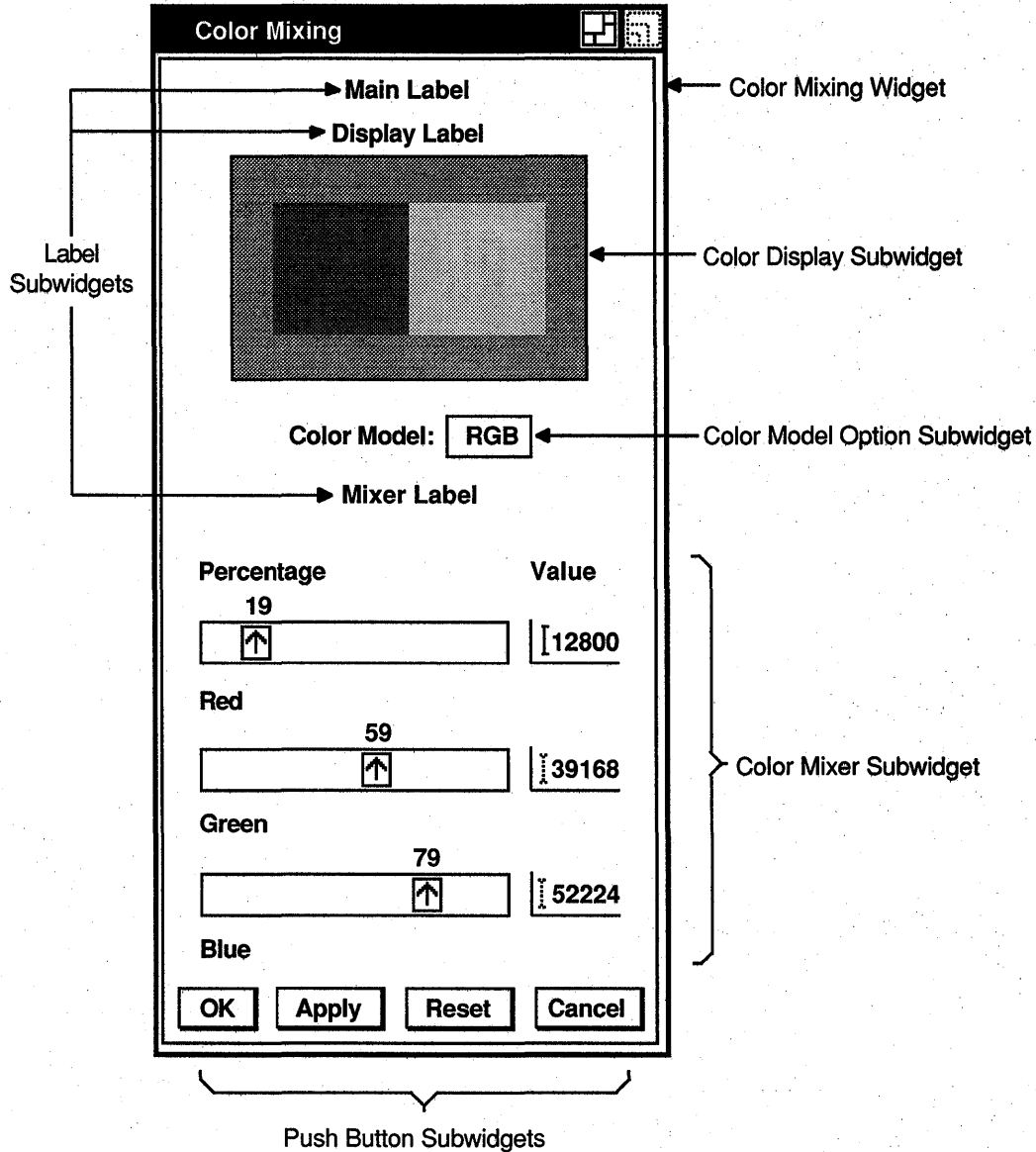
ZK-1888A-GE

Figure 11–2 shows these components in a color mixing widget with the RGB color model selected. The following sections describe these components.

Using the Color Mixing Widget

11.1 Overview of the Color Mixing Widget

Figure 11-2 Components of the Color Mixing Widget (RGB Color Model)



ZK-1277A-GE

11.1.2.1 Color Display Subwidget

The default color display subwidget is a dialog box widget (work area style) that contains two window widgets: one to display the original color and one to display the new color. The color displayed in the new-color window widget changes to represent the new color as it is defined. When you create the color mixing widget, you can specify the initial values of both the original color and the new color. If you do not specify an initial value for the new color, the color mixing widget sets it to match

Using the Color Mixing Widget

11.1 Overview of the Color Mixing Widget

the original color. You can also specify the background color of the color display subwidget (which is gray by default).

The color mixing widget allocates three color cells to represent the new color, the original color, and the background color. If the widget cannot allocate three color cells, it uses the cells it can allocate to represent the colors in the following order:

- 1 New color
- 2 Original color
- 3 Background color

When displayed on a gray scale device, the color display subwidget represents the color values as shades of gray. However, the color mixing widget does not display gray scale values on a color display device. On static gray and monochrome devices, the color display subwidget is not visible in the color mixing widget.

You can replace the default color display subwidget with a widget of your own design. For information about this topic, see Section 11.3.1.

11.1.2.2 Color Model Option Menu Subwidget

The color model option menu subwidget allows users of your application to choose the color model supported by the color mixer subwidget. (See Section 11.1.2.3 for information about the color mixer subwidget.) The two default color models appear as items in the option menu. Users can switch between color models at any time. When the color model is changed, the color mixing widget preserves the current color definition, translating the values that define the color in the current color model into values appropriate to the new color model.

The color model option menu subwidget appears in the color mixing widget only when the default color mixer subwidget is used.

11.1.2.3 Color Mixer Subwidget

The color mixer subwidget provides users with the graphic tools with which to define colors. When a user changes a value in the color mixer subwidget, the color mixing widget immediately updates the color displayed in the new-color window of the color display subwidget.

The default color mixer subwidget can support either the HLS or the RGB color models. You can specify which color model the color mixer subwidget supports initially by assigning a value to the **color_model** attribute. If you do not specify a color model, by default the color mixer subwidget initially supports the HLS color model. Use the constants listed in Table 11-1 to specify the color model in the **color_model** attribute.

Table 11-1 Color Model Constants

Color Model	Vax Binding Constant	C Binding Constant
HLS	DWT\$C_COLOR_MODEL_HLS	DwtColorModelHLS
RGB	DWT\$C_COLOR_MODEL_RGB	DwtColorModelRGB

Using the Color Mixing Widget

11.1 Overview of the Color Mixing Widget

To support the HLS color model, the color mixer subwidget contains three scales that represent the ranges of hue, lightness, and saturation. The hue scale presents color values as a range between 0 and 360. The lightness and saturation scales present their values as a range of percentages between 0 and 100.

To support the RGB color model, the color mixer subwidget contains three scales that represent the ranges of intensity of red, green, and blue. Each scale presents these color values as a percentage between 0 and 100. In addition, when supporting the RGB color model, the color mixer subwidget also contains text widgets in which users of your application can enter RGB values directly as text. The text widgets and the scales are linked: a change in one effects a corresponding change in the other.

You can replace the default color mixer subwidget with a widget of your own design. For information about this topic, see Section 11.3.2.

11.1.2.4 Push Button Subwidgets

By default, the color mixing widget contains four push button subwidgets labeled OK, Apply, Reset, and Cancel. When activated, the OK, Apply, and Cancel push buttons cause the color mixing widget to perform a callback to your application. (The Reset push button does not trigger a callback to your application because it has a built-in function that is internal to the color mixing widget. When activated, the Reset button changes the values in the color mixer subwidget and the color displayed in the new-color window of the color display subwidget back to their initial values.)

You implement the functions associated with the color mixing widget push buttons. The *XUI Style Guide* contains specific recommendations about what functions should be associated with push buttons containing labels such as OK, Apply, and Cancel. The following list restates these recommendations as they might be implemented with the color mixing widget:

- The OK push button makes the newly defined color appear in your application and then remove the color mixing widget from the display.
- The Apply push button makes the newly defined color appear in your application while the color mixing widget remains active on the display.
- The Cancel push button removes the color mixing widget from the display without implementing any of the changes a user might have made.

You implement as callback routines the functions you want associated with these push buttons. You associate these callback routines with the callback attributes of the color mixing widget. For example, to associate a function with the OK push button, use the **activate_callback** attribute. (For more information about associating callback routines with the color mixing widget, see Section 11.4.)

Note that you can change the text displayed in the push button subwidgets (see Section 11.2.2.3 for more information). You can also remove any of the push button subwidgets by specifying a null value for the text label.

Using the Color Mixing Widget

11.1 Overview of the Color Mixing Widget

11.1.2.5 Label Subwidgets

The color mixing widget contains more than a dozen labels that you can use to provide descriptive text for the components of the color mixing widget. Section 11.2.2.3 describes how to specify text for these labels.

11.1.2.6 Work Area Subwidget

The color mixing widget can contain a work area subwidget, if your application supplies one. The color mixing widget manages this subwidget and positions it below the color mixer subwidget and above the push button subwidgets.

The work area subwidget can be any other XUI Toolkit widget, such as a label, push button, or dialog box widget. If you use a dialog box widget, use only the work area style of this widget.

For example, your application can use this additional subwidget to include additional push button widgets to extend the functions of the color mixing widget.

11.2 Creating a Color Mixing Widget

To create a color mixing widget, perform the following steps:

- 1 Create the color mixing widget using any of the widget creation mechanisms listed in Table 11–2.

Note that there is no high-level mechanism for creating a color mixing widget.

Table 11–2 Mechanisms for Creating the Color Mixing Widget

Mechanism	Routine Name or Object Type
Low-level routine	Use the COLOR MIX CREATE routine to create a color mixing widget.
UIL object type	Use the UIL object type color_mix to define a color mixing widget in a UIL module. At run time, the DRM routine FETCH WIDGET creates the widget according to this definition.

- 2 Manage the color mixing widget using the intrinsic routine MANAGE CHILD.

After completing these steps, if the parent of the color mixing widget has been realized, the color mixing widget appears on the display.

As an illustration, Example 11–1 creates the color mixing widget shown in Figure 11–1. The example defines a color mixing widget that uses the default color display subwidget and the default color mixer subwidget. Because no color model is specified, the color mixing widget created uses the HLS color model, by default. Note that, in the example, the initial values for the original color are specified as RGB values even though the color model supported is HLS. You always use RGB values to specify colors in the color mixing widget and the color mixing widget always returns RGB values, regardless of the color model supported.

Using the Color Mixing Widget

11.2 Creating a Color Mixing Widget

Example 11-1 Creating a Color Mixing Widget

```
object
  ❶ color_input : color_mix {
    arguments {
      ❷ main_label      = sample_colormix_text;
        display_label  = default_color_display_text;
        mixer_label    = default_color_mixer_text;
      ❸ orig_red_value  = 12800;
        orig_green_value = 39168;
        orig_blue_value = 52224;
    };
    ❹ callbacks {
      ok      = procedure ok_proc();
      apply  = procedure apply_proc();
      cancel = procedure cancel_proc();
    };
  };
```

- ❶ The object declaration defines a color mixing widget named *color_input*. The UIL keyword for the color mixing widget is *color_mix*.
- ❷ The argument list section of the UIL object declaration assigns initial values to attributes of the color mixing widget. The first three statements define the text contents of the label subwidgets. In this UIL module, all text strings are defined as constants.
- ❸ The arguments list section also contains initial values for the original-color window in the color display subwidget. By default, the new color will appear as the same color.
- ❹ The callbacks list section of the UIL object declaration assigns values to each of the primary callbacks performed by the color mixing widget.

11.2.1 Setting and Retrieving New Color Values

To set or retrieve the RGB values of the new color displayed in the color display subwidget, you can use the SET VALUES and GET VALUES intrinsic routines. However, the XUI Toolkit provides support routines that allow you to perform these tasks much faster.

To set the values of the **new_red_value**, **new_green_value**, and **new_blue_value** attributes, use the COLOR MIX SET NEW COLOR support routine. You specify the values of these attributes as arguments to the routine. The default color display subwidget updates the new-color window to represent the newly defined color.

To retrieve the value of the new-color attributes, use the COLOR MIX GET NEW COLOR support routine. This support routine writes the current values of the **new_red_value**, **new_green_value**, and **new_blue_value** attributes into variables that you pass as arguments to the routine.

Using the Color Mixing Widget

11.2 Creating a Color Mixing Widget

Table 11-3 summarizes the support routines for the color mixing widget.

Table 11-3 Support Routines for the Color Mixing Widget

Routine	Description
COLOR MIX GET NEW COLOR	Retrieves the current values of the new-color attributes.
COLOR MIX SET NEW COLOR	Assigns values to the new-color attributes.

11.2.2 Customizing the Color Mixing Widget

You can customize the following aspects of the appearance and function of the color mixing widget:

- Size
- Margins
- Labels
- Background color
- Work area subwidget

11.2.2.1 Specifying the Size

The color mixing widget sizes itself to fit the subwidgets that it contains. For example, if you specify long compound strings as values for the label subwidgets, the color mixing widget increases its size to accommodate the labels. (You do not need to set the common widget attributes **width** and **height** to 0 [zero] to get the default size.)

In the default color display subwidget, you can specify the size of the windows in which the original and new colors are displayed. By default, each of these windows is 80 pixels square. Use the **display_col_win_width** attribute and the **display_col_win_height** attribute to specify the dimensions of these windows. Specify these dimensions in pixels. These attributes affect only the default color display subwidget.

11.2.2.2 Specifying Margins

You can specify the amount of space surrounding the subwidgets that the color mixing widget contains. Use the common widget attribute **margin_width** to specify the amount of space between the left and right edges of the subwidgets (the default is 10 pixels). Use the common widget attribute **margin_height** to specify the amount of space between the top and bottom edges of the subwidgets (the default is 10 pixels). Specify these margins in pixels.

In addition, you can specify the amount of space surrounding the two window widgets in the default display subwidget. Use the **display_win_margin** attribute to specify the size for all the margins in the display subwidget (the default is 20 pixels). The **display_win_margin** attribute affects only the default color display subwidget.

Using the Color Mixing Widget

11.2 Creating a Color Mixing Widget

11.2.2.3 Labeling the Color Mixing Widget

You can specify the text in each of the labels that the color mixing widget contains by assigning values to color mixing widget attributes. You must specify these labels as compound strings.

Table 11–4 lists the color mixing widget label attributes, describing the location of the label in the color mixing widget and the default text value of the label.

Table 11–4 Color Mixing Widget Label Attributes

Label Attribute	Function	Default Text
Common Labels		
main_label	Specifies the text that appears at the top of the color mixing widget, centered between the left and right borders	No default
display_label	Specifies the text that appears above the color display subwidget, centered between the left and right borders	No default
mixer_label	Specifies the text that appears above the color mixer subwidget, centered between the left and right borders	No default
option_label	Specifies the text that appears inside the color model option menu subwidget	"Color Model: "
hls_label	Specifies the text that appears as the top item in the color model option menu	"HLS"
rgb_label	Specifies the text that appears as the bottom item in the color model option menu	"RGB"
ok_label	Specifies the text that appears inside the OK push button	"OK"
apply_label	Specifies the text that appears inside the Apply push button	"Apply"
reset_label	Specifies the text that appears inside the Reset push button	"Reset"
cancel_label	Specifies the text that appears inside the Cancel push button	"Cancel"
HLS Color Model Labels		
hue_label	Specifies the text that appears to the left of the top scale subwidget	"Hue"
sat_label	Specifies the text that appears to the left of the middle scale subwidget	"Saturation"
light_label	Specifies the text that appears to the left of the bottom scale subwidget	"Lightness"

(continued on next page)

Using the Color Mixing Widget

11.2 Creating a Color Mixing Widget

Table 11–4 (Cont.) Color Mixing Widget Label Attributes

Label Attribute	Function	Default Text
HLS Color Model Labels		
black_label	Specifies the text that appears below the left end of the middle scale subwidget	"Black"
white_label	Specifies the text that appears below the right end of the middle scale subwidget	"White"
gray_label	Specifies the text that appears below the left of the bottom scale subwidget	"Gray"
full_label	Specifies the text that appears below the right end of the bottom scale subwidget	"Full"
RGB Color Model Labels		
slider_label	Specifies the text that appears above the left end of the top scale subwidget	"Percentage"
red_label	Specifies the text that appears below the left end of the top scale subwidget	"Red"
green_label	Specifies the text that appears below the left end of the middle scale subwidget	"Green"
blue_label	Specifies the text that appears below the left end of the bottom scale subwidget	"Blue"
value_label	Specifies the text that appears above the column of text subwidgets	"Value"

If you do not specify values for the **main_label**, **display_label**, or **mixer_label** attributes, the color mixing widget does not include these label subwidgets. If you specify a null value for the **ok_label**, **apply_label**, **reset_label**, or **cancel_label** attributes, the color mixing widget deletes the push button subwidget.

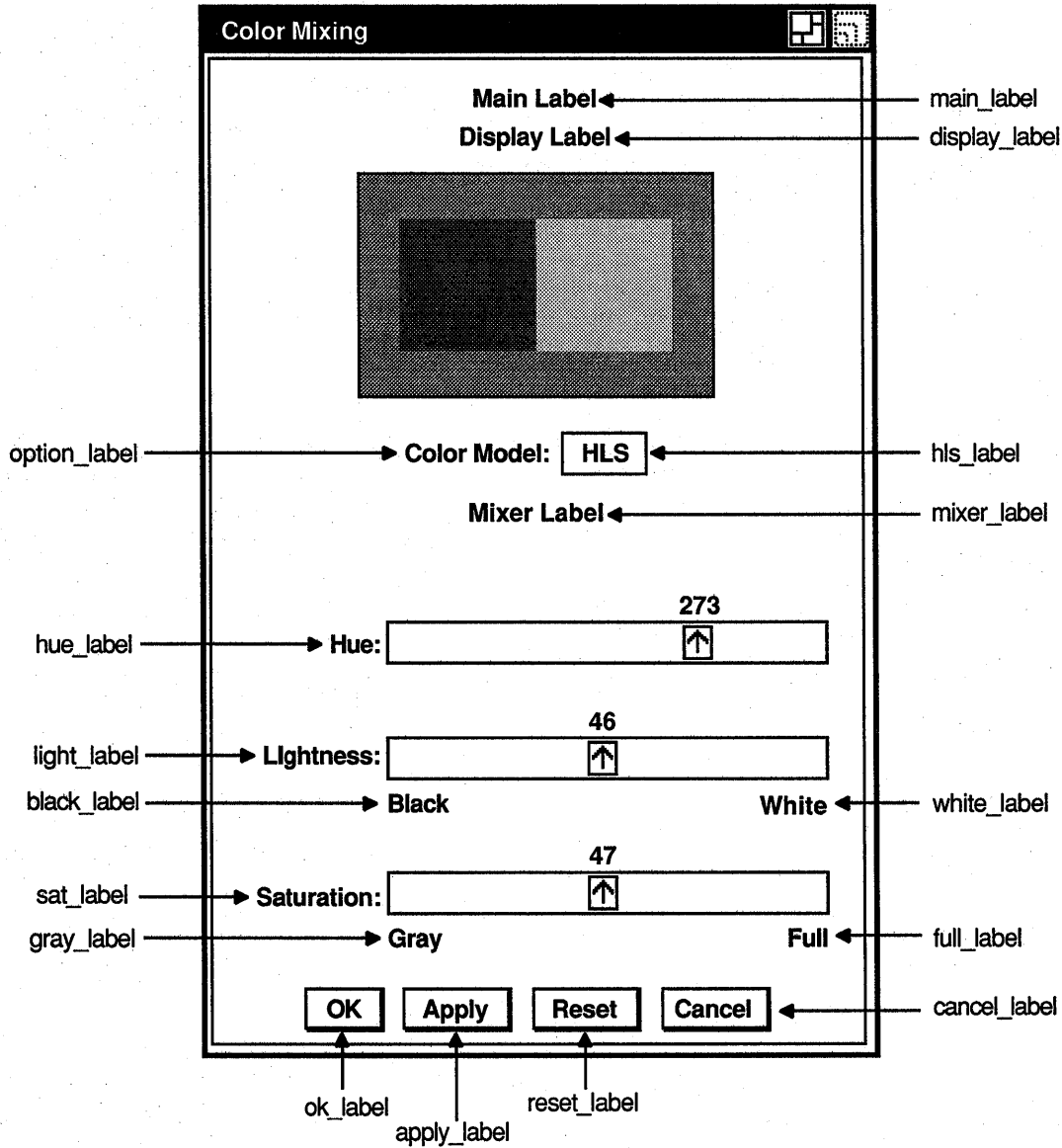
Note that the attributes that specify the text labels in the color mixer subwidget, both the HLS and RGB versions, work only with the default color mixer subwidget.

Figure 11–3 shows the labels in the color mixing widget using the HLS color model.

Using the Color Mixing Widget

11.2 Creating a Color Mixing Widget

Figure 11-3 Labels in the Color Mixing Widget (HLS Color Model)



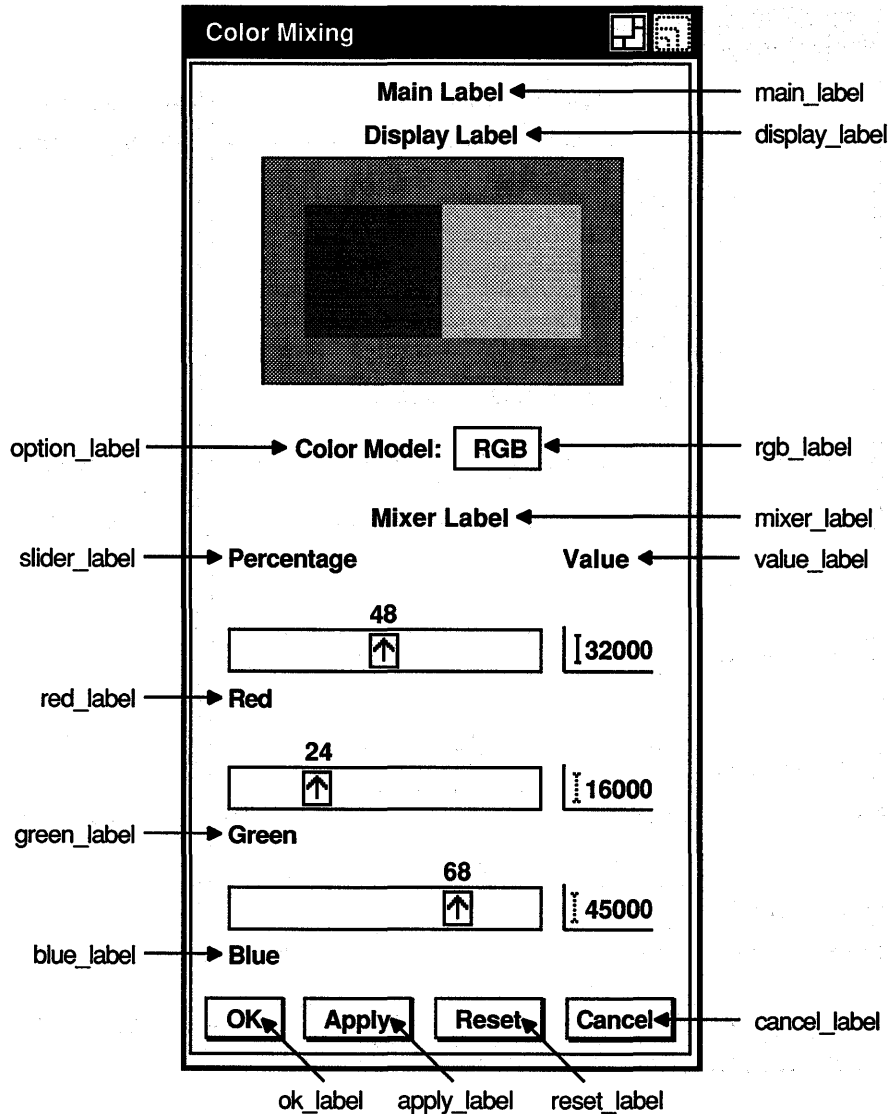
ZK-1889A-GE

Figure 11-4 show the labels in the color mixing widget using the RGB color model.

Using the Color Mixing Widget

11.2 Creating a Color Mixing Widget

Figure 11-4 Labels in the Color Mixing Widget (RGB Color Model)



ZK-1276A-GE

11.2.2.4 Defining the Background Color of the Color Display Subwidget

Use the **back_red_value**, **back_green_value**, and **back_blue_value** attributes to define the color of the background of the display subwidget. These attributes work only with the default color display subwidget.

11.2.2.5 Adding a Work Area to the Color Mixing Widget

To specify that the color mixing widget contain a work area subwidget, create the widget that you want to be the subwidget and assign the widget identifier as the value of the **work_window** attribute.

You do not have to manage the work area subwidget.

Using the Color Mixing Widget

11.3 Supporting Other Color Models

11.3 Supporting Other Color Models

The color mixing widget has built-in support for the HLS and the RGB color models. You can extend the color mixing widget to support other color models by replacing the default color mixer subwidget and the color display subwidget with widgets of your own design. Section 11.3.1 and Section 11.3.2 describe how to replace these subwidgets.

Whatever color system you choose to support, remember that the X Window System, Version 11, defines colors by their RGB values. Your custom subwidget must convert whatever values it accepts into RGB values and provide these values to the color mixing widget, which returns the values to the application as callback data. (For more information about obtaining color resources as well as an example of converting color values from another color model to RGB, see the color example program in the *VMS DECwindows Xlib Programming Volume*.)

11.3.1 Replacing the Color Display Subwidget

To replace the default color display subwidget, specify the identifier of the new color display subwidget as the value of the **display_window** attribute. You can switch back to the default color display subwidget at any time by setting this attribute to null. If you do not specify a value for this attribute, the color mixing widget uses the default color display subwidget.

If you replace the default color display subwidget, you must provide a procedure to update the new-color window when a user changes the color mixer widget. The color mixing widget calls this routine whenever a user changes a value in the color mixer subwidget. Pass the address of this routine as the value of the **set_new_color_proc** attribute.

11.3.2 Replacing the Color Mixer Subwidget

To replace the default color mixer subwidget with one of your own design, assign the widget identifier of the new subwidget as the value of the **mixer_window** attribute. To switch back to the default color mixer subwidget, set this attribute to null. If you do not specify a value for this attribute, the color mixing widget uses the default color mixer subwidget.

11.4 Associating Callbacks with a Color Mixing Widget

When a user presses the OK, Apply, or Cancel push button, the color mixing widget performs a callback to your application. (Activating the Reset button does not trigger a callback.) When the color mixing widget performs a callback, it returns data to your application, including the RGB values that define the original color (specified in the **orig_red_value**, **orig_green_value**, and **orig_blue_value** attributes) and the RGB values that define the new color (specified in the **new_red_value**, **new_green_value**, and **new_blue_value** attributes). For complete information about the data returned in the callback by the color mixing widget, see the *VMS DECwindows Toolkit Routines Reference Manual*.

Using the Color Mixing Widget

11.4 Associating Callbacks with a Color Mixing Widget

To associate a callback routine with a color mixing widget callback, pass a callback routine list to one of the color mixing widget callback attributes. Table 11–5 lists the callback attributes and describes the conditions that trigger these callbacks.

Table 11–5 Color Mixing Widget Callbacks

Callback Attribute	Conditions for Callback
activate_callback	The user has clicked the OK push button widget in the color mixing widget.
apply_callback	The user has clicked the Apply push button widget in the color mixing widget.
cancel_callback	The user has clicked the Cancel push button widget in the color mixing widget.
help_callback	A user has pressed the Help key while clicking MB1 in the color mixing widget.

12 Using Help

VMS DECwindows applications can use the XUI Toolkit help widget to display general and context-sensitive user assistance information in response to a user request. This chapter describes how to include the help widget in an application. The following topics are described:

- An overview of the XUI Toolkit help widget
- Help library information
- Creating the help widget
- Using the help widget

The *XUI Style Guide* describes the recommended appearance and behavior of the help widget.

12.1 Overview of the Help Widget

The XUI Toolkit help widget is a modeless widget that allows you to display appropriate, context-sensitive help text in response to a user query.

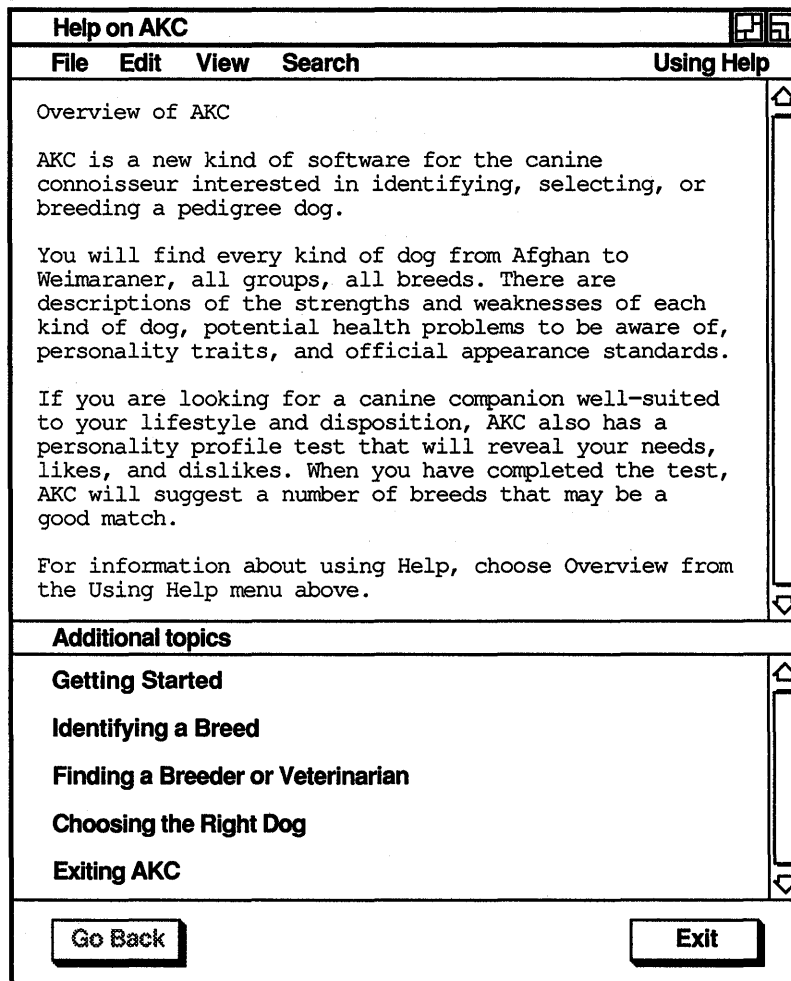
The help widget can be viewed as an independent application that your application calls to provide help functions. Using the help widget, you can create and manage one or more help windows and determine the first topic to be displayed to the user. The modeless behavior of the help widget permits an application to support one or more concurrent help widgets.

Figure 12–1 shows a sample help widget.

Using Help

12.1 Overview of the Help Widget

Figure 12–1 Sample XUI Toolkit Help Widget



ZK-0241A-GE

Your application is responsible for creating a Help pull-down menu widget with push button widgets for your chosen help topics. The labels for the push button widgets should indicate the types of help available. You may want to provide the following help topics:

- The Overview topic, to display overview or context-sensitive help
- The About topic, to display information about the application, such as its formal name and version number
- The Glossary topic, to display an application-specific glossary

The *XUI Style Guide* recommends that applications include Overview and About topics in the Help pull-down menu widget. You can also add application-specific help topics.

There are four possible ways to invoke help:

- The user clicks on the Help option in the menu bar. Your application calls the help widget to create a help window.
- The user presses the Help key. Pressing the Help key is equivalent to selecting the Help pull-down menu widget of the window that has input focus.
- The user types a help topic command string in a command window widget. Your application must include a command window widget to support this mechanism.
- The user holds down the Help key and clicks on a widget in the user interface. This combination of actions causes the widget to perform a help callback to your application. This is called **context-sensitive help**. You can use the callback routine to create a help widget (or change an existing help widget) to display appropriate help text. See Section 12.5 for more information.

12.1.1 Help Widget Terminology

This chapter uses the terms defined in Table 12–1 to describe the help widget.

Table 12–1 Help Widget Terminology

Term	Definition
help widget	The general name for all modules that comprise the widget.
help window	The window that contains all of the help information. There is one help window for each help widget. Help display is synonymous with help window.
help session	All the help interactions (requests, answers, and so on) that occur while an application is running. It can be composed of several help widgets.

12.2 Help Library Information

When you create a help widget, you pass a help library specification to the help widget creation routine. The help widget uses this specification to locate and read the help files. The help libraries for VMS DECwindows applications are based on a conventional help library.

This section describes how to use help libraries with the help widget. For more information about the VMS Librarian Utility (LIBRARIAN), see the *VMS Librarian Utility Manual*.

Use the **library_type** and **library_spec** attributes to specify a library type and specification for the help widget. Predefined values for **library_type** are DWT\$C_TEXT_LIBRARY or DwtTextLibrary; **library_spec** specifies a help library file specification. The help widget uses these attributes to identify the location and type of help topic

Using Help

12.2 Help Library Information

database. Once you have invoked the help widget, it is able to navigate only within the selected help library.

A help library has a default file type of HLB and defaults the file type of input files to HLP. The files that you insert into help libraries are text files that you build using a program or a text editor. Each help input file can contain one or more modules; each module contains a group of related keys numbered key 1 to key 9. Each key represents a hierarchical level within the module.

LIBRARIAN stores a key-1 name as its module name. The key-2 through key-9 names identify subtopics that are related to the key-1 name. For the purpose of making the HLP file easier to maintain, it is good practice to associate top-level help topics with key-1 names. There is no requirement to do so.

Your application can access a module from the key-1 name or from any key in the module. For example, if you have help push button widgets for Overview, About, and Glossary top-level topics, you might maintain the help library as one file called APPLICATION.HLP and create a VMS help library called APPLICATION.HLB. APPLICATION.HLP would contain a separate module, identified by a key-1 name, for each top-level topic. You can also maintain the help library modules in multiple HLP files.

When a user asks for help on the Overview topic, your application could determine from the push button widget help callback that the user wanted Overview help. Your application could then create a help widget and, through the **first_topic** or the **overview_topic** attribute, pass the help widget the string that identifies the correct help topic. This string would identify a key-1 name or another key in the module.

The help widget uses the key name hierarchy to find the help topic. For example, if you want to directly access the help topic identified by a key-3 name, you must also specify the key-1 and key-2 names that form a path to the key-3 name.

The help widget looks in the specified library for the module defined by the string and displays the text. If the string identifies a key-1 name, any key-2 subentries in the Overview module automatically appear as additional topics, which are hotspots.

If the user then asks for help on a key-2 subentry, the help widget displays the key-2 text, and the key-3 subentries appear as additional topics, and so on.

12.2.1 VMS Help Library Enhancements

The help widget provides several extensions to LIBRARIAN. The extensions provide the help widget with more sophisticated search capabilities. The extensions take the form of help widget commands (special text lines) in conventional VMS help topics. These commands have the following format:

```
=name operand(s)
```

Using Help

12.2 Help Library Information

The following syntax rules apply to all commands:

- Commands should be the first lines of text in a help topic.
- The first character of a command line must be an equal sign (=).
- The command name must immediately follow the equal sign.
- Command names are not case sensitive and cannot be abbreviated.
- At least one space must precede the command operand.
- The remainder of the line is the command operand.

The extensions to LIBRARIAN are described in Table 12–2.

Table 12–2 VMS Librarian Utility Extensions

Command Name	Description
=TITLE	Permits a case-sensitive title to be associated with the help topic. This title is displayed in situations where a topic is identified. For example, <i>Overview of the Help Widget</i> . If no title is provided, the help library topic key becomes the topic title. The help widget Search menu allows users to search by keyword and title.
=KEYWORD	Permits one or more case-insensitive keywords to be associated with the help topic. If more than one keyword is specified, the individual keywords must be separated by a comma or at least one space. The help widget Search menu allows users to search by keyword and title.
=NOSEARCH	Disables search operations for title and keywords on a specific topic.
=INCLUDE	Permits help topics to be shared across modules within a single help library. The operand of the INCLUDE command is a help topic key name. See Section 12.3.1 for more information about help topic key names. The title of the included topic is automatically added as an additional topic.

Example 12–1 shows the contents of a sample help file.

Using Help

12.2 Help Library Information

Example 12-1 Sample Help File

```
①1 overview
②=Title Overview of the Help Widget
③=Keyword overview
④=Include programming creating create_help_widget
```

A help widget is a modeless widget that allows you to display appropriate, context-sensitive help text in response to a user query. The help widget can be viewed as an independent application that your application calls to provide help functions.

The help widget creates and manages one or more help windows and determines the first topic to be displayed to the user.

```
⑤2 functions_1
=Title Using the help widget
=Keyword overview functions
To use the help widget, you perform the following
steps:
```

1. Use the VMS Librarian Utility (LIBRARIAN) to create a help library.
2. Create a Help menu bar item for your application. The Help menu item should be located at the right of the menu bar. If the menu bar is wider than a line, the Help menu item should be located at the bottom right.

.
.
.

-
- ① The name of the key-1 module is *overview*. You pass the string *overview* to the help widget. The help widget then searches the help library for a module with this name and displays the text. A module is terminated by either another key-1 name or by an end-of-file record.
 - ② The title that the help widget displays for the Overview topic is **Overview of the Help Widget**.
 - ③ The name of the keyword topic to search for with the help widget Search function is *overview*.
 - ④ The included topic key name from the programming module is **programming creating create_help_widget**. The title of the key identified by the =INCLUDE tag is displayed as an additional topic, which is a hotspot.
 - ⑤ The name of the key-2 subentry in the Overview module is **functions_1**. The **functions_1** subentry appears as an additional topic.

12.3 Modifying Help Widget Appearance

You can use the help widget attributes described in Table 12–3 to modify the appearance of the help widget. Use one of the help widget creation mechanisms to assign values to these attributes when you create the help widget.

Table 12–3 Help Widget Appearance Attributes

Attribute	Description
help_font	Specifies the font of the text displayed in the help text widget. The default is language dependent. The American English default uses <code>-*-TERMINAL-MEDIUM-R-NARROW-*-140-*-C-*-ISO8859-1</code> for all display text. This attribute is ignored if library_type is not <code>DWT\$C_TEXT_LIBRARY</code> or <code>DwtTextLibrary</code> .
cols	Specifies the width, in characters, of the help text displayed by the help widget. The default is language dependent. The American English default is 55 characters. This attribute is ignored if library_type is not <code>DWT\$C_TEXT_LIBRARY</code> or <code>DwtTextLibrary</code> .
rows	Specifies the height, in characters, of the help text displayed by the help widget. The American English default is 20 lines. This attribute is ignored if library_type is not <code>DWT\$C_TEXT_LIBRARY</code> or <code>DwtTextLibrary</code> .
default_position	Specifies whether to use the default help window position. If default_position is true, any x- and y-coordinate values you may have specified in the argument list are ignored. The default position is adjacent to the parent of the help widget, which is the top-level window of your application.

12.3.1 Help Widget Topic Information

You can use the help widget attributes described in Table 12–4 to specify the topics of the help widget.

Table 12–4 Help Widget Topic Attributes

Attribute	Description
first_topic	Specifies the first help topic to be displayed. If the first_topic attribute is not specified (set to null), the help widget displays an empty window with a list of level 1 topics in the additional topic list box. See Section 12.5 for information about using first_topic to specify context-sensitive help.
overview_topic	Specifies the Overview topic to be displayed. The Overview topic is displayed when you select the Go To Overview menu item from the View menu. As described in Section 12.2, your application uses the overview_topic attribute to pass the help widget a string that identifies the key name of the Overview module. Overview is generally a key-1 name.
glossary_topic	Specifies the Glossary topic to be displayed. Your application uses the glossary_topic attribute to pass the help widget a string that identifies the key name of the Glossary module. Glossary is generally a key-1 name. Set glossary_topic to null if your application does not support glossary help.

If you specify a help topic identified by a subkey name, you must also specify the key names that form the path to the subkey name. The key names must be separated by at least one space.

Using Help

12.3 Modifying Help Widget Appearance

For example, given the following module, if you wanted to display the `create_help_widget` key-3 help text as the first topic in the help widget, you would pass the compound string "programming creating create_help_widget".

- 1 programming
- 2 creating
- 3 create_help_widget

12.4 Using the Help Widget

This section describes general programming considerations for using the help widget.

The most basic approach to using the help widget is to create it, manage it to cause the help window to appear, and destroy it when the user is done. However, any changes to the help window, such as resizing, are lost when the widget is destroyed.

If your application destroys a help widget and then re-creates it, your application assumes the help widget creation overhead. However, a help library is initialized when it is first opened by the help widget and is cached in memory until the application closes down. Once a help widget initializes a help library on behalf of your application, the library is not reinitialized unless your application is restarted.

The recommended approach is to create the help widget once and use the same help widget each time the user requests help. You can do this by specifying a new first topic in the **first_topic** attribute (using the SET VALUES routine) and managing the widget (using the MANAGE CHILD routine) to cause the help window to appear.

To use the help widget, perform the following steps:

- 1 Use LIBRARIAN to create a help library. See Section 12.2 for more information.
- 2 Create a Help menu bar item for your application. To conform to the guidelines of the *XUI Style Guide*, use the **help_menu_right** attribute of the menu bar widget to position the Help menu item at the right end of the menu bar. If the menu bar widget wraps onto additional lines, the menu bar widget positions the Help menu item at the bottom right of the menu bar.
- 3 Create a Help pull-down menu widget with items such as Overview, About, and Glossary. For information about creating a pull-down menu, see Section 6.4.

An application that does not support a specific help menu item should not include that item in its Help pull-down menu widget. To accommodate the situation where the user does not select a help topic, the Help pull-down menu widget should have a callback to the Help menu widget.

- 4 Create the help buttons for the pull-down menu widget. Create one push button widget for each topic on the Help pull-down menu widget. The push button widgets are associated with the routines to call when the buttons are pressed.
- 5 Use any of the widget creation routines listed in Table 12–5 to create an instance of the help widget.

Table 12–5 Help Widget Creation Routines

High-level routine	Use the HELP routine to create a help widget.
Low-level routine	Use the HELP CREATE routine to create a help widget.
UIL object type	Use the help_box object type identifier to create a help widget in a UIL module.

- 6 Specify the callback routine to be called when the help widget is unmapped.

Example 12–2 creates a help widget in the C language using the MIT C binding.

Example 12–2 Creating a Help Widget

```
void help_menu_cb(widgetID, tag, cb_struct)
Widget *widgetID;
caddr_t tag;
DwtMenuCallbackStruct *cb_struct;
{
    DwtCompString appname, libname, overview, topic, glossary;
    unsigned int ac;
    Arg arglist[10];
    DwtCallback help_unmap_CB [2] = {
        {help_unmap, NULL},
        {NULL, NULL}
    };
    topic = NULL;
    if (cb_struct->s_widget == data.help_about)
        topic = DwtLatin1String ("About");
    if (cb_struct->s_widget == data.help_glossary)
        topic = DwtLatin1String ("Glossary");
    if (data.help_widget == NULL)
    {
        appname = DwtLatin1String ("My_App_Name");
        libname = DwtLatin1String ("Help_File_Library");
        overview = DwtLatin1String ("Overview");
        glossary = DwtLatin1String ("Glossary");
```

(continued on next page)

Using Help

12.4 Using the Help Widget

Example 12–2 (Cont.) Creating a Help Widget

```
data.help_widget = DwtHelp( data.toplevel, "Help", TRUE, 0, 0,
                             appname, DwtTextLibrary, libname, topic,
                             overview, glossary, help_unmap_CB);

XtManageChild (data.help_widget);
XtFree (appname);
XtFree (libname);
XtFree (overview);
XtFree (glossary);
}
else
{
    ac = 0;
    ④ XtSetArg (arglist[ac], DwtNfirstTopic, topic); ac++;
      XtSetValues (data.help_widget, arglist, ac);
}
    ⑤ if( !XtIsManaged( data.help_widget )
        XtManageChild( data.help_widget );

    if( topic != NULL)
        XtFree (topic);
return;
}
.
.
.
```

① All of the push button widgets on the Help pull-down menu can be associated with a single help widget creation routine, in this example *help_menu_cb*.

② The *help_menu_cb* routine uses the identity of the push button widget that called it to set the value for **first_topic**. The **first_topic** argument determines which help topic is displayed first.

If the About push button widget has called *help_menu_cb*, **first_topic** is set to "About". If the Glossary push button widget has called *help_menu_cb*, **first_topic** is set to "Glossary".

③ If the help widget does not already exist, create it.

The help widget sizes and locates the help window based on the size and location of its parent, which is the top-level window of your application. The help widget opens the help library that you specify. All of the help widgets for an application can (but are not required to) use the same help library.

④ If the help widget already exists, set **first_topic**. If the About or Glossary push button widget has called *help_menu_cb*, **first_topic** is already set to About or Glossary.

If the Overview push button widget has called *help_menu_cb*, the value of **first_topic** is null, and the **overview_topic** is displayed first.

Once the widget is created, you can use the SET VALUES routine to specify a new first topic.

- ⑤ If the widget is not already managed, call the `MANAGE CHILD` routine to cause the help window to appear.

Example 12-3 shows a sample UIL help widget implementation.

Example 12-3 UIL Help Widget Implementation

```
.
.
.
PROCEDURE display_help(compound_string);
PROCEDURE create_help();
PROCEDURE unmap_help();

VALUE
    overview_fr    : compound_string("overview");
    about_fr       : compound_string("aboutframe");
    glossary_fr    : compound_string("glossary");
.
.
.
object main_help : HELP_BOX
{
    ARGUMENTS
    {
        APPLICATION_NAME = 'Help Example';
        GLOSSARY_TOPIC = compound_string("glossary");
        OVERVIEW_TOPIC = compound_string("overview");
        LIBRARY_SPEC = compound_string("sys$help:decw$helphelp.hlb");
        LIBRARY_TYPE = DwtTextLibrary;
    };

    CALLBACKS
    {
        CREATE = PROCEDURE create_help();
        UNMAP = PROCEDURE unmap_help();
    };
};

.
.
.
object s_menu_bar : MENU_BAR
{
    ARGUMENTS
    {
        ORIENTATION = DwtOrientationHorizontal;
        ① MENU_HELP_WIDGET = PULLDOWN_ENTRY help_menu_entry;
    };

    CONTROLS
    {
        .
        .
        .
        PULLDOWN_ENTRY help_menu_entry;
    };
};
```

(continued on next page)

Using Help

12.4 Using the Help Widget

Example 12-3 (Cont.) UIL Help Widget Implementation

```
.  
. .  
.  
  
object help_menu_entry : PULLDOWN_ENTRY  
{  
  ARGUMENTS  
  {  
    LABEL_LABEL = "Help";  
  };  
  
  CONTROLS  
  {  
    PULLDOWN_MENU help_menu;  
  };  
};  
  
② object help_menu : PULLDOWN_MENU  
{  
  CONTROLS  
  {  
    PUSH_BUTTON help_button;  
    PUSH_BUTTON help_about;  
    PUSH_BUTTON help_glossary;  
  };  
};  
  
③ object help_button : PUSH_BUTTON  
{  
  ARGUMENTS  
  {  
    LABEL_LABEL = 'Help';  
  };  
  
  CALLBACKS  
  {  
    ACTIVATE = PROCEDURE display_help(overview_fr);  
  };  
};  
  
④ object help_about : PUSH_BUTTON  
{  
  ARGUMENTS  
  {  
    LABEL_LABEL = 'About';  
  };  
  
  CALLBACKS  
  {  
    ACTIVATE = PROCEDURE display_help(about_fr);  
  };  
};  
  
⑤ object help_glossary : PUSH_BUTTON  
{  
  ARGUMENTS  
  {  
    LABEL_LABEL = 'Glossary';  
  };  
};
```

(continued on next page)

Example 12–3 (Cont.) UIL Help Widget Implementation

```
CALLBACKS
{
    ACTIVATE = PROCEDURE display_help(glossary_fr);
};
};
```

- ❶ Define the Help menu item in the menu bar widget.
- ❷ Define the Help pull-down menu widget.
- ❸ Define the Help push button widget.
- ❹ Define the About push button widget.
- ❺ Define the Glossary push button widget.

12.5 Context-Sensitive Help

In context-sensitive help, the application presents direct help on the current topic rather than starting at a higher level and working down through a help hierarchy. Users do not have to navigate through several layers of help to find the information they need.

The help widget does not distinguish general help from context-sensitive help and cannot tell which type of help your application requests. Your application is responsible for implementing context-sensitive help and the help callback.

You can use the **first_topic** attribute of the help widget to specify a context-sensitive help topic. To do this, associate a help callback routine with the widgets for which you want to provide help. When a user moves the pointer cursor onto the widget, holds down the Help key, and presses MBI, the widget's help callback routine is called.

Note: All widgets that are a subclass of the common widget class support a help callback. Other widgets may also support the help callback, but there is no requirement to do so.

There are two possible ways to implement the help callback routine:

- You can directly specify the key name for the help topic in the callback routine. The disadvantage to this method is that the key names specified in **first_topic** must match the key names in the help library. This might be difficult to maintain if you have help support for a large number of widgets.
- The callback routine can specify the key name as a resource name. Create a UIL module that maps the resource names to the key names for the help topics. If you change the key name of the help topic, you do not have to change application code.

Using Help

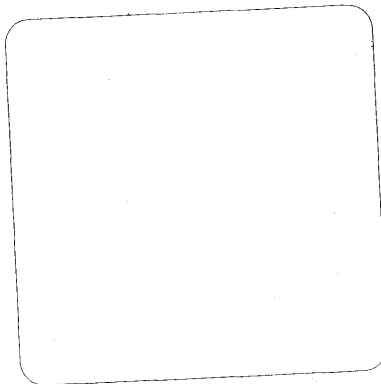
12.5 Context-Sensitive Help

Regardless of which method you choose, the help callback routine should perform the following tasks:

- Check to see if a help widget already exists. If a help widget has not yet been created, the routine creates a help widget with **first_topic** set to the key name or resource name of the topic for which you want to display help.

If a help widget has been created but is not managed, use the SET VALUES routine to specify the key name or resource name of the topic for which you want to display help as **first_topic**. This avoids the overhead of creating a new help widget.

- Call the MANAGE CHILD routine to display the help window. The appropriate help text is displayed. Any subtopics or included topics are displayed as additional topics. Use include commands in the HLB library to link together context-sensitive help topics.



13 Using the Cut and Paste Routines

This chapter provides the following:

- An overview of the cut and paste routines
- A detailed description of how to use the cut and paste routines in your application
- A description of how to implement the QuickCopy function

13.1 Overview of the Cut and Paste Routines

The XUI Toolkit includes a set of cut and paste routines that provide convenient access to the **clipboard**. The clipboard is a buffer, external to your application, in which you can temporarily store data. You use the cut and paste routines to copy data to the clipboard, inquire about the contents of the clipboard, or copy data from the clipboard. Table 13-1 lists the cut and paste routines in groups by function; later sections describe their use.

Table 13-1 Cut and Paste Routines

Routine Name	Description
Copying to the Clipboard	
START COPY TO CLIPBOARD	Sets up storage and data structures to receive clipboard data. (See also BEGIN COPY TO CLIPBOARD.)
COPY TO CLIPBOARD	Copies a data item to the clipboard.
END COPY TO CLIPBOARD	Ends the COPY TO CLIPBOARD operation and places the data in the clipboard data structure.
CANCEL COPY TO CLIPBOARD	Cancels the current COPY TO CLIPBOARD operation.
UNDO COPY TO CLIPBOARD	Deletes the last data item placed on the clipboard if the item was placed there by this application.
RECOPY TO CLIPBOARD	Copies to the clipboard a data item that was previously passed by name.
LIST PENDING ITEMS	Returns a list of pending items as data ID and private ID pairs for a specified format name.
CANCEL COPY FORMAT	Indicates that the application will no longer supply a data item that the application had previously passed by name to the clipboard.

(continued on next page)

Using the Cut and Paste Routines

13.1 Overview of the Cut and Paste Routines

Table 13–1 (Cont.) Cut and Paste Routines

Routine Name	Description
Copying to the Clipboard	
CLIPBOARD REGISTER FORMAT	Registers the length of the data for formats not specified by the conventions defined in the ICCCM ¹ .
Copying from the Clipboard	
START COPY FROM CLIPBOARD	Indicates that the application is ready to start copying data from the clipboard and locks the clipboard.
COPY FROM CLIPBOARD	Retrieves a data item from the clipboard.
END COPY FROM CLIPBOARD	Indicates that the application has completed copying data from the clipboard and unlocks the clipboard.
Inquire Routines	
INQUIRE NEXT PASTE COUNT	Returns the number of data item formats that are available for the next paste clipboard data item.
INQUIRE NEXT PASTE FORMAT	Returns a specified format name for the next paste data item on the clipboard.
INQUIRE NEXT PASTE LENGTH	Returns the length of the data stored under a specified format name for the next paste clipboard item.
Obsolete Routines²	
BEGIN COPY TO CLIPBOARD	Superseded by the START COPY TO CLIPBOARD routine. Like the START COPY TO CLIPBOARD routine, this routine sets up storage and data structures to receive clipboard data. However, the START COPY TO CLIPBOARD routine accepts the time stamp of the event causing the copy operation as one of its arguments. The addition of this argument makes the START COPY TO CLIPBOARD routine comply with the ICCCM conventions. The BEGIN COPY TO CLIPBOARD routine is not ICCCM compliant.

¹ICCCM is described in the X Window System, Version 11 *Inter-Client Communication Conventions Manual* by David S. H. Rosenthal.

²These routines have been superseded by other cut and paste routines, but are still supported.

(continued on next page)

Using the Cut and Paste Routines

13.1 Overview of the Cut and Paste Routines

Table 13–1 (Cont.) Cut and Paste Routines

Routine Name	Description
Obsolete Routines²	
CLIPBOARD LOCK	Locks the clipboard from access by other applications. If you use the START COPY FROM CLIPBOARD routine, which locks the clipboard, you do not have to use the CLIPBOARD LOCK routine.
CLIPBOARD UNLOCK	Unlocks the clipboard, enabling it to be accessed by other applications. If you use the END COPY FROM CLIPBOARD routine, which unlocks the clipboard, you do not have to use the CLIPBOARD UNLOCK routine.

²These routines have been superseded by other cut and paste routines, but are still supported.

13.1.1 Communicating with Other Applications

Because the clipboard is available to all applications running on a workstation, it enables communication between applications. For example, your application may copy data to the clipboard. While that data remains on the clipboard, any other application running on the workstation can obtain a copy of the data you copied to the clipboard.

The application performing a clipboard operation, such as putting data on the clipboard, has temporary ownership of the clipboard during the operation. This is called **locking** the clipboard. After the clipboard operation completes, any other application can perform a clipboard operation.

The clipboard can hold only a single data item (although it can hold that data item in more than one format). When an application copies data to the clipboard, the new data supersedes the previous contents of the clipboard.

(The QuickCopy function enables applications to exchange data without using the clipboard. For information about this feature, see Section 13.5.)

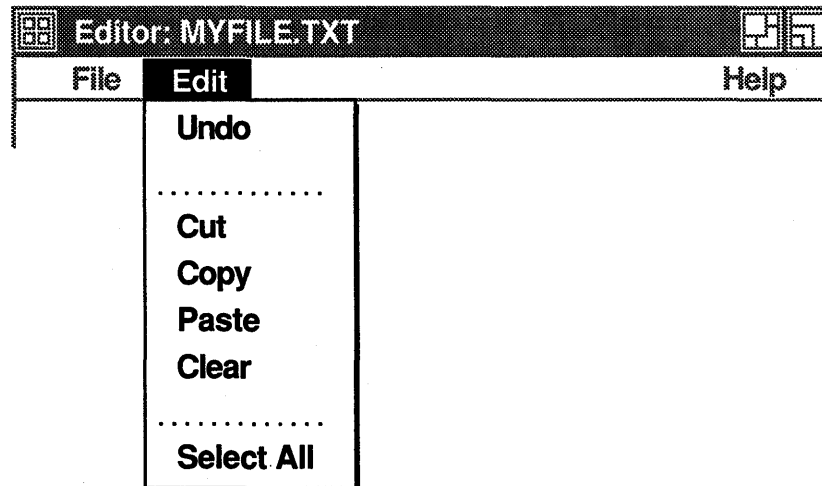
13.1.2 Implementing the Copy, Cut, and Paste Functions

The cut and paste routines implement the functions available in the Edit menu in the main menu bar of an application. Every application that allows a user to select text or graphics in its windows should provide functions in the Edit menu, such as copy, cut, and paste, that allow the user to manipulate the selected text or graphics. Figure 13–1 illustrates a typical Edit menu.

Using the Cut and Paste Routines

13.1 Overview of the Cut and Paste Routines

Figure 13-1 Edit Menu



ZK-0543A-GE

By using the functions in the Edit menu, a user can copy text or graphics from one application to another. The following sequence describes how a user performs a typical copy and paste operation:

- 1 The user selects a portion of text or graphics in an application window by pressing and holding MB1, dragging the pointer cursor through the data to be selected, and releasing MB1.
- 2 The user displays the Edit menu in the main menu bar, drags the pointer cursor to the Copy menu item, and releases MB1. In the callback routine associated with the copy function, your application determines what data the user has selected and copies the data to the clipboard. Section 13.2 describes how to copy data to the clipboard.
- 3 The user moves the pointer cursor to the application receiving the data and chooses the Paste menu item from the Edit menu. In the callback routine associated with the paste function, your application copies the current contents of the clipboard. Section 13.3 describes how to copy data from the clipboard.
- 4 Your application inserts the data at the current pointer cursor location. Your application is responsible for determining where the user wants to paste the data.

If you want to give the user maximum access to the clipboard, include the functions listed in Table 13-2 in the Edit menu in your application. The *XUI Style Guide* gives recommendations on how these functions should operate and how the Edit menu should look in your application. (Section 6.5 describes how to create a menu bar with pull-down menu items, such as an Edit menu, using the DECburger sample application as an example.)

Using the Cut and Paste Routines

13.1 Overview of the Cut and Paste Routines

Table 13–2 Edit Menu Functions

Edit Menu Item	Description
Copy	Copies the selected data to the clipboard.
Cut	Copies the selected data to the clipboard and deletes it from the window. Your application should save the deleted data if you allow users to cancel (undo) a cut operation, because an action by another application might alter the contents of the clipboard between the cut and the cancel operations.
Paste	Copies the data from the clipboard and allows the user to place the data in an application window.
Select All	Selects all of the data in the application window.
Clear	Cancel the selection of data in the application window. The Clear function does not affect the clipboard.
Undo	Cancel a cut, copy, or paste operation and restores the previous state. To cancel a cut operation, redraw the deleted data and delete it from the clipboard. Your application should not use the clipboard contents to redraw the deleted data. To cancel a copy operation, your application needs only to delete the data from the clipboard. To cancel a paste operation, your application should delete the pasted data from the application window. Cut and paste routines are not involved in canceling a paste operation. Save the deleted paste data if your application allows users to repeat a canceled operation.
Redo	Repeats a canceled operation. Repeating a paste operation restores the data saved from the last canceled paste operation. The clipboard is not involved with repeating a paste operation.

13.2 Copying Data to the Clipboard

To copy data to the clipboard, perform the following steps:

- 1 Identify the data that the user has selected and copy it into a buffer.

Your application is responsible for determining and highlighting the selected data on the display. The cut and paste routines do not perform this function but instead only copy a buffer of data to the clipboard.

The XUI Toolkit text widgets, described in Section 9.2.3, support the selection mechanism. These widgets highlight text selected by the user and can supply the selected text to your application.

- 2 Start the copy operation using the `START COPY TO CLIPBOARD` routine.

This routine sets up the data structures needed to transfer data to the clipboard. The clipboard is locked during the execution of the `START COPY TO CLIPBOARD` routine.

- 3 Copy the data to the clipboard using the `COPY TO CLIPBOARD` routine.

Using the Cut and Paste Routines

13.2 Copying Data to the Clipboard

In this call, you specify the data to be copied, its length, and its format. The clipboard is locked during execution of the COPY TO CLIPBOARD routine.

To make the single data item available in multiple formats, call the COPY TO CLIPBOARD routine for each format. To append data to the data item on the clipboard in any of the available formats, make additional calls to the COPY TO CLIPBOARD routine and specify the format.

If you have a large amount of data to copy to the clipboard and do not want to incur the overhead of a copy operation, you can copy the data to the clipboard **by name**. When passing data by name, you notify the clipboard that you have data available. If an application makes a request to the clipboard for the data, the clipboard requests that your application supply the data. For more information about copying data to the clipboard by name, see Section 13.2.1.

- 4 End the copy operation by using the END COPY TO CLIPBOARD routine.

The clipboard is locked during the execution of the END COPY TO CLIPBOARD routine.

Example 13-1 illustrates a simple copy-to-clipboard operation.

Example 13-1 Copying Data to the Clipboard

```
.
.
.
static void copy_proc( widget, tag, callback_data )
    Widget widget;
    char *tag;
    DwtAnyCallbackStruct *callback_data;
{
    ① Display *display;
      Window window;
      Time timestamp;
      int status;
      unsigned long item_id, count, buf_len;
      char copy_to_buffer[100];
      DwtCompString clip_label = DwtLatin1String("Your Application Name");
      .
      .
    ② display = XtDisplay( toplevel );
    ③ window = XtWindow( toplevel );
```

(continued on next page)

Using the Cut and Paste Routines

13.2 Copying Data to the Clipboard

Example 13-1 (Cont.) Copying Data to the Clipboard

```
④ switch( callback_data->event->type )
{
    case KeyPress:
    case KeyRelease:
        timestamp = callback_data->event->xkey.time;
        break;
    case ButtonPress:
    case ButtonRelease:
        timestamp = callback_data->event->xbutton.time;
        break;
    case MotionNotify:
        timestamp = callback_data->event->xmotion.time;
        break;
    case EnterNotify:
    case LeaveNotify:
        timestamp = callback_data->event->xcrossing.time;
        break;
    case PropertyNotify:
        timestamp = callback_data->event->xproperty.time;
        break;
    case SelectionClear:
        timestamp = callback_data->event->xselectionclear.time;
        break;
    case SelectionRequest:
        timestamp = callback_data->event->xselectionrequest.time;
        break;
    case SelectionNotify:
        timestamp = callback_data->event->xselection.time;
        break;
    default:
        timestamp = CurrentTime;
        break;
}
.
.
.
/*
 * Find out what the user has selected
 * and fill copy_to_buffer.
 */
.
.
.
⑤ status = DwtStartCopyToClipboard( display, window, clip_label,
                                   timestamp, 0, 0, &item_id );

if( status != ClipboardSuccess )
{
    return( 0 );
}

buf_len = strlen( copy_to_buffer );

⑥ status = DwtCopyToClipboard( display, window, item_id, "STRING",
                              copy_to_buffer, buf_len, 0, 0 );
```

(continued on next page)

Using the Cut and Paste Routines

13.2 Copying Data to the Clipboard

Example 13-1 (Cont.) Copying Data to the Clipboard

```
if( status != ClipboardSuccess )
{
    DwtCancelCopyToClipboard(display, window, item_id );
    return( 0 );
}

7 status = DwtEndCopyToClipboard( display, window, item_id );
if( status != ClipboardSuccess )
{
    DwtCancelCopyToClipboard(display, window, item_id );
    return( 0 );
}
.
.
return( 1 );
}
```

- ❶ The declarations of variables required by the cut and paste routines include a pointer to a Display structure (*display*), a window identifier (*window*), and a buffer to hold the data to be transferred to the clipboard (*copy_to_buffer*).
- ❷ The DISPLAY intrinsic routine returns a pointer to the display to which the application is connected.
- ❸ The WINDOW intrinsic routine returns the identifier of a window associated with one of the widgets in the user interface of the application. You can use any widget in the user interface; the example obtains the identifier of the window associated with the topmost widget in its application widget hierarchy, named *toplevel*.
- ❹ The time stamp of the event that triggered the callback is obtained from the X Event structure returned as callback data. You must find out the type of event that triggered the callback in order to access the time member of the X Event structure.
- ❺ The START COPY TO CLIPBOARD routine is called to start the copy operation. For arguments to the routine, you must pass the display and window information, a text string that can be associated with the contents of the clipboard, and the time stamp of the event that triggered the copy operation. In addition, you can also pass a valid widget identifier and the address of a callback routine to the routine, but these arguments are required when you pass data to the clipboard by name. (For information about passing data by name, see Section 13.2.1.) The START COPY TO CLIPBOARD routine returns a unique identifier assigned to this copy operation by the clipboard in its last argument.

If another application has the clipboard locked, the START COPY TO CLIPBOARD routine returns a value that indicates this condition (ClipboardLocked). If the clipboard is available, the routine returns a value that indicates success (ClipboardSuccess).

Using the Cut and Paste Routines

13.2 Copying Data to the Clipboard

- ⑥ The COPY TO CLIPBOARD routine is called to copy data to the clipboard. For arguments to this routine, you specify the identifier returned by the START COPY TO CLIPBOARD routine (**item_id**), the buffer of data being copied to the clipboard, and the length and format of the data. In addition, you can associate some private data with the clipboard data. The COPY TO CLIPBOARD routine returns an identifier in its last argument that can be used if you pass data by name.

You specify the format name argument as a character string. The ICCCM¹ supplies a set of predefined, named data formats. The *VMS DECwindows Toolkit Routines Reference Manual* lists these format names. Format names are case sensitive and require all uppercase letters. For more information about clipboard data formats, see Section 13.2.4.

- ⑦ The copy operation is ended by calling the END COPY TO CLIPBOARD routine.

To abort a copy operation, call the CANCEL COPY TO CLIPBOARD routine any time before the call to the END COPY TO CLIPBOARD routine. You do not need to call the END COPY TO CLIPBOARD routine after aborting a copy operation with CANCEL COPY TO CLIPBOARD.

13.2.1 Copying Data to the Clipboard by Name

To avoid incurring the overhead of copying to the clipboard a large amount of data that may not be requested by another application, you can copy the data to the clipboard **by name**. When passing data by name, you notify the clipboard that you have data available. If an application requests the data, the clipboard asks your application to supply the data.

To pass data to the clipboard by name, perform the following steps:

- 1 Start the copy-by-name operation using the START COPY TO CLIPBOARD routine.

However, to pass the data to the clipboard by name, you must specify values for the following arguments:

- Specify a valid widget identifier in the **widget** argument. You can choose any widget in your application widget hierarchy.
- Specify the address of a callback routine in the **callback** argument. (Pass the address of the callback routine, not a callback routine list as you would with a XUI Toolkit widget.)

The callback routine used for a clipboard callback is different from the standard callback routine that is used with XUI Toolkit widgets. The clipboard callback routine accepts four standard arguments. For more information about creating a clipboard callback, see Section 13.2.2.

- 2 Copy the name of the data to the clipboard by using the COPY TO CLIPBOARD routine.

¹ ICCCM is described in the X Window System, Version 11 *Inter-Client Communication Conventions Manual* by David S. H. Rosenthal.

Using the Cut and Paste Routines

13.2 Copying Data to the Clipboard

However, to pass the data to the clipboard by name, you must specify values for the following arguments:

- Specify the value of the **buffer** argument as null. Note that you must still specify a length for the data in the **length** argument.
- Specify private identification information, if desired, in the **private_id** argument.
- Specify the address of a variable to store the identifier returned by the COPY TO CLIPBOARD routine.

3 End the copy operation by using the END COPY TO CLIPBOARD routine.

When an application requests the data you have passed to the clipboard by name, the clipboard executes the callback routine specified in the START COPY TO CLIPBOARD routine.

13.2.2 Creating a Clipboard Callback Routine

The format of a clipboard callback routine is different from the standard callback routine used with the XUI Toolkit widgets. The clipboard callback routine requires four arguments; the standard widget callback requires three arguments. The format of a clipboard callback is as follows:

```
clipboard_callback( widget, data_id, private_id, reason)
Widget *widget;
int *data_id;
int *private;
int *reason;
```

In the **widget** argument, the clipboard returns the widget identifier passed to the START COPY TO CLIPBOARD routine.

The **data_id** argument is the identifier returned by the COPY TO CLIPBOARD routine.

The **private_id** argument is the data that you associated with the clipboard data in the COPY TO CLIPBOARD routine.

The **reason** argument contains the reason why the clipboard is executing the callback routine. The clipboard executes a callback when an application requests the data that had been passed by name (DwtCRClipboardDataRequest) or when the data passed by name has been deleted from the clipboard (DwtCRClipboardDataDelete). A data item passed by name is deleted from the clipboard if another application places data on the clipboard.

If an application has requested the data passed by name, use the RECOPY TO CLIPBOARD routine to copy the data to the clipboard. If your application will no longer supply the data, use the CANCEL COPY FORMAT routine to inform the clipboard.

If the data passed by name has been deleted from the clipboard, your application might respond, for example, by freeing the data buffer. A callback for this reason, however, does not require a response from your application.

Using the Cut and Paste Routines

13.2 Copying Data to the Clipboard

Before exiting, your application should check to see whether data that it passed to the clipboard by name is still on the clipboard. Use the LIST PENDING ITEMS routine to determine this information. This routine returns a list of the contents of the clipboard as data identifier/private identifier pairs. If the copy-by-name operation is still pending, your application should either copy the data to the clipboard or cancel the copy-by-name operation before exiting.

13.2.3 Deleting Data from the Clipboard

To delete a data item that you have copied to the clipboard, use the UNDO COPY TO CLIPBOARD routine. This routine can only delete data on the clipboard that was placed there by the application identified by the **display** and **window** arguments passed to the routine. If another application has placed data in the clipboard since your application, the UNDO COPY TO CLIPBOARD has no effect. When your application uses the UNDO COPY TO CLIPBOARD routine to delete data it had previously copied to the clipboard using the COPY TO CLIPBOARD routine, the previous contents of the clipboard are restored.

13.2.4 Specifying Clipboard Data Formats

When copying data to the clipboard, you must specify the format of the data in the COPY TO CLIPBOARD routine. When you assign a format to clipboard data, you instruct the X server to transfer the data in 8-bit, 16-bit, or 32-bit quantities.

To help communication between applications, the ICCCM¹ has established a set of standard data formats. The *VMS DECwindows Toolkit Routines Reference Manual* lists the names and sizes of the formats. With these conventions, applications can exchange data in predictable formats.

You can define your own clipboard data format. The format must use 8-bit, 16-bit, or 32-bit quantities. To ensure that X servers running on different machine architectures can perform proper byte-swapping operations on your data, register the format you define using the CLIPBOARD REGISTER FORMAT routine.

13.3 Copying Data from the Clipboard

To copy data from the clipboard, perform the following steps:

- 1 Start the copy operation using the START COPY FROM CLIPBOARD routine.

This routine locks the clipboard and leaves it locked until you call the END COPY FROM CLIPBOARD routine.

- 2 Copy the data from the clipboard using the COPY FROM CLIPBOARD routine.

¹ ICCCM is described in the X Window System, Version 11 *Inter-Client Communication Conventions Manual* by David S. H. Rosenthal.

Using the Cut and Paste Routines

13.3 Copying Data from the Clipboard

If the size of the data item is larger than the buffer you have provided, the COPY FROM CLIPBOARD routine returns the constant ClipboardTruncate. Calling the COPY FROM CLIPBOARD routine before the START COPY FROM CLIPBOARD routine enables you to copy the remaining data from the clipboard in increments. If you call the COPY FROM CLIPBOARD routine again, specifying the same data format, the routine starts copying at the point where the data was truncated in the preceding call.

- 3 End the copying operation using the END COPY FROM CLIPBOARD routine.

This routine unlocks the clipboard that had previously been locked by a call to the START COPY FROM CLIPBOARD routine.

You can use the COPY FROM CLIPBOARD routine without enclosing it between calls to START COPY FROM CLIPBOARD and END COPY FROM CLIPBOARD. However, when used without these routines, the COPY FROM CLIPBOARD routine cannot be used to copy data from the clipboard incrementally. Also, to ensure that your application complies with ICCCM conventions, use the START COPY FROM CLIPBOARD and END COPY FROM CLIPBOARD routines.

Example 13–2 illustrates the basic steps in copying data from the clipboard.

Example 13–2 Copying Data from the Clipboard

```
.
.
.
static void paste_proc( widget, tag, callback_data )
    Widget widget;
    char *tag;
    DwtAnyCallbackStruct *callback_data;
{
    1 Display *display;
    Window window;
    Time timestamp;
    int status;
    unsigned long count, format_name_len, bytes_copied, private_id;
    char copy_from_buffer[BUFLLEN];
    DwtCompString clip_label = DwtLatin1String("Test_label");
.
.
.
    display = XtDisplay( toplevel );
    window = XtWindow( toplevel );
```

(continued on next page)

Using the Cut and Paste Routines

13.3 Copying Data from the Clipboard

Example 13-2 (Cont.) Copying Data from the Clipboard

```
② switch( callback_data->event->type )
    {
        case KeyPress:
        case KeyRelease:
            timestamp = callback_data->event->xkey.time;
            break;
        case ButtonPress:
        case ButtonRelease:
            timestamp = callback_data->event->xbutton.time;
            break;
        case MotionNotify:
            timestamp = callback_data->event->xmotion.time;
            break;
        case EnterNotify:
        case LeaveNotify:
            timestamp = callback_data->event->xcrossing.time;
            break;
        case PropertyNotify:
            timestamp = callback_data->event->xproperty.time;
            break;
        case SelectionClear:
            timestamp = callback_data->event->xselectionclear.time;
            break;
        case SelectionRequest:
            timestamp = callback_data->event->xselectionrequest.time;
            break;
        case SelectionNotify:
            timestamp = callback_data->event->xselection.time;
            break;
        default:
            timestamp = CurrentTime;
            break;
    }
    .
    .
    .
③ status = DwtStartCopyFromClipboard( display, window, timestamp );
    if( status != ClipboardSuccess )
    {
        return( 0 );
    }
④ status = DwtCopyFromClipboard( display, window, "STRING",
                                copy_from_buffer, BUFLen,
                                &bytes_copied, &private_id );

    if( status != ClipboardSuccess )
    {
        return( 0 );
    }
⑤ status = DwtEndCopyFromClipboard( display, window );
```

(continued on next page)

Using the Cut and Paste Routines

13.3 Copying Data from the Clipboard

Example 13-2 (Cont.) Copying Data from the Clipboard

```
if( status != ClipboardSuccess )
{
    return( 0 );
}
.
.
return( 1 );
}
```

- ❶ The variables required by the cut and paste routines include a buffer to accept the data copied from the clipboard (*copy_from_buffer*).
- ❷ The time stamp of the event that triggered the callback is obtained from the X Event structure returned as callback data. You must find out the type of event that triggered the callback to access the time member of the X Event structure.
- ❸ The START COPY FROM CLIPBOARD routine is called to begin copying data from the clipboard. This routine locks the clipboard until a call to END COPY FROM CLIPBOARD unlocks it. For arguments, the START COPY FROM CLIPBOARD routine accepts a pointer to the display to which the application is connected, a window identifier associated with a widget used in the application user interface, and the time stamp of the event that triggered the operation.

If an application has locked the clipboard, the START COPY FROM CLIPBOARD routine returns a value (ClipboardLocked) that indicates this state. If the routine is able to lock the clipboard, it returns a value (ClipboardSuccess) to indicate success.

- ❹ The COPY FROM CLIPBOARD routine is called to copy the data to the clipboard. For arguments to this routine, you specify the name of the buffer into which the clipboard data will be copied, the length of the buffer, and the format in which you want the data. (For information about specifying format names, see Section 13.2.4.) In the final two arguments, the COPY FROM CLIPBOARD routine returns the number of bytes copied from the clipboard and any private data associated with the data on the clipboard.

The COPY FROM CLIPBOARD returns a value that indicates success (ClipboardSuccess). If the COPY FROM CLIPBOARD returns the value ClipboardTruncate, which indicates that more data is available on the clipboard, another call to COPY FROM CLIPBOARD specifying the same format will copy data from the clipboard starting where the last call stopped.

- ❺ The paste operation is ended by calling the END COPY FROM CLIPBOARD routine. The END COPY FROM CLIPBOARD routine unlocks the clipboard that had previously been locked by the call to the START COPY FROM CLIPBOARD routine.

Using the Cut and Paste Routines

13.3 Copying Data from the Clipboard

Your application should allow the user to indicate where the data is to be placed on the window. This may be implicit; for example, text data may be automatically pasted at the current text cursor location.

13.4 Inquiring About Clipboard Contents

Before copying data to or from the clipboard, you may want to examine its contents. For example, in a paste operation, you may be looking for data in a specific format. Using the cut and paste inquire routines, you can obtain the following information about the contents of the clipboard:

- The number of formats in which the data item on the clipboard is available
- The names of the formats in which the data item on the clipboard is available
- The length of the data item in a specified format

Use the `INQUIRE NEXT PASTE COUNT` routine to find out the number of formats in which the data item on the clipboard is available. This routine returns the number of formats and the length of the longest format name. (You use the length information returned by this routine to determine the size of the buffer that you will provide to the `INQUIRE NEXT PASTE FORMAT` routine. This ensures that your buffer is large enough to fit any of the format names available.)

Use the `INQUIRE NEXT PASTE FORMAT` routine to find out the name of a particular data format in which the data item on the clipboard is available. You specify the format that you are inquiring about by number. If you specify a number that is greater than the total number of formats available, the `INQUIRE NEXT PASTE FORMAT` routine returns 0 (zero) in the `copied_len` argument. The routine returns the name of the format as a character string in the `format_name_buf` argument and returns the number of bytes in the format name string in the `copied_len` argument. (When using the C programming language, you must also specify the size of the buffer in the `buffer_len` argument.)

Use the `INQUIRE NEXT PASTE LENGTH` routine to find out the length of the data item on the clipboard. You specify the format by name, and the routine returns the length in bytes.

You typically use the values returned by the inquire routines as arguments to other cut and paste routines, such as the `COPY FROM CLIPBOARD` routine. To ensure that the values returned are still valid when you make these routine calls, lock the clipboard between the call to the inquire routine and calls to the other routines. For example, if you are inquiring about data that you intend to copy from the clipboard, call the inquire routines after locking the clipboard with the `START COPY FROM CLIPBOARD` routine. (You can use the obsolete `CLIPBOARD LOCK` routine to lock the clipboard. If you use this routine, be sure to unlock the clipboard using the `CLIPBOARD UNLOCK` routine after you have completed your clipboard operation.)

Using the Cut and Paste Routines

13.5 QuickCopy Implementation

13.5 QuickCopy Implementation

The QuickCopy function allows users to copy data between applications without first having to copy the data to the clipboard and then paste it in their chosen application. The QuickCopy function acts as an accelerator for the copy function in the Edit menu.

Table 13–3 describes the QuickCopy operations.

Table 13–3 QuickCopy Operations

Operation	Description
CopyFrom	The user presses and holds MB3, drags the pointer cursor to select the area to copy, and releases MB3. This creates a secondary selection (independent of any clipboard selection) that is copied to the window with input focus. Before initiating the CopyFrom operation, the user should make sure that the window to receive the data has the input focus.
CopyTo	Using one of the primary selection mechanisms (dragging MB1, Select All, and so on), the user creates a primary selection. The user then moves the pointer cursor to the desired location and clicks MB3 to copy the primary selection.
MoveFrom	The user presses and holds Ctrl/MB3, drags the pointer cursor to select the area to copy, and releases the Ctrl/MB3 combination. This creates a secondary selection that is inserted into the active input position and removed from the original location. Before initiating the MoveFrom operation, the user should make sure that the window receiving the data has input focus.
MoveTo	Using one of the primary selection mechanisms (dragging MB1, Select All, and so on), the user creates a primary selection. The user then moves the pointer cursor to the desired location and presses and releases Ctrl/MB3 to insert the primary selection into the indicated position and remove it from the original location.

13.5.1 QuickCopy Message Types

Because users can use the QuickCopy function to copy data between applications, applications need a mechanism for telling other applications that data is available.

When an application receives an MB3 Button Release event from a CopyFrom operation, it sends a STUFF_SELECTION client message event to the window that has the input focus. The data field of the client message structure should contain the atom XA_SECONDARY.

In a MoveTo operation, the application that receives the data must tell the owner of the selection when to delete the data, because the Ctrl/MB3 combination that identifies a move operation is seen only by the receiving application. To notify the sender of the data that you have received the data, send a KILL_SELECTION client message event to the owner of the primary selection. Set the *data* member of the client message structure to the atom XA_PRIMARY. The owner of the primary selection deletes the data when it receives a KILL_SELECTION message.

13.5.2 Selection Threshold Resource

When an application receives an MB3 Button Press event, the application does not know whether the user intends to drag the mouse and perform a CopyFrom operation, or to release the button and perform a CopyTo operation.

Applications can use the **selection threshold** to specify the number of pixels that the pointer must cross in order for the application to treat the drag operation as intentional.

When your application receives the MB3 Button Press event, record the pointing device coordinates and compare them with the MB3 Button Release event or pointing device motion events. If the pointing device has moved more than the number of pixels specified in the selection threshold, treat the drag operation as intentional.

The resource name should be **selectionThreshold**. The default selection threshold value should be 5 pixels.

13.5.3 Implementing the QuickCopy Function

The four QuickCopy operations are similar in that they allow users to copy or move data between applications without having to use the clipboard. However, there are some differences in how the four operations function.

The CopyFrom and MoveFrom operations affect the secondary selection, which means that the user uses MB3 or Ctrl/MB3 to make the selection. The CopyFrom and MoveFrom operations have the window with the input focus as their destination.

The CopyTo and MoveTo operations use the primary selection, which means that the user uses MB1 to make the selection, as in a typical cut and paste operation. The user can then choose the destination for the data.

The CopyTo and MoveTo operations do not use the STUFF_SELECTION message to notify the application that is to receive the data.

13.5.3.1 CopyFrom and MoveFrom Operations

The following sample illustrates the CopyFrom and MoveFrom operations:

- 1 The user presses MB3 and drags the pointer cursor.

Your application uses the pointer screen location to find out what data is affected and highlights that data. If the user presses Ctrl/MB3, your application recognizes that this is a MoveFrom operation.

The simple text widget is sensitive to MB3 events within its borders. If the user clicks MB3 within the boundaries of the simple text widget, the widget highlights its contents.

- 2 When the user releases MB3 (or Ctrl/MB3) and your application receives the Button Release event, your application calls the OWN_SELECTION routine with the **selection** argument set to XA_SECONDARY to indicate ownership of the secondary selection atom as shown in Example 13-3.

Using the Cut and Paste Routines

13.5 QuickCopy Implementation

Example 13-3 Calling the OWN SELECTION Routine

```
.  
. .  
. .  
    XtOwnSelection( w, XA_SECONDARY, time, convert_proc,  
                  lose_selection, NULL)  
. .  
. .
```

The **convert_proc** argument is a callback routine for the XUI Toolkit to call when an application requests the current value of the selection. The **convert_proc** argument is described in Example 13-6.

The **notify_proc** argument, passed as null in Example 13-3, specifies the routine to call after the requesting application has received the selection.

The **lose_selection** argument is a routine to be called when the widget has lost selection ownership. Widgets can lose selection ownership if another widget later asserts ownership of the selection or if the original widget voluntarily gives up ownership.

- 3 Your application notifies the receiving application that the QuickCopy data is available, as shown in Example 13-4.

Example 13-4 Notifying the Receiving Application that Data Is Available

```
.  
. .  
. .  
static void quick_copy(w, event)  
Widget w;  
XEvent *event;  
{  
  ❶ XClientMessageEvent cm;  
    int revert;  
  
    cm.type = ClientMessage;  
    cm.display = XtDisplay(w);  
  ❷ cm.message_type = XInternAtom(XtDisplay(w), "STUFF_SELECTION",  
                                  FALSE);  
  ❸ XGetInputFocus(XtDisplay(w), &cm.window, &revert);  
  ❹ cm.format = 32;  
  ❺ cm.data.l[0] = XA_SECONDARY;  
  ❻ cm.data.l[1] = event->xbutton.time;  
  ❼ XSendEvent(XtDisplay(w), cm.window, TRUE, NoEventMask, &cm);  
}
```

- ❶ The XUI Toolkit uses client message events to transfer messages about the QuickCopy selection. Your application QuickCopy code must be able to accommodate these client messages.

Using the Cut and Paste Routines

13.5 QuickCopy Implementation

- ② The XUI Toolkit uses the STUFF_SELECTION message to notify applications that there is data available for them to insert. The STUFF_SELECTION message is associated with the XA_SECONDARY atom.

Your application needs to identify the atom identifier for the STUFF_SELECTION message in the *message_type* member of the client message structure. The atom identifier can be different for each invocation of the server. Therefore, your application calls the Xlib INTERN_ATOM routine to determine the atom identifier for the STUFF_SELECTION message.

The Xlib INTERN_ATOM routine returns the atom identifier for the *atom_name* argument, which in this case is STUFF_SELECTION. The *only_if_exists* argument, in this case false, creates an atom identifier for an atom if one does not already exist.

See the *VMS DECwindows Xlib Programming Volume* for more information about the INTERN_ATOM routine.

- ③ Because the QuickCopy data is copied to the window with the input focus, the application calls the Xlib GET_INPUT_FOCUS routine to get the input focus. In this example, the input focus is returned to the *window* member of the client message structure.

See the *VMS DECwindows Xlib Programming Volume* for more information about the GET_INPUT_FOCUS routine.

- ④ The data associated with this property is stored in 32-bit format.
- ⑤ The first field of the *data* member of the client message structure identifies the XA_SECONDARY atom.
- ⑥ The second field of the *data* member of the client message structure identifies the time at which the MB3 Button Release event occurred.
- ⑦ The application sends the STUFF_SELECTION client message event, using the Xlib SEND_EVENT routine, to the window that has input focus.

- 4 The receiving application gets the STUFF_SELECTION message and calls the INTERN_ATOM routine to determine its message type.

The receiving application knows that it is being asked to insert some data and calls the GET_SELECTION_VALUE routine to get the data, as shown in Example 13-5.

Using the Cut and Paste Routines

13.5 QuickCopy Implementation

Example 13-5 Getting the Selection Value

```
.
.
static void get_selection(w, event)
Widget w;
XEvent *event;
{
    XClientMessageEvent *cm = (XClientMessageEvent *)event;
    if(event->type != ClientMessage)
        return;
    if(cm->message_type !=
        XInternAtom(XtDisplay(w), "STUFF_SELECTION", FALSE))
        return;
    ❶ XtGetSelectionValue(w, XA_SECONDARY, XA_STRING, stuff_proc, 0,
        cm->data.l[1]);
}
.
.
```

- ❶ This call gets the selection value for the secondary selection. The **target** argument, in this case **XA_STRING**, indicates that the selection value should be returned as an **XA_STRING** atom.

As part of the call to the GET SELECTION VALUE routine, the receiving application passes in a callback routine to call, in this case **stuff_proc**, when the selection value has been obtained. The **cm->data.l[1]** argument identifies the time at which the MB3 Button Release event occurred.

- 5 The XUI Toolkit gets the GET SELECTION VALUE request. The server already knows which widget owns the secondary selection because of the prior call to OWN SELECTION. The **convert_proc** argument of OWN SELECTION specifies an application-specific callback routine for the XUI Toolkit to call to get the data, as shown in Example 13-6.

The **convert_proc** callback routine cancels the secondary selection in addition to transferring the data.

Example 13-6 Getting the Secondary Selection Data

```
.
.
static Boolean convert_proc(w, selectionp, desiredtypep,
                           typep, value, lengthp, formatp)
①   Widget w;
    Atom *selectionp;
    Atom *desiredtypep;
②   Atom *typep;
    caddr_t *value;
    int *lengthp;
    int *formatp;
{
    char *ptr;
    int line, column;
    int type;

    if(*selectionp == XA_PRIMARY ) {
        type = 0;
    } else {
        if(*selectionp == XA_SECONDARY )
            type = 1;
        else
            return(FALSE);
    }
    if(! Source_has_selection(w, type))
        return (FALSE);

    if (*desiredtypep == 1) {
        *typep = 1;
    } else {
        *typep = XA_STRING;
    }
}
.
.
.
/* Allocate a value buffer with XtMalloc */
/* Fill in the buffer */
/* Set the length of the buffer */
/* If secondary selection, clear selection */
.
.
.
```

-
- ① The XUI Toolkit indicates that it wants the XA_SECONDARY selection that this widget owns. The **desiredtypep** argument specifies the desired target atom type, in this case XA_STRING.
 - ② The **typep** argument returns the atom type of the secondary selection. The **value** argument returns the selection data. The **lengthp** and **formatp** arguments return the length and format of the selection data.

Using the Cut and Paste Routines

13.5 QuickCopy Implementation

- 6 When the XUI Toolkit gets the data for the QuickCopy procedure, it calls the **callback** argument that the receiving application passed in the GET SELECTION VALUE routine, in this case **stuff_proc**, as shown in Example 13–7. Your application is responsible for displaying the data.

Example 13–7 QuickCopy Callback Routine

```
.
.
static void stuff_proc(w, closure, selectionp, typep, value,
                      lengthp, formatp)
Widget w;
Opaque closure;
①Atom *selectionp;
②Atom *typep;
③char *value;
④int *lengthp;
⑤int *formatp;
{
.
.
/* Insert data */
.
.
.
```

- ① Specifies the primary or secondary atom.
- ② A pointer to the selection type returned from **convert_proc**.
- ③ A pointer to the selection data returned from **convert_proc**.
- ④ A pointer to the selection length returned from **convert_proc**.
- ⑤ A pointer to the selection format returned from **convert_proc**.

13.5.3.2 CopyTo and MoveTo Operations

The following sample illustrates the CopyTo and MoveTo operations:

- 1 The user presses MB1 and drags the pointer cursor.
Your application uses the pointer screen location to find out what data is affected and highlights that data.
- 2 When the user releases MB1 and your application receives the Button Release event, your application calls the OWN SELECTION routine with the **selection** argument set to XA_PRIMARY to indicate ownership of the primary selection atom.
Your application provides callback routines for the XUI Toolkit to call when an application requests the current value of the selection and after the requesting application has received the selection.
- 3 The user decides where to insert the data and clicks either MB3 to copy the data or Ctrl/MB3 to move the data.

Using the Cut and Paste Routines

13.5 QuickCopy Implementation

When the receiving application gets the MB3 or Ctrl/MB3 Button Release event, the application knows that it is being asked to insert some data and calls the GET SELECTION VALUE routine to get the primary selection data.

4 The XUI Toolkit gets the GET SELECTION VALUE request.

Because of the prior call to OWN SELECTION, the server already knows which application owns the primary selection and the routine for the XUI Toolkit to call to get the primary selection data.

5 When the XUI Toolkit gets the data for the QuickCopy procedure, the toolkit calls the callback routine that the receiving application passed in the GET SELECTION VALUE routine.

6 The receiving application copies the data.

If this were a MoveTo operation, the receiving application would send the sending application the atom identifier for a KILL_SELECTION message. The atom identifier is of type XA_PRIMARY.

The receiving application needs to identify the atom identifier for the KILL_SELECTION message in the *message_type* member of the client message structure. As with the STUFF_SELECTION message, the atom identifier can be different for each invocation of the server. Therefore, your application calls the Xlib INTERN_ATOM routine to determine the atom identifier, as shown in Example 13-8.

Example 13-8 Sending a KILL_SELECTION Message

```
.
.
.
static void send_kill(w, event)
    Widget w;
    XEvent *event;
{
XClientMessageEvent cm;
    int revert;

    cm.type = ClientMessage;
    cm.display = XtDisplay(w);
    cm.message_type = XInternAtom(XtDisplay(w), "KILL_SELECTION", FALSE);

    ❶ cm.window = XGetSelectionOwner(XtDisplay(w), XA_PRIMARY);
    cm.format = 32;
    cm.data.l[0] = XA_PRIMARY;
    cm.data.l[1] = event->xbutton.time;
    ❷ XSendEvent(XtDisplay(w), cm.window, TRUE, NoEventMask, &cm);
}
.
.
.
```

Using the Cut and Paste Routines

13.5 QuickCopy Implementation

- ① The receiving application calls the Xlib `GET SELECTION OWNER` routine to get the identifier of the resource that owns the primary selection.

See the *VMS DECwindows Xlib Programming Volume* for more information about the `GET SELECTION OWNER` routine.

- ② The receiving application sends an event with the atom identifier for a `KILL_SELECTION` message.

14 Communicating with the Window Manager

This chapter describes how your application interacts with the window manager, and includes information about the following topics:

- Making requests of the window manager
- Retrieving information about window manager restrictions

You should write your application to run effectively under a variety of window managers. The chapter uses the DECwindows window manager as an example and includes specific information about this window manager.

The chapter describes how to communicate with the window manager using the XUI Toolkit and Xlib routines.

14.1 Overview

Different applications running on the same system simultaneously might have different requirements for display space and other resources. Window manager programs enable users to manipulate windows on the display and thereby control the final layout of the screen. Users can move windows on the display, resize windows, change the stacking order of windows, shrink windows to icons, and expand windows from icons.

For example, in the DECwindows environment, a user can change the size of a window by clicking on the resize button in the title bar at the top of each window. Once the resize button is activated, the user can expand or contract the boundaries of the window using the pointer cursor. (The *VMS DECwindows User's Guide* describes this and other functions provided by the DECwindows window manager.)

Your application can request a desired size or location from the window manager. Section 14.2 describes how your application makes requests, called **hints**, of the window manager. The window manager can grant the request, ignore the request, or provide a compromise.

Although you can write an application that bypasses the window manager, users cannot move or resize the application, nor can they shrink it to an icon. (See Section 14.7.6 for more information about bypassing the window manager in your application.)

14.2 Making Requests of the Window Manager

Your application can communicate its requests to the window manager in two ways:

- Using window properties (predefined and vendor specific)
- Using shell widget attributes

Communicating with the Window Manager

14.2 Making Requests of the Window Manager

You must use Xlib routines to communicate using window properties. If your application is written using the XUI Toolkit, you can use shell widget attributes. The shell widget attributes hide some of the complexity of working with window properties and, in most cases, provide identical capabilities.

14.2.1 Using Window Properties

Your application can communicate with the window manager by setting properties on the window associated with the top-level widget of your application widget hierarchy (each widget has an associated window). A property is data associated with a particular window; every X window can have properties associated with it. Every property has a name, a data type, and an identifier (the identifier is known as an **atom**). (For more information on window properties, see *VMS DECwindows Xlib Programming Volume*.) The window manager reads the properties you place on the top-level window to get information from your application. Your application reads the properties of the root window to get information from the window manager. (The root window is the window that covers the entire screen.)

You can use two types of properties to communicate with the window manager:

- Predefined window properties
- Vendor-specific window properties

14.2.1.1 Predefined Window Properties

The predefined window properties are part of the X Window System, Version 11, standard. These properties enable you to make commonly needed requests of the window manager, such as the following:

- Associate a name with the top-level window of your application
- Specify the initial size and screen location of your application
- Specify the pixmap used as the icon for your application
- Customize other aspects of the appearance and behavior of your application

By convention, the names of the predefined properties begin with the characters WM_. For example, the property you use to associate a name with your application window is called WM_NAME. Property names are case sensitive. The predefined window manager properties use all uppercase characters. Table 14-1 lists the predefined window manager properties.

Communicating with the Window Manager

14.2 Making Requests of the Window Manager

Table 14–1 Predefined Window Manager Properties

Property	Data Type	Description
WM_NAME	STRING	Specifies the name you want to associate with a window.
WM_ICON_NAME	STRING	Specifies the name you want to display in the icon associated with your application.
WM_NORMAL_HINTS	WM_SIZE_HINTS	Specifies the size of a window in its normal state.
WM_ZOOM_HINTS ¹	WM_SIZE_HINTS	Specifies the size of a window in its zoomed state.
WM_HINTS	WM_HINTS	Specifies information about the initial state of your application, the pixmap used as the icon, the position of the icon, and other aspects of your application.
WM_COMMAND	STRING	Specifies the command that starts your application.
WM_ICON_SIZE	WM_ICON_SIZE	Lists the icon sizes supported by the window manager.
WM_CLASS	STRING	Specifies the name of an instance of your application and its class name.
WM_TRANSIENT_FOR	WINDOW	Indicates that a window, such as a dialog box, is transient.

¹Not supported by the DECwindows window manager.

Each property listed in Table 14–1 has an associated data type. For example, the data type of the WM_NAME property is STRING. (Data type names are also case sensitive.) The data types of some of the predefined properties are data structures. For example, Xlib defines the data type of the WM_HINTS property as the WM Hints data structure. The following example illustrates the WM Hints data structure. To see the definitions of all the predefined window properties, see the *VMS DECwindows Xlib Programming Volume*.

```
typedef struct {
    long flags;
    Bool input;
    int initial_state;
    Pixmap icon_pixmap;
    Window icon_window;
    int icon_x, icon_y;
    Pixmap icon_mask;
    XID window_group;
} XWmHints;
```

Communicating with the Window Manager

14.2 Making Requests of the Window Manager

Table 14–2 defines the members of the WM Hints data structure.

Table 14–2 Members of the WM Hints Data Structure

Member Name	Contents												
flags	Specifies the members of the data structure that are defined.												
input	Indicates whether the client relies on the window manager to get keyboard input.												
initial_state	Defines how the window should initially appear. Possible initial states are: <table border="1"><thead><tr><th>Constant Name</th><th>Description</th></tr></thead><tbody><tr><td>DontCareState</td><td>Application can start up in any state.</td></tr><tr><td>NormalState</td><td>Main application window is mapped.</td></tr><tr><td>ZoomState¹</td><td>Window starts in zoomed state.</td></tr><tr><td>IconicState</td><td>Main application window is not mapped.</td></tr><tr><td>InactiveState</td><td>Application appears as option in a menu.</td></tr></tbody></table>	Constant Name	Description	DontCareState	Application can start up in any state.	NormalState	Main application window is mapped.	ZoomState ¹	Window starts in zoomed state.	IconicState	Main application window is not mapped.	InactiveState	Application appears as option in a menu.
Constant Name	Description												
DontCareState	Application can start up in any state.												
NormalState	Main application window is mapped.												
ZoomState ¹	Window starts in zoomed state.												
IconicState	Main application window is not mapped.												
InactiveState	Application appears as option in a menu.												
icon_pixmap	Identifies the pixmap used to create the window icon.												
icon_window	Identifies the window to be used as an icon.												
icon_x	Specifies the initial x-coordinate of the icon.												
icon_y	Specifies the initial y-coordinate of the icon.												
icon_mask	Specifies the pixels of the icon pixmap used to create the icon.												
window_group	Specifies that the window belongs to a group of other windows.												

¹Not supported by the DECwindows window manager.

14.2.1.2 Vendor-Specific Window Properties

Vendors that create window managers can extend the set of predefined window manager properties to enable applications to communicate with their window managers. You use these additional properties to specify values for the additional capabilities provided by the vendor's window manager. For example, the DECwindows window manager enables you to specify the icon it displays in the title bar of your application.

By convention, vendors distinguish the name of their window manager properties with some identifying prefix. For example, the names of properties specific to the DECwindows window manager begin with the characters DEC_WM_. Table 14–3 lists the properties defined by the DECwindows window manager.

Communicating with the Window Manager

14.2 Making Requests of the Window Manager

Table 14-3 Properties Defined by the DECwindows Window Manager

Property	Data Type	Description
DEC_WM_HINTS	DEC_WM_HINTS	Specifies the shrink-to-icon button pixmap, position of the icon in the icon box, and appearance of the title bar.
DEC_WM_DECORATION _GEOMETRY ¹	DEC_WM_DECORATION _GEOMETRY	Specifies the font the DECwindows window manager uses in the title bar and icon and specifies the sizes it supports for the shrink-to-icon button and other aspects of the title bar.

¹The application should not attempt to set this property. The DECwindows window manager uses this property to communicate with your application.

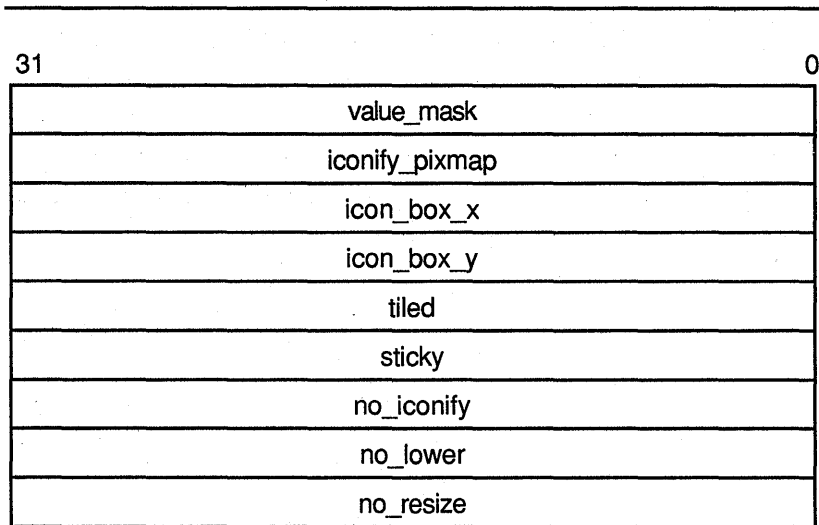
The data type of the DEC_WM_HINTS property is a DEC WM Hints data structure. The following illustrates the DEC WM Hints data structure; Figure 14-1 illustrates the VAX definition of this data structure. Note the distinction between the DEC_WM_HINTS property, which contains vendor-specific information, and the WM_HINTS property, which specifies information predefined by the Xlib standard.

```
typedef struct {
    unsigned long value_mask;
    Pixmap iconify_pixmap;
    int icon_box_x;
    int icon_box_y;
    Bool tiled;
    Bool sticky;
    Bool no_iconify_button;
    Bool no_lower_button;
    Bool no_resize_button;
} DECWmHintsRec, *DECWmHints;
```

Communicating with the Window Manager

14.2 Making Requests of the Window Manager

Figure 14–1 DEC WM Hints Data Structure (VAX Binding)



ZK-1317A-GE

Table 14–4 defines the members of the DEC WM Hints data structure.

Table 14–4 Members of the DEC WM Hints Data Structure

Member	Contents																		
value_mask	Specifies the members of the data structure that are defined. Possible values are: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Constant Name</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>DECWmIconifyPixmapMask</td> <td>Iconify_pixmap member is defined.</td> </tr> <tr> <td>DECWmIconBoxXMask</td> <td>Icon_box_x member is defined.</td> </tr> <tr> <td>DECWmIconBoxYMask</td> <td>Icon_box_y member is defined.</td> </tr> <tr> <td>DECWmTiledMask</td> <td>Tiled member is defined.</td> </tr> <tr> <td>DECWmStickyMask</td> <td>Sticky member is defined.</td> </tr> <tr> <td>DECWmNoIconifyButtonMask</td> <td>No_iconify_button member is defined.</td> </tr> <tr> <td>DECWmNoLowerButtonMask</td> <td>No_lower_button member is defined.</td> </tr> <tr> <td>DECWmNoResizeButtonMask</td> <td>No_resize_button member is defined.</td> </tr> </tbody> </table>	Constant Name	Description	DECWmIconifyPixmapMask	Iconify_pixmap member is defined.	DECWmIconBoxXMask	Icon_box_x member is defined.	DECWmIconBoxYMask	Icon_box_y member is defined.	DECWmTiledMask	Tiled member is defined.	DECWmStickyMask	Sticky member is defined.	DECWmNoIconifyButtonMask	No_iconify_button member is defined.	DECWmNoLowerButtonMask	No_lower_button member is defined.	DECWmNoResizeButtonMask	No_resize_button member is defined.
Constant Name	Description																		
DECWmIconifyPixmapMask	Iconify_pixmap member is defined.																		
DECWmIconBoxXMask	Icon_box_x member is defined.																		
DECWmIconBoxYMask	Icon_box_y member is defined.																		
DECWmTiledMask	Tiled member is defined.																		
DECWmStickyMask	Sticky member is defined.																		
DECWmNoIconifyButtonMask	No_iconify_button member is defined.																		
DECWmNoLowerButtonMask	No_lower_button member is defined.																		
DECWmNoResizeButtonMask	No_resize_button member is defined.																		
iconify_pixmap	Identifies the pixmap used to create the shrink-to-icon button in the title bar and, if the user specified use of small icons, the icon pixmap.																		
icon_box_x	Specifies the initial x-coordinate of the icon in the icon box. Specify this value in icon units.																		
icon_box_y	Specifies the initial y-coordinate of the icon in the icon box. Specify this value in icon units.																		

(continued on next page)

Communicating with the Window Manager

14.2 Making Requests of the Window Manager

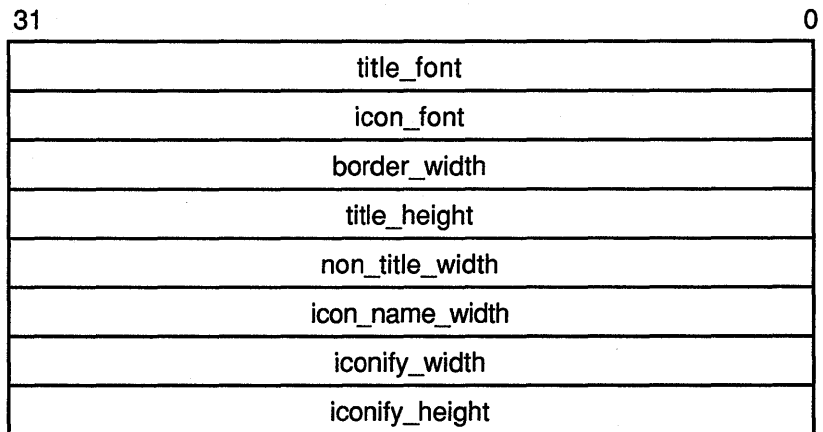
Table 14–4 (Cont.) Members of the DEC WM Hints Data Structure

Member	Contents
tiled	Specifies whether windows should overlap.
sticky	Specifies whether windows should remain in their place in the window stacking order.
no_iconify_button	Specifies whether the shrink-to-icon button should be present in the title bar.
no_lower_button	Specifies whether the push-to-back button should be present in the title bar.
no_resize_button	Specifies whether the resize button should be present in the title bar.

The data type of the DEC_WM_DECORATION_GEOMETRY property is the WM Decoration Geometry data structure. The following illustrates the WM Decoration Geometry data structure; Figure 14–2 illustrates the VAX definition of this data structure. You cannot specify values for the members of this property; the window manager uses this property to communicate information to applications.

```
typedef struct {
    Font title_font;
    Font icon_font;
    int border_width;
    int title_height;
    int non_title_width;
    int icon_name_width;
    int iconify_width;
    int iconify_height;
} WmDecorationGeometryRec, *WmDecorationGeometry;
```

Figure 14–2 WM Decoration Geometry Data Structure (VAX Binding)



ZK–1318A–GE

Table 14–5 defines the members of the WM Decoration Geometry data structure.

Communicating with the Window Manager

14.2 Making Requests of the Window Manager

Table 14–5 Members of the WM Decoration Geometry Data Structure

Member	Contents
title_font	Identifies the font in which the window manager displays the name you associate with the top-level window of your application.
icon_font	Identifies the font in which the window manager displays the name you associate with your application icon.
border_width	Specifies the width of the border with which the window manager surrounds your top-level window.
title_height	Specifies the height of the title bar the window manager displays at the top of your application's top-level window.
non_title_width	Specifies the width of the title bar, not including the width of the title area. This includes the left and right border widths, the width of the shrink-to-icon button, and the push-to-back and resize buttons and the borders around each of these buttons.
icon_name_width	Specifies the width of the icon name.
iconify_width	Specifies the width of the shrink-to-icon button in the title bar.
iconify_height	Specifies the height of the shrink-to-icon button in the title bar.

14.2.2 Using Shell Widget Attributes

As an alternative to using window properties, you can communicate with a window manager by using shell widget attributes. The shell widget is always the top-level widget of an application. (Section 2.3.1 describes the application widget hierarchy.)

With shell widget attributes, you can communicate with the window manager the same way you assign values to the attributes of any other XUI Toolkit widget: by using an argument list. Section 14.5 describes how to communicate with the window manager using shell widget attributes.

The XUI Toolkit defines shell widget attributes that allow you to set both standard and vendor-specific window manager properties. The *VMS DECwindows Toolkit Routines Reference Manual* describes the class hierarchy of the shell widgets and lists all the attributes supported by the shell widgets.

14.3 Setting and Retrieving Predefined Window Manager Properties

To set or retrieve the value of a predefined window manager property, use one of the routines that Xlib provides for performing these tasks. For example, to specify the text string that the window manager displays in your application's icon, use the SET ICON NAME Xlib routine. Table 14–6 lists these routines. For more information about these routines, see the *VMS DECwindows Xlib Routines Reference Manual*.

Communicating with the Window Manager

14.3 Setting and Retrieving Predefined Window Manager Properties

Table 14–6 Xlib Routines for Setting and Retrieving Predefined Window Manager Properties

Property	Set Routine	Retrieve Routine
WM_NAME	STORE NAME	FETCH NAME
WM_ICON_NAME	SET ICON NAME	GET ICON NAME
WM_HINTS	SET WM HINTS	GET WM HINTS
WM_NORMAL_HINTS	SET NORMAL HINTS	GET NORMAL HINTS
WM_CLASS	SET CLASS HINT	GET CLASS HINT
WM_ICON_SIZE	SET ICON SIZES ¹	GET ICON SIZES
WM_TRANSIENT_FOR	SET TRANSIENT FOR HINT	GET TRANSIENT FOR HINT

¹Applications should only read this property; the window manager sets the value of this property to provide information to applications.

When you use one of these routines to set a window manager property whose data type is a data structure, the routine changes the values of all the members of the data structure, not just the value of the members for which you have specified values. To change the value of a member of a window manager property data structure, first retrieve the current value of the property (using the appropriate Xlib routine) and modify the member of the data structure you want to change. Then assign the modified data structure as the value of the property.

Example 14–1 illustrates how to use Xlib routines to set the value of the WM_HINTS and WM_NAME predefined properties. Note that the WM_HINTS property is a data structure.

Example 14–1 Assigning Values to Predefined Window Manager Properties

```
①#include <decw$include/Xutil.h>
#include <decw$include/Xatom.h>
.
.
.
②XWMHints wmhints;
.
.
.
③wmhints.icon_pixmap = XCreatePixmapFromBitmapData(dpy, root, checker32_bits,
checker32_width, checker32_height, fg, bg, depth);
wmhints.flags = IconPixmapHint;
④XSetWMHints(dpy, win, &wmhints);
⑤XStoreName(dpy, win, "Checkers");
.
.
.
```

① The example includes two Xlib symbol definition files. The Xutil.h file defines the contents of the property data structures. The Xatom.h file contains the declarations of the atoms used to refer to the properties.

Communicating with the Window Manager

14.3 Setting and Retrieving Predefined Window Manager Properties

- ② The example declares a variable, named *wmhints*, of the data type WM Hints data structure. This data structure holds the values to be assigned to the WM_HINTS property.
- ③ In the next two statements, the example assigns values to members of the data structure. First, the example assigns the identifier of a pixmap as the value of the appropriate member of the data structure. Second, the example sets the flag member of the WM Hints data structure to indicate which member of the data structure has been assigned a value.
- ④ After setting up the data structure with values, the example uses the SET WM HINTS Xlib routine to change the value of the WM_HINTS property. The filled-in WM Hints data structure is passed to the routine.
- ⑤ To associate a name with its main application window, the example uses the STORE NAME Xlib routine. This routine assigns the string passed as an argument to the routine as the value of the WM_NAME property. Do not use compound strings for this text.

14.4 Setting and Retrieving Vendor-Specific Window Manager Properties

As with predefined window manager properties, vendors can provide support routines that you can use to set the value of the properties they define. If the vendor does not provide support routines, you must use the generic property manipulation Xlib routine, CHANGE PROPERTY, to set a vendor-specific property. Example 14-2 illustrates how to set the value of the DEC WM Hints property to specify the icon that appears in the title bar.

Example 14-2 Setting Vendor-Specific Window Manager Properties

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>
#include <decw$include/Xatom.h>
#include <decw$include/decwmhints.h>
#include <stdio.h>
.
.
.
①DECWmHints dwmhints;
   DECWmHintsRec *h = &dwmhints;
②Atom wmatom;
.
.
.
③wmatom = XInternAtom(dpy, "DEC_WM_HINTS", 0);
④if (wmatom != None)
{
   h->value_mask = DECWmIconifyPixmapMask;
   h->iconify_pixmap = XCreatePixmapFromBitmapData( dpy, root,
                                                    checker16_bits, checker16_width,
                                                    checker16_height, fg, bg, depth);
```

(continued on next page)

Communicating with the Window Manager

14.4 Setting and Retrieving Vendor-Specific Window Manager Properties

Example 14–2 (Cont.) Setting Vendor-Specific Window Manager Properties

```
⑤ XChangeProperty( dpy, win, wmatom, wmatom, 32, PropModeReplace,  
                  h, sizeof(DECWmHintsRec)/4 );  
    }  
    .  
    .  
    .
```

- ① The example declares a variable of the DEC WM Hints data structure, named *dwmhints*, and a pointer to the data structure, named *h*.
- ② The example declares a variable to hold the DECwindows window manager atom.
- ③ The example creates an atom called DEC_WM_HINTS using the INTERN ATOM Xlib routine.
- ④ If the example successfully creates the atom, it then assigns the pixmap identifier as the value of the appropriate member of the DEC WM Hints data structure. The example indicates which member of the data structure has been assigned a value in the value mask member.
- ⑤ The example uses the CHANGE PROPERTY Xlib routine to assign the pixmap as the value of the DEC_WM_HINTS property.

To read the value of a vendor-specific window manager property, use the GET WINDOW PROPERTY Xlib routine. For example, the DECwindows window manager provides information about the sizes of pixmaps it accepts for the shrink-to-icon button in the title bar, the height of the title bar, and other read-only information.

14.5 Setting and Retrieving Shell Widget Attributes

As with any XUI Toolkit widget, you assign values to shell widget attributes by using an argument list (Section 2.4.1.2 describes how to create an argument list). Unlike other XUI Toolkit widgets, certain shell widget attributes can be set only when you create the shell widget. Section 14.5.1 lists these attributes and describes how to set them. The remaining shell widget attributes can be set either when you create the shell widget or after it has been created. Section 14.5.2 describes how to set these attributes.

14.5.1 Setting Shell Widget Attributes at Widget Creation Time

The following are the shell widget attributes that can be assigned values only when you create the shell widget.

Communicating with the Window Manager

14.5 Setting and Retrieving Shell Widget Attributes

argc	input	width_inc
argv	geometry	height_inc
icon_x	min_width	min_aspect_x
icon_y	min_height	max_aspect_y
iconic	max_width	wm_timeout
initial_state	max_height	wait_for_wm

To assign a value to one of these attributes, you must use the `APPLICATION CREATE SHELL` intrinsic routine to create the widget. You cannot set shell widget attributes at creation time if you use the `INITIALIZE` intrinsic routine to create the top-level shell widget of your application. (To assign values to the attributes of a pop-up shell widget, pass an argument list to the `CREATE POPUP SHELL` intrinsic routine.)

Example 14-3 illustrates how to set shell widget attributes at widget creation time.

Example 14-3 Setting Shell Widget Attributes at Widget Creation Time

```
.
.
Display *display;
XtAppContext context;
Widget toplevel;
Arg arglist[25];
int ac;
.
.
.
❶ XtToolkitInitialize();

context = XtCreateApplicationContext();

display = XtOpenDisplay( context, "mynode::0", "appl_test",
                        "testclass", NULL, 0, &argc, &argv );

ac = 0;
❷ XtSetArg( arglist[ac], DwtNallowShellResize, TRUE ); ac++;
  XtSetArg( arglist[ac], DwtNx, 150 ); ac++;
  XtSetArg( arglist[ac], DwtNy, 150 ); ac++;

❸ toplevel = XtAppCreateShell( "Appl Test", "testclass",
                             applicationShellWidgetClass,
                             display, arglist, ac);
.
.
.
```

- ❶ The example calls the `TOOLKIT INITIALIZE` intrinsic routine to initialize the XUI Toolkit and then calls the `DISPLAY` intrinsic routine to open a connection to the display device. The example also calls the `CREATE APPLICATION CONTEXT` routine to create the application context.
- ❷ After initializing the XUI Toolkit and opening the connection to the display, the example creates an argument list in which it assigns values to shell widget attributes. The example sets the `allow_shell_resize` attribute to true, which instructs the shell widget to accept

Communicating with the Window Manager

14.5 Setting and Retrieving Shell Widget Attributes

resize requests from the application. The example also specifies its initial position in the `x` and `y` attributes.

- ③ The example then creates the shell widget using the `APPLICATION CREATE SHELL` intrinsic routine, passing the argument list to the routine. This routine returns the identifier of a shell widget.

14.5.2 Setting Shell Widget Attributes After Creation Time

To assign values to shell widget attributes after the widget has been created, use the `SET VALUES` intrinsic routine. Create an argument list in which you assign values to the shell widget attributes you want to set and then pass the argument list to the `SET VALUES` intrinsic routine. Example 14-4 shows how to use the `SET VALUES` intrinsic routine to assign values to shell widget attributes.

Example 14-4 Using the `SET VALUES` Intrinsic Routine to Set Shell Widget Attributes

```
.
.
.
①#include decw$include/vendor.h
.
.
.
②toplevel = XtInitialize("Hi","helloworldclass",NULL, 0, &argc, argv);
.
.
.
count = 0;
③XtSetArg( arglist[count], XtNallowShellResize, TRUE ); count++;
XtSetArg( arglist[count], XtNx, 150 ); count++;
XtSetArg( arglist[count], XtNy, 150 ); count++;
XtSetArg( arglist[count], XtNiconPixmap, IconPixmap ); count++;
XtSetArg( arglist[count], XtNiconifyPixmap, SmallIconPixmap ); count++;
④XtSetValues (toplevel, arglist, count );
.
.
.
```

- ① The XUI Toolkit symbol definition file `vendor.h` contains definitions of the vendor specific properties, such as the `iconify_pixmap` attribute.
- ② The `INITIALIZE` intrinsic routine is called to create the shell widget. This routine initializes the XUI Toolkit and returns the identifier of an application shell widget.
- ③ After creating the shell widget, the example creates an argument list in which it assigns values to shell widget attributes. The example sets the `allow_shell_resize` attributes to true, which instructs the shell widget to accept resize requests from the application, and specifies the initial screen location by its x-coordinate and y-coordinate. In addition, the example specifies the pixmaps it wants used in its icon and in the shrink-to-icon button in the title bar. The example assigns the pixmap identifiers as the value of the `icon_pixmap` and `iconify_pixmap` attributes.

Communicating with the Window Manager

14.5 Setting and Retrieving Shell Widget Attributes

- ④ The example assigns values to these attributes using the SET VALUES intrinsic routine.

14.6 Receiving Messages from the Window Manager

The preceding sections describe how to communicate with the window manager by setting and reading the values of window properties. However, some applications may need to know when the window manager changes the value of a property. For example, your application might want to know when it is iconified. Window managers tell your application when the value of a property changes by sending a client message event. To receive these events from a window manager, you must indicate that you want to receive property change events on the top-level widget of your application and on the root window.

If a window manager is not going to fulfill a request, it sends the WM_CONFIGURE_DENIED client message event.

If the window manager moves your main application window, it sends the WM_MOVED client message event to your application. The data in this event contains the new x-coordinate and y-coordinate of your window. Your application should not assume that requests to the window manager are complete until the window manager notifies your application that the operation completed. To receive these events, choose to receive StructureNotify events on your top-level window.

The DECwindows window manager uses the DEC_WM_TAKE_FOCUS client message event to instruct your application to set the input focus to one of its children. If your application previously had input focus, restore focus to the child that had input focus. Otherwise, either give the input focus to a child that can accept it or ignore the event.

If your application uses the XUI Toolkit, you do not need to choose explicitly to receive these events. The shell widget handles these events automatically. However, you can specify the amount of time your application will wait for notification from a window manager. Specify the time as the value of the `wm_timeout` attribute. If the window manager does not respond in the allotted time, the XUI Toolkit sets the value of the `wait_for_wm` attribute to false. Later events may reset this value.

14.7 Customizing Your Application Using Window Manager Hints

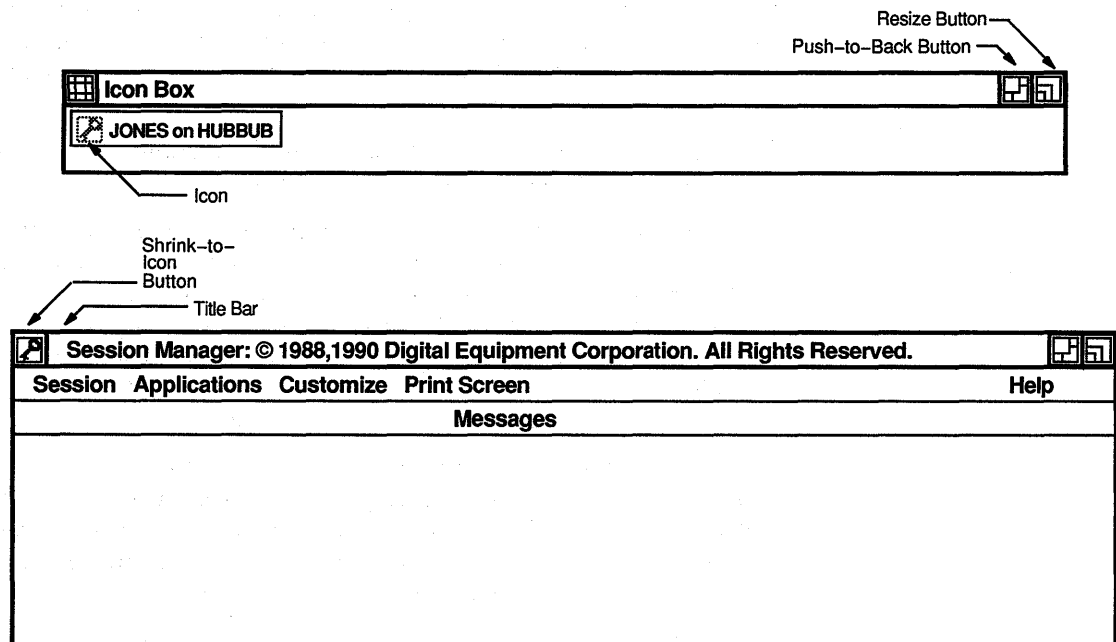
The following sections describe the aspects of your application that you can customize using window manager hints. The following sections use the DECwindows window manager as an example; other window managers might implement some of these functions differently. For example, the DECwindows window manager decorates each window with a title bar in which it displays the name that you associate with the window. Other window managers might use another mechanism to display the text you specify as the name. If you want to write an application that runs with any window manager, do not rely on the presence of vendor-specific features.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Figure 14-3 presents the screen appearance of an application running with the DECwindows window manager. The figure illustrates the two main areas you can customize: the main application window and your application icon. The DECwindows window manager decorates each window with a title bar that contains a shrink-to-icon button, the name of your window, a push-to-back button, and a resize button. In addition, the DECwindows window manager creates an icon for every application and displays the icons in the Icon Box.

Figure 14-3 Appearance of an Application Running Under the DECwindows Window Manager



ZK-1278A-GE

Table 14-7 lists the common tasks you can perform using hints to the window manager. The table also lists the shell widget attribute and window property you must use to perform the tasks.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Table 14–7 Common Tasks Performed with the Window Manager

Task	Shell Widget Attribute	Window Manager Property
Main Application Window		
Specify the name that appears in the title bar	title	WM_NAME
Position the application on display at startup	x y	NORMAL_HINTS
Specify the initial width and height of the application	width height	NORMAL_HINTS
Specify the minimum initial width and height of the application	min_width min_height	NORMAL_HINTS
Specify the maximum initial width and height of the application	max_width max_height	NORMAL_HINTS
Specify how much to increase the width or height when resized	width_inc height_inc	NORMAL_HINTS
Specify the aspect ratio of the display	min_aspect_x max_aspect_x min_aspect_y max_aspect_y	NORMAL_HINTS
Specify the pixmap used in the shrink-to-icon button in the title bar	iconify_pixmap	DEC_WM_HINTS
Specify to omit the shrink-to-icon button in the title bar	no_iconify_button	DEC_WM_HINTS
Specify to omit the push-to-back button in the title bar	no_restack_button	DEC_WM_HINTS
Specify to omit the resize button in the title bar	no_resize_button	DEC_WM_HINTS
Icon		
Associate a name with the application icon	icon_name	WM_ICON_NAME
Specify the pixmap used in the application icon	icon_pixmap	WM_HINTS
Create an application icon that appears nonrectangular	icon_mask	WM_HINTS
Replace the pixmap used in the application icon with a window	icon_window	WM_HINTS
Position the application icon in the Icon Box	icon_box_x icon_box_y	DEC_WM_HINTS

In addition to the attributes you can set, you can also read certain attributes to obtain information about aspects of the environment. The application cannot set these attributes; the window manager places information in these attributes. Table 14–8 lists the information that the window manager provides in these attributes.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Table 14–8 Information Provided by the Window Manager

Information	Shell Widget Attribute	Window Manager Property
Minimum width and height of an icon	No attribute	WM_ICON_SIZE
Maximum width and height of an icon	No attribute	WM_ICON_SIZE
Amount width and height of icon can be increased	No attribute	WM_ICON_SIZE
Font used by the window manager in the title bar	title_font	DEC_WM_DECORATION_GEOMETRY
Font used by the window manager in the application icon	icon_font	DEC_WM_DECORATION_GEOMETRY
Border width	border_width	DEC_WM_DECORATION_GEOMETRY
Height of the title bar	title_height	DEC_WM_DECORATION_GEOMETRY
Width of the title bar, not including the title	non_title_width	DEC_WM_DECORATION_GEOMETRY
Maximum width of the title in the icon	icon_name_width	DEC_WM_DECORATION_GEOMETRY
Maximum width and height of the shrink-to-icon button in the title bar	iconify_width iconify_height	DEC_WM_DECORATION_GEOMETRY
Current application state	icon_state	WM_STATE

14.7.1 Customizing Your Main Application Window

You can customize the following aspects of your main application window:

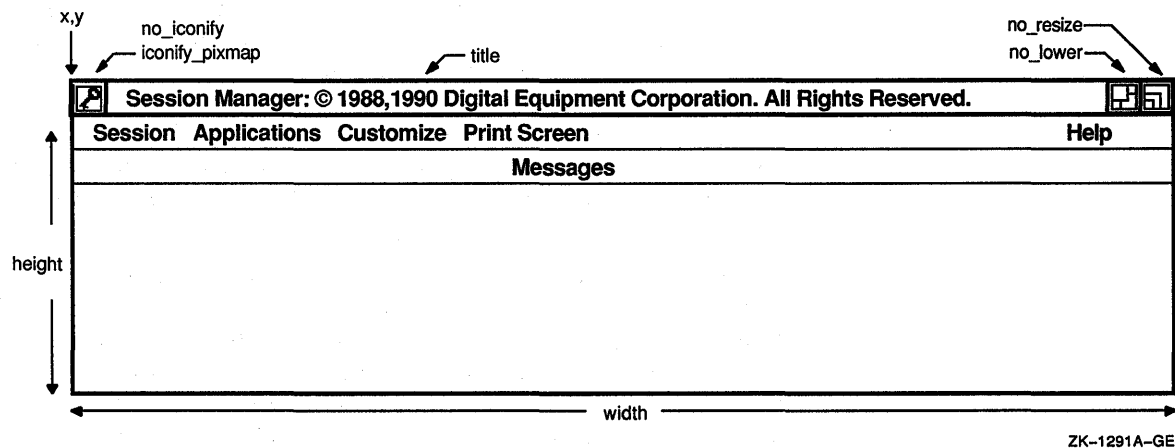
- Name associated with the window
- Initial size and position of the window
- Title bar

Figure 14–4 presents the main application window from Figure 14–3 and labels the parts you can customize with the name of the shell widget attribute or window property you would use.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Figure 14-4 Customizable Aspects of the Main Application Window



14.7.1.1 Associating a Name with Your Main Application Window

To associate a name with your main application window, assign the address of a text string as the value of the **title** shell widget attribute or the **WM_NAME** property. The **WM_NAME** property typically is used to communicate some information to the user that can change during the execution of your application. For example, you could display a file name in the title bar that changes as the user manipulates different files.

Using the XUI Toolkit, specify the name as an argument to the **APPLICATION CREATE SHELL** intrinsic routine. If your application uses the **INITIALIZE** intrinsic routine to create the top-level shell widget, pass the text you want the window manager to use as the name in the **name** argument to the routine. To change this text during execution of your application, use the **SET VALUES** intrinsic routine to pass a new value to the **title** attribute of the application shell widget. The window manager does not use compound strings for the name.

Using window properties, use the **STORE NAME** Xlib routine to assign a value to the **WM_NAME** property.

The DECwindows window manager displays the text in the title bar it places at the top of the window. The DECwindows window manager places no restrictions on the amount of text you can specify as the name. However, long names will be clipped by the size of the title area in the title bar. The window manager displays only a single line of text in the title bar.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

14.7.1.2 Specifying the Initial Size and Position of Your Application

To specify the initial size and position of your application, assign values to the **width**, **height**, **x**, and **y** attributes of the shell widget. Using properties, these attributes are members of the X Size Hints data structure, which is the data type of the WM_NORMAL_HINTS property. Use the SET NORMAL HINTS Xlib routine to assign values to this property. Specify these values in pixels; the x-coordinate and y-coordinate specify the position relative to the root window. Example 14-3 in Section 14.5.1 uses shell widget attributes to specify the initial position of the application on the display.

You can also specify both a range of appropriate sizes for your application and the increment to be used within this range. Specify the minimum width dimension in the **min_width** attribute and the maximum width in the **max_width** attribute. Similarly, use the **min_height** and **max_height** attributes to specify a range of heights. Specify the increment in the **width_inc** and **height_inc** attributes. For example, if you were writing a terminal emulator program, you might set the **height_inc** attribute to be the height of the font.

Your application should avoid resizing or repositioning itself. These functions should be under the user's control by way of the window manager. However, under some circumstances, self-resizing may be appropriate; for example, a bitmap editor may need to resize itself in order to accommodate a bigger bitmap that has just been read in from a file.

14.7.1.3 Customizing the Title Bar

You can customize certain aspects of the title bar that the window manager places at the top of the main window of your application. In addition to specifying the title bar text (described in Section 14.7.1.1), you can:

- Specify the pixmap used as the shrink-to-icon button in the title bar
- Specify whether or not the title bar should include a shrink-to-icon button, push-to-back button, or resize button

To specify the pixmap used in the shrink-to-icon button, pass the identifier of a pixmap as the value of the **iconify_pixmap** attribute. Using window properties, you would pass the identifier in the DEC_WM_HINTS property. If you do not specify a pixmap for the shrink-to-icon button, the window manager uses a pixmap of a paned window as the default. Example 14-5 illustrates how to specify the pixmap used for the shrink-to-icon button using window manager properties. (For an example of how to specify the pixmap used for the shrink-to-icon button using shell widget attributes, see Section 14.7.3.2.)

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Example 14-5 (Cont.) Specifying the Shrink-to-Icon Pixmap Using the CHANGE PROPERTY Xlib Routine

```
XStoreName(dpy, win, "Checkers");
XSetIconName(dpy, win, "Checkers");

⑤ wmatom = XInternAtom(dpy, "DEC_WM_HINTS", 0);
   if (wmatom != None) {
⑥     h->value_mask = DECWmIconifyPixmapMask;
       h->iconify_pixmap = XCreatePixmapFromBitmapData(dpy,
                                                       root, checker16_bits, checker16_width,
                                                       checker16_height, fg, bg, depth);
⑦     XChangeProperty(dpy, win, wmatom, wmatom, 32, PropModeReplace,
                       h, sizeof(DECWmHintsRec)/4);
   }
XMapWindow(dpy, win);
for (;;)
    XNextEvent(dpy, &event);
}
```

- ① The bits that make up the shrink-to-icon button pixmap, in X11 format.
- ② The example declares a variable, named *dwmhints*, of the DEC WM HINTS data structure, the data type of the DEC_WM_HINTS property. You use this property to specify the shrink-to-icon button pixmap.
- ③ The example declares an atom, named *wm_atom*. To use the DEC_WM_HINTS property, you must first define an atom for the structure.
- ④ The example creates the pixmap using the CREATE_PIXMAP_FROM_BITMAP_DATA Xlib routine. This routine returns a pixmap identifier. For more information about this routine, see the *VMS DECwindows Xlib Routines Reference Manual*.
- ⑤ To use the DEC_WM_HINTS property, the example must create an atom for the property. The example uses the INTERN_ATOM Xlib routine to create this atom. For more information about this routine, see the *VMS DECwindows Xlib Routines Reference Manual*.
- ⑥ If it creates the atom successfully, the example then assigns values to members of the DEC WM Hints data structure. In the example, the identifier of the small icon pixmap is assigned as the value of the *iconify_pixmap* member of the data structure. In addition, the example indicates which member of the data structure has been specified by assigning the constant DECWmIconifyPixmapMask as the value of the *value_mask* member of the data structure. Table 14-4 lists the constants you would use to identify the members of the DEC WM Hints data structure to which you have assigned values.
- ⑦ The example sets the value of this attribute using the CHANGE_PROPERTY Xlib routine. The example passes the DECwindows window manager structure as an argument to the routine. For more information about this routine, see the *VMS DECwindows Xlib Routines Reference Manual*.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

If the user has chosen to use small icons, the pixmap specified for the shrink-to-icon button is also used in the icon. In the DECwindows environment, a user can choose between two sizes of icon: large and small. For information about the icon pixmap, see Section 14.7.3.2.

14.7.1.4 Including Shrink-to-Icon, Push-to-Back, and Resize Buttons in the Title Bar

By default, the window manager includes shrink-to-icon, push-to-back, and resize buttons in the title bar. These buttons invoke the window manager functions that allow a user to shrink an application to an icon, change the position of a window in the stacking order, or change the size of the window. You can specify that the title bar not include these buttons by setting the `no_iconify`, `no_lower`, and `no_resize` attributes to true.

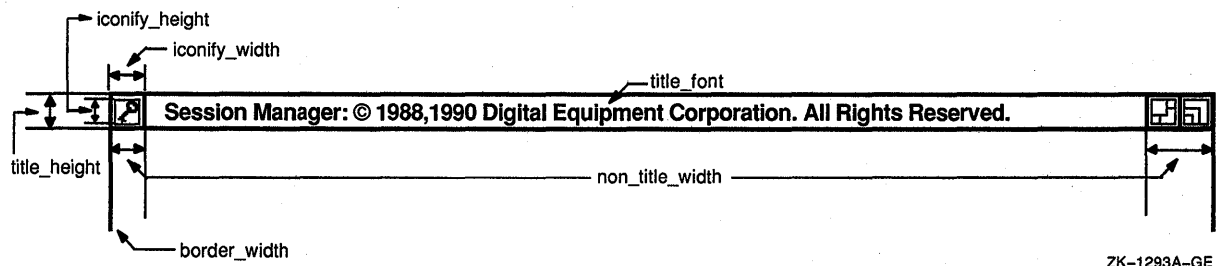
14.7.2 Getting Information About Your Main Application Window

In addition to the attributes and properties that you can set, you can also use certain attributes and properties to inquire about aspects of your main application window. The window manager uses these attributes and properties to communicate information about sizes it supports and other restrictions. You cannot assign values to these attributes; however, you can use them to find out the following information:

- Font used to display text in the title bar
- Height of the title bar
- Width of the title bar, not including the width of the title
- Width and height of the pixmap used for the shrink-to-icon button
- Width of the border the window manager puts around the main window of your application

Figure 14-5 shows the parts of the main application window about which you can inquire. Each part is labeled with the name of the shell widget attribute you would use to specify it.

Figure 14-5 Informational Attributes Provided by the Window Manager



Use the `title_font` attribute to find out the font the window manager uses to display text. To find out the height of the title bar, read the value of the `title_height` attribute. To find out how large to make the pixmap that you want to use as the shrink-to-icon button, read the value of the `iconify_height` and `iconify_width` attributes.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

To find out how much space is available for displaying text in the title bar, subtract the value of the **non_title_width** attribute from the total width of your main application window. The value of the **non_title_width** attribute is the sum of the widths of the shrink-to-icon, push-to-back, and resize buttons plus the sum of the border widths on each side of the window with the border widths around each button. For example, if the window manager supports shrink-to-icon button pixmaps that are 17 pixels wide, the value of the **non_title_width** attribute would be 61 pixels. This value is calculated by multiplying 17 pixels by 3 (for each button in the title bar) and adding 2 pixels for each of the 5 borders.

You can also obtain this information by using the GET WINDOW PROPERTY Xlib routine on the DEC_WM_DECORATION_GEOMETRY property.

14.7.3 Customizing Your Application Icon

The window manager creates an icon for every application running on a workstation. The icon is a rectangular window containing a graphical pixmap and a text string. When an application is active (that is, when its main window is mapped), the window manager displays a dimmed image of the pixmap. When an application is in its iconic state, the image appears at full brightness. The DECwindows window manager displays these icons in an **Icon Box**.

You can customize various aspects of the icon associated with your application, including:

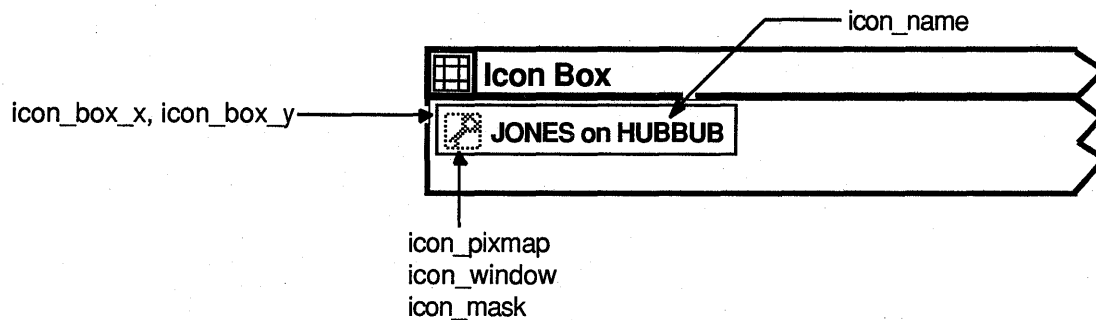
- Text that appears in the icon
- Pixmap used in the icon
- Position of the icon in the Icon Box

Figure 14–6 shows the Icon Box from Figure 14–3. In this illustration, the parts of the icon you can customize are labeled with the name of the shell widget attribute name you would use to specify it.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Figure 14-6 Customizable Aspects of Your Application Icon



ZK-1292A-GE

14.7.3.1 Specifying the Text in the Icon

To specify the text you want to appear in the icon associated with your application, assign the address of a text string as the value of the **icon_name** attribute. Using window properties, you would assign the address of the text string as the value of the predefined Xlib property `WM_ICON_NAME` using the `SET ICON NAME` Xlib routine. You must pass standard text strings for these text values; the window manager does not use compound strings.

If you do not specify an icon name, the DECwindows window manager uses the text you specified for the value of the **title** attribute as the icon name.

Some window managers restrict the length of the text string they display in icons. To find out this restriction, read the value of the **icon_name_width** attribute or retrieve the value from the `DEC_WM_DECORATION_GEOMETRY` property. The DECwindows window manager places no restriction on the length of the icon name (the attribute value is 0).

You can also find out what font the window manager uses to display the text by reading the value of the **icon_font** attribute or retrieving the value from the `DEC_WM_DECORATION_GEOMETRY` property.

14.7.3.2 Specifying the Pixmap Used in Your Application Icon

To specify the pixmap that the window manager will use in your application icon, assign a pixmap identifier as the value of the **icon_pixmap** attribute. Example 14-6 presents a version of the Hello World! application, introduced in Chapter 1, that includes a customized icon. (For an example of how to specify the icon pixmap using the `SET WM HINTS` Xlib routine, see Section 14.7.1.3.) If you do not specify an icon pixmap, the DECwindows window manager uses the default paned window pixmap in the icon. However, if the user has selected the small icon option, the window manager uses the pixmap that you specified for the shrink-to-icon button as the pixmap in the icon.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Example 14-6 Using Shell Widget Attributes to Specify Your Application Icon

```
#include <stdio>
#include <decw$include/DwtAppl.h>
①#include <decw$include/vendor.h>
#define icon_width 32
#define icon_height 32
②static char icon_bits[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0xc0, 0x01, 0x00,
    0x00, 0xc2, 0x41, 0x00, 0x00, 0xc7, 0xe1, 0x00, 0x00, 0xc7, 0xe1, 0x00,
    0x00, 0xc7, 0xe1, 0x00, 0x00, 0xcf, 0xe1, 0x00, 0x00, 0xce, 0x61, 0x18,
    0x00, 0xce, 0x71, 0x1c, 0x00, 0xce, 0x71, 0x1c, 0x18, 0xce, 0x71, 0x0e,
    0x1c, 0xce, 0x71, 0x0f, 0x1c, 0xce, 0x39, 0x07, 0x1c, 0xde, 0xb9, 0x03,
    0x1c, 0xfe, 0xff, 0x03, 0x3c, 0xfe, 0xff, 0x03, 0x3c, 0xfe, 0xff, 0x03,
    0xf8, 0xff, 0xff, 0x03, 0xf8, 0xff, 0xff, 0x03, 0xf8, 0xff, 0xff, 0x03,
    0xf0, 0xff, 0xff, 0x03, 0xf0, 0xff, 0xff, 0x03, 0xc0, 0xff, 0xff, 0x03,
    0x80, 0xff, 0xff, 0x01, 0x80, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x00,
    0x00, 0xfe, 0xff, 0x00, 0x00, 0xfc, 0x7f, 0x00, 0x00, 0xfc, 0x7f, 0x00,
    0x00, 0xfc, 0x7f, 0x00, 0x00, 0x00, 0x00, 0x00};

#define sm_icon_width 17
#define sm_icon_height 17
static char sm_icon_bits[] = {
    0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x80, 0x19, 0x00, 0x90, 0x19, 0x00,
    0xb0, 0x99, 0x00, 0xb0, 0xc9, 0x00, 0xb0, 0x4d, 0x00, 0xb0, 0x6d, 0x00,
    0xb4, 0x6d, 0x00, 0xf6, 0x7f, 0x00, 0xe6, 0x7f, 0x00, 0xee, 0x7f, 0x00,
    0xec, 0x7f, 0x00, 0xfc, 0x3f, 0x00, 0xf0, 0x3f, 0x00, 0xc0, 0x1f, 0x00,
    0xc0, 0x1f, 0x00};

static Display *display;

static void helloworld_button_activate();

static DwtCallback callback_arg[2];

int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Widget toplevel, helloworldmain, button, label;
    Arg arglist[5];
    int count = 0;
    ③Pixmap IconPixmap;
    Pixmap SmallIconPixmap;

    toplevel = XtInitialize("Hi", "helloworldclass", NULL, 0, &argc, argv);
    display = XtDisplay( toplevel);

    ④IconPixmap = XCreateBitmapFromData( display,
        XDefaultRootWindow( display),
        icon_bits,
        icon_width,
        icon_height );

    SmallIconPixmap = XCreateBitmapFromData( display,
        XDefaultRootWindow( display),
        sm_icon_bits,
        sm_icon_width,
        sm_icon_height );
```

(continued on next page)

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Example 14–6 (Cont.) Using Shell Widget Attributes to Specify Your Application Icon

```
⑤ XtSetArg( arglist[count], XtNallowShellResize, TRUE ); count++;
   XtSetArg( arglist[count], XtNiconPixmap, IconPixmap ); count++;
   XtSetArg( arglist[count], XtNiconifyPixmap, SmallIconPixmap ); count++;

   XtSetValues (toplevel, arglist, count );

/**** Remainder of Hello World! sample application ****/
.
.
.
```

- ① The example includes the XUI Toolkit symbol definition file *vendor.h*. This symbol definition file contains the definitions of the shell widget attributes that are needed to specify the shrink-to-icon button pixmap.
- ② The example includes the bits that define the icon pixmap.
- ③ The example declares two variables of type *Pixmap* to hold the pixmap identifiers for the shrink-to-icon button pixmap and the icon pixmap.
- ④ The example creates the pixmaps using the CREATE BITMAP FROM DATA Xlib routine. The routine returns the identifier of the bitmap created. For more information about this routine, see the *VMS DECwindows Xlib Routines Reference Manual*.
- ⑤ The example assigns the identifiers of the pixmaps as values in an argument list that is used to set the values of shell widget attributes.

If the depth of the pixmap you provide does not match the depth of the display, the window manager uses the default pixmap. Use only the default colormap for icons because you cannot assume other colormaps are installed.

To create an icon that appears nonrectangular, use a pixmap mask that specifies which pixels in the icon pixmap should be used as the icon. Either assign the identifier of the pixmap mask as the value of the **icon_mask** shell widget attribute, or use the SET WM HINTS Xlib routine to specify this value in the WM_HINTS property.

14.7.3.3 Using a Window in Your Icon

You can specify that your application icon contain a window instead of a pixmap. Assign the identifier of the window as the value of the **icon_window** attribute or, if you are using window properties, use the SET WM HINTS Xlib routine to set this value in the WM_HINTS property.

By using a window instead of a pixmap, you can actively draw graphics into your icon instead of using a static graphic. For example, a clock application, which continuously displays the time in its icon, could replace the pixmap with a window into which it continuously updates the clock face. Your application should not depend on user input through your icon window.

Communicating with the Window Manager

14.7 Customizing Your Application Using Window Manager Hints

Even if your application uses an icon window, you should still supply an icon pixmap because the window manager creates the dimmed image by stippling the icon pixmap. If your application's icon contains a window, the window manager creates the dimmed image by taking a snapshot of the window and stippling it. Because the application may have been painting into its icon window at the time of the snapshot, the stippling operation can produce unexpected results. If you specify an icon pixmap in addition to the icon window, the window manager uses the pixmap to display the dimmed version of the graphic when your application is in its active mapped state.

14.7.3.4 Positioning Your Icon on the Display

You can specify the position of your application icon by assigning values to the **icon_box_x** and **icon_box_y** shell widget attributes or by using the SET WM HINTS Xlib routine to set these values in the DEC_WM_HINTS property. Specify the values for these attributes in pixels. If you do not specify values for these attributes, the DECwindows window manager positions your application icon in the Icon Box so that it does not obscure any other icon.

14.7.4 Specifying the Initial State of Your Application

To make your application initially appear in its iconic state, set the **iconic** shell widget attribute to true. This sets the value of the **initial_state** attribute to the constant **IconicState**. If you are using window properties, use the SET WM HINTS Xlib routine to specify the value of this member of the WM_HINTS property. Table 14-2 lists the constants Xlib has defined as possible values for this attribute.

To find out whether your application is in its iconic state, read the value of the **icon_state** shell widget attribute. If you are using window properties, use the GET WINDOW PROPERTY Xlib routine to read the value of the DEC_WM_ICON_STATE property.

14.7.5 Creating Transient and Sticky Windows

To create a transient window, set the **transient** attribute to true or use the WM_TRANSIENT_FOR property.

To make a window stay in its position in the stacking order, even if it gets input focus, set the **sticky** attribute to true.

14.7.6 Bypassing the Window Manager

You can specify that your application should bypass the window manager by setting the **override_redirect** attribute to true. When you bypass the window manager, your application appears without a title bar and icon. Users cannot move or resize the application, nor can they shrink it to an icon.

A

Using the DECTERM PORT Routine

Your application can use the DECTERM PORT routine to create a DECterm window on any node, local or remote. You can also create DECterm windows by spawning a CREATE/TERMINAL command; however, using the DECTERM PORT routine provides better performance. (Users can create a DECterm from the session manager's Applications menu or by using the CREATE/TERMINAL command in DCL.)

Example A-1 illustrates how to use the DECTERM PORT routine to create a DECterm on a remote system.

Example A-1 Creating a DECterm on a Remote Node

```
#include descrip      /* descriptor definitions */
#include ssdef        /* system status codes */
#include prcdef       /* stsflg bits for creating process */

main( )
{
    int status, stsflg;
    short device_length;

    ❶ char device_name[50];
      $DESCRIPTOR( command, "SYS$SYSTEM:LOGINOUT.EXE" );
      $DESCRIPTOR( input_file, "" );
      $DESCRIPTOR( output_file, "" );

    /* send the message to the controller */

    ❷ status = DECwTermPort( 0, 0, 0, device_name, &device_length );
      if ( status != SS$NORMAL )
          printf( "DECterm creation failed, status is %x\n", status );
      else
      {
          /* create a process that is already logged in */
          /* input from TWn: */

          ❸ input_file.dsc$w_length = device_length;
            input_file.dsc$a_pointer = device_name;

          /* output to TWn: */
            output_file.dsc$w_length = device_length;
            output_file.dsc$a_pointer = device_name;

          /* make it detached, interactive, logged in */
            stsflg = PRC$M_DETACH | PRC$M_INTER | PRC$M_NOPASSWORD;

          /* create the process */

          ❹ status = sys$creprc( 0, &command, &input_file,
                                &output_file, 0, 0, 0, 0, 4, 0, 0, stsflg );
            if ( status != SS$NORMAL )
                printf( "Could not run LOGINOUT.EXE, status is %x\n", status );
      }
}
```

- ❶ The DECTERM PORT routine returns the name of the virtual terminal device in this character array.

Using the DECTERM PORT Routine

- ② This call to the DECTERM PORT routine creates a DECTerm window on a remote node. In the example, the **display** argument is specified as 0. This indicates that the default display should be used. By specifying the second argument as 0, the example uses the default setup file. By specifying the third argument as 0, the example specifies that the default values in the setup and resource files should not be overridden.

The DECTERM PORT routine returns the name of the virtual terminal device in the fourth argument, *device_name*. The DECTERM PORT routine writes the length of the virtual terminal device name in the last argument, *device_length*.

- ③ After successfully creating a remote DECTerm, the example creates a process that is already logged in.
- ④ This call to SYS\$CREPRC creates the process that runs in the DECTerm window. The SYS\$INPUT of the process is the DECTerm window, and the process is created with a priority of 4. The process is logged in as a detached process.

Example A-2 provides a command procedure to compile, link, and run the example program.

Example A-2 Command Procedure to Compile, Link, and Run a DECTerm on a Remote Node

```
①$ cc create_decterm
②$ link create_decterm, sys$input/opt
sys$share:decw$plibshr/share
sys$share:decw$dwtlibshr/share
sys$share:vaxcrtl/share
sys$share:decw$terminalshr/share
③$ set display/create/node=mynode
④$ run create_decterm
```

- ① The command procedure invokes the compiler to compile the example program.
- ② The command procedure invokes the linker, specifying the name of the object module and an options file as command line arguments. The options file lists the shareable libraries needed to run the example program. The DECTerm shareable image is named *decw\$terminalshr*.
- ③ The default display is set to point to *mynode*. Because the display argument to the DECTERM PORT routine in Example A-1 was specified as 0 (zero), the DECTerm is created on *mynode*. The same effect could have been achieved by specifying the display argument to the DECTERM PORT routine as "mynode::0".
- ④ The command procedure runs the example program.

DECTERM PORT

Creates a DECterm window on a node.

VAX FORMAT

status = **DECW\$TERM_PORT**
(*display*, *setup_file*, *customization*, *result_dev*,
result_len[,*controller*][,*char_buff*][,*char_chng_buff*])

argument information

Argument	Usage	Data Type	Access	Mechanism
<i>status</i>	uns longword	uns longword	write	value
<i>display</i>	char string	char string	read	descriptor
<i>setup_file</i>	char string	char string	read	descriptor
<i>customization</i>	char string	char string	read	descriptor
<i>result_dev</i>	char string	char string	write	descriptor
<i>result_len</i>	word	word	write	reference
<i>controller</i>	char string	char string	read	descriptor
<i>char_buff</i>	record	uns longword	read	descriptor
<i>char_chng_buff</i>	record	uns longword	read	descriptor

MIT C FORMAT

status = **DECwTermPort**
(*display*, *setup_file*, *customization*, *result_dev*,
result_len[,*controller*][,*char_buff*][,*char_chng_buff*])

argument information

```
int DECwTermPort (display, setup_file, customization, result_dev,
                  result_len[, controller][, char_buff]
                  [, char_chng_buff])
    char          *display;
    char          *setup_file;
    char          *customization;
    char          *result_dev;
    short        *result_len;
    char          *controller;
    struct tt_chars *char_buff;
    struct tt_chars *char_chng_buff;
```

Using the DECterm Port Routine

DECTERM PORT

RETURNS

status

SS\$_NORMAL VMS status code indicating successful completion.

ARGUMENTS

display

A character string that identifies the server and screen on which the created DECterm appears. If the string address is 0, the default display is used.

setup_file

A character string that specifies the name of the setup file. The setup file changes DECterm's initial settings. (See the **customization** argument for information about the syntax of a setup file.) If the string address is 0, the default setup file, DECW\$USER_DEFAULTS:DECW\$TERMINAL_DEFAULT.DAT, is used.

customization

A character string that specifies setup options that override the default values established in resource and setup files. If the string address is 0, default values are not to be overridden. The syntax is the same as the syntax for resource and setup files:

```
"param: value \n param: value \n param: value...."
```

In languages other than C, replace "\n" with a line-feed character, ASCII code 10.

You can create a customization file using the Application menu of the session manager. To do this, create a DECterm using the Applications menu, use the Customize option to change settings, and save the new settings in a file (be sure to use a nondefault file name). You can use the name of this file as the value of the customization argument.

result_dev

A character string that specifies the virtual terminal device name for the created DECterm. This argument is intended for applications that want to assign the created DECterm or pass the name to a new process.

result_len

The address of a 16-bit word into which the length of the returned device name is written. When using the VAX calling format, you can point this argument directly at the *result_dev* descriptor to trim the descriptor for subsequent use.

controller

An optional argument that is a character string that specifies which controller should be used with the DECterm window. For example, you can specify a foreign language variant of DECterm. The default is SYS\$SYSTEM:DECW\$TERMINAL.EXE.

char_buff

An optional argument that is the address of a 12-byte terminal characteristic buffer in which the terminal characteristics of the DECterm are specified. See the *VMS I/O User's Reference Manual: Part I* for further information.

char_chng_buff

An optional argument that is the address of a 12-byte terminal characteristic buffer that specifies which characteristics are set in the **char_buff** argument. This argument must be specified with the **char_buff** argument. Only those terminal characteristics that have nonzero values in the **char_chng_buff** buffer are set to the values specified in the **char_buff** argument. Otherwise, the terminal characteristic is not changed. See the *VMS I/O User's Reference Manual: Part I* for further information.

DESCRIPTION

The DECTERM PORT routine creates a DECTerm window on a local or remote node.

B Using the VAX Bindings

This appendix presents information on using the XUI Toolkit with the VAX bindings. The appendix includes three example programs that re-create the Hello World! application using the VAX binding version of the XUI Toolkit routines. In addition, the appendix includes specific information about using the VAX bindings with the Ada programming language.

Example B-1 is a version of the Hello World! sample application written in VAX Ada. Example B-2 is a version of the Hello World! sample application written in VAX FORTRAN. Both of these sample programs use the same UIL module as the C language version of the Hello World! sample application described in Chapter 2. Example 2-16 shows this UIL module.

Example B-3 presents a VAX Pascal version of the Hello World! sample application written using the high-level widget creation routines.

B.1 Using the DECwindows Ada Programming Interfaces

DECwindows provides programming interface definitions for the Ada language. When you select Ada support at the time of the DECwindows kit installation, the following Ada package source files are placed in the SYS\$LIBRARY: directory of your system:

- CDA\$CDA_.ADA—Package CDA (Compound Document Architecture)
- DDIF\$DDIF_.ADA—Package DDIF (Digital Document Interchange Format)
- DTIF\$DTIF_.ADA—Package DTIF (Digital Table Interchange Format)
- DECW\$DWT_.ADA—Package DWT (DECwindows Toolkit)
- DECW\$X_.ADA—Package X (Xlib)

These package source files can be individually compiled into your Ada program libraries or compiled into the systemwide Ada predefined library. To make the packages available systemwide, the command file SYS\$UPDATE:DECW\$COMPILE_ADA_UNITS.COM is provided.

This command procedure compiles all four packages into the predefined Ada library, and, if the VAX Source Code Analyzer (SCA) product is present, loads SCA analysis data for the packages into the SCA library for the predefined library. The command procedure should be run as a batch job and should have available a minimum of 2000 pages in the working set; however, 3000 pages is preferable. A page file quota of at least 30,000 pages is suggested.

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

Once the units are compiled into the predefined Ada library, you must execute the following Ada program library manager command to make the units visible:

```
ACS ENTER UNIT ADA$PREDEFINED CDA,DDIF,DTIF,DWT,X
```

You need do this only once. This step is also performed automatically for all Ada program libraries created after the DECwindows units are compiled into the predefined library.

Future installations of DECwindows might replace the Ada packages. If so, the new packages must be compiled as shown. If you have already entered the units into your own library, you must then execute the following command to make your library current:

```
ACS REENTER *
```

Future installations of VAX Ada might replace the Ada predefined library and remove the DECwindows units. If this occurs, reexecute the DECW\$COMPILE_ADA_UNITS.COM command procedure.

If you want to compile the units into your program libraries directly, you must execute the following commands after compilation of packages DWT and X:

```
ACS ENTER FOREIGN SYS$SHARE:DECW$DWTLIBSHR/SHAREABLE DWT  
ACS ENTER FOREIGN SYS$SHARE:DECW$XLIBSHR/SHAREABLE X
```

This step is not necessary if the units are entered from the predefined library.

Once the units are entered into your program library, applications that use the DECwindows packages are linked in the normal manner using ACS LINK. It is not necessary to explicitly specify the shareable images when linking.

B.1.1 Using the Ada Packages

Each package (CDA, DDIF, DTIF, DWT, and X) contains definitions of constants, structures, status codes, and routines for each facility. All the packages observe certain common conventions for naming and use; these conventions are outlined as follows:

- All dollar signs (\$) in symbols have been replaced with underscores (_). The dollar sign is not allowed in Ada identifiers.
- In each package, the facility prefix (CDA\$, DDIF\$, DTIF\$, DWT\$, X\$) has been removed from all the symbols defined in that package. It is intended that the Ada USE clause not be used with these packages. This encourages clarity in the application source and also improves compiler efficiency. For example:

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

```
with DWT;
procedure CALLBACK (
WIDGET: in DWT.WIDGET_TYPE) is
...
ARGLIST: DWT.ARG_ARRAY_TYPE (0..0);
CSTRING: DWT.COMP_STRING_TYPE;
...
begin
DWT.LATIN1_STRING (.....);
```

In some packages, symbols defined with other facility prefixes are present; these have not been removed from the symbol names. For example, routine XT\$INITIALIZE is DWT.XT_INITIALIZE.

- When a symbol would conflict with an Ada reserved word or predefined identifier, the last letter of the symbol name is removed. For example, the routine DWT\$STRING is DWT.STRIN. See the individual package descriptions for a list of affected identifiers.
- Unconstrained array types are defined as *name_ARRAY_TYPE* for an array of *name_TYPE* elements. The DECwindows documentation sometimes uses *name_LIST* for such arrays; in the Ada packages, these names are used when the address of an array is desired, most commonly as an element of a structure.
- All functions are defined as “valued procedures.” The function return value is usually named STATUS or RESULT, depending on the type of value returned.
- The null-terminated strings required by some procedures can be created by concatenating the string with ASCII.NUL. Further information about the interfaces can be found by examining the package sources provided in SYS\$LIBRARY:, as described previously.

Usage information for the specific packages is given in the following sections.

B.1.1.1 Package CDA

This package defines constants and types for the Compound Document Architecture (CDA) facility. There are no package-specific usage comments for package CDA.

B.1.1.2 Package DDIF

This package defines constants and types for the Digital Document Interchange Format (DDIF) facility. There are no package-specific usage comments for package DDIF.

B.1.1.3 Package DTIF

This package defines constants and types for the Digital Table Interchange Format (DTIF) facility. There are no package-specific usage comments for package DDIF.

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

B.1.1.4 Package DWT

This package defines constants, types, and procedures for the XUI Toolkit facility. The following usage comments are specific to package DWT:

- The procedure `STRING` is renamed `STRIN` to avoid conflict with the predefined type.
- The parameter `ADDRESS` of procedure `XT_FREE` is renamed `ADDRES` to avoid conflict with the predefined type.
- The subtype definitions listed in Table B-1 rename types from package DWT.

Table B-1 Subtype Definitions—Package DWT

Subtype	Definition
<code>DISPLAY_TYPE</code>	<code>X.DISPLAY_TYPE</code>
<code>EVENT_TYPE</code>	<code>X.EVENT_TYPE</code>
<code>GC_TYPE</code>	<code>X.GC_ID_TYPE</code>
<code>PIXMAP_TYPE</code>	<code>X.PIXMAP_ID_TYPE</code>
<code>TIME_TYPE</code>	<code>X.TIME_TYPE</code>
<code>SCREEN_TYPE</code>	<code>X.SCREEN_ID_TYPE</code>
<code>WINDOW_TYPE</code>	<code>X.WINDOW_ID_TYPE</code>
<code>XRMDATABASE_TYPE</code>	<code>X.DATABASE_ID_TYPE</code>

- The types `INTEGER_ARRAY` and `ADDRESS_ARRAY` are defined for use with procedures in package DWT, being unconstrained arrays of `INTEGER` and `ADDRESS`, respectively.
- The type `DESCRIPTOR_TYPE` is defined for constructing string descriptors required by certain procedures.

B.1.1.5 Package X

This package defines types, structures, and procedures for the Xlib facility. The following usage comments are specific to package X:

- The subtype definitions listed in Table B-2 are provided.

Table B-2 Subtype Definitions—Package X

Subtype	Definition
<code>ATOM_ID_TYPE</code>	<code>SYSTEM.UNSIGNED_LONGWORD</code>
<code>BITMAP_ID_TYPE</code>	<code>SYSTEM.UNSIGNED_LONGWORD</code>
<code>CLASS_LIST_ID_TYPE</code>	<code>SYSTEM.UNSIGNED_LONGWORD</code>
<code>COLORMAP_ID_TYPE</code>	<code>SYSTEM.UNSIGNED_LONGWORD</code>
<code>CURSOR_ID_TYPE</code>	<code>SYSTEM.UNSIGNED_LONGWORD</code>
<code>DATABASE_ID_TYPE</code>	<code>SYSTEM.UNSIGNED_LONGWORD</code>

(continued on next page)

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

Table B-2 (Cont.) Subtype Definitions—Package X

Subtype	Definition
DISPLAY_ID_TYPE	SYSTEM.ADDRESS
DISPLAY_TYPE	SYSTEM.ADDRESS
DRAWABLE_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
FONT_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
GC_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
KEYSYM_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
NAME_LIST_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
PIXMAP_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
PROPERTY_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
REGION_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
SCREEN_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
SEARCH_LIST_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
SELECTION_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
TARGET_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
TIME_TYPE	SYSTEM.UNSIGNED_LONGWORD
TYPE_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD
WINDOW_ID_TYPE	SYSTEM.UNSIGNED_LONGWORD

- The argument `EVENT_TYPE` of procedures `CHECK_TYPED_EVENT` and `CHECK_TYPED_WINDOW_EVENT` has been renamed to `EVENT_TYP` to avoid conflict with the `EVENT_TYPE` type definition.
- `EVENT_TYPE` has been defined as a variant record; subtypes for specific event types are also defined as specific instances of the variant record. The discriminant for `EVENT_TYPE` is the field `EVNT_TYPE`; each variant uses its own prefixes for the field names, for example, `KYEV_DISPLAY` for a key event.

When you declare a variable as being type `EVENT_TYPE`, Ada automatically allocates the maximum possible event size for the variable. When examining event variables, be sure to use only the correct fields for the variant defined by `EVNT_TYPE`; otherwise an Ada constraint error can be generated. For example, the following code is correct:

```
if EVENT.EVNT_TYPE = X.C_EXPOSE then
if EVENT.EXEV_WINDOW = WINDOW_2 then
.
.
.
```

while the following is incorrect:

```
if EVENT.EVNT_TYPE = X.C_EXPOSE and
EVENT.EXEV_WINDOW = WINDOW_2 then
.
.
.
```

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

The second code fragment would raise a constraint error on the reference to `EVENT.EXEV_WINDOW` if the value of the discriminant (`EVNT_TYPE`) was not `C_EXPOSE`.

B.1.2 Callbacks

When using the DECwindows Toolkit with callback routines written in Ada, the tag provided to the Toolkit must be the address of the value to be returned as the tag. The actual value cannot be used due to a conflict in the calling mechanisms. The DECwindows Toolkit passes the tag to the callback routine by immediate value. Callback routines written in Ada expect the tag to be passed by reference. Passing the address of the value to be returned allows the Ada code to access the tag by reference.

Ada procedures that are to be used as callback routines must be made visible by means of the `EXPORT_PROCEDURE` pragma. This requires that the procedure be a library unit or be declared in the outermost declarative part of a library package. See the section on Exporting Subprograms in the *VAX Ada Language Reference Manual* for more details.

Be aware that `EXPORT_PROCEDURE` implicitly declares the procedure name as a global symbol. If the same procedure name is used in multiple packages, you should specify an "external designator" as the second argument of pragma `EXPORT_PROCEDURE` to give the procedure a unique external name.

Callback routines used in tasking applications must also specify pragma `SUPPRESS_ALL`. This suppresses the task stack check that might fail for routines called from outside the context of an Ada task.

B.1.3 Tasking Considerations

Ada programs that use tasking can call DECwindows routines, but applications designers should be aware that the DECwindows design philosophy is oriented towards event polling and not asynchronous notification of events.

An important consideration is that calling DECwindows routines that wait for an event, such as `X.NEXT_EVENT`, potentially can block all tasks until the event occurs. To VAX Ada, a task that is blocked in a DECwindows call appears to be continuing to execute. VAX Ada does not know that the task is in any way blocked. As a result, the only tasks that can execute while a task is blocked in a DECwindows call are tasks that have higher priorities than the calling task, unless time slicing is in effect. If time slicing is enabled with pragma `TIME_SLICE`, then in addition to the higher priority tasks, other tasks of equal priority will get a chance to run at the end of the time slice; but when the blocked task is again scheduled, it will block the tasks of equal priority until its time slice has expired. Tasks of lower priority will not run.

B.1 Using the DECwindows Ada Programming Interfaces

Another consideration is that even though other tasks may be scheduled, they must not call DECwindows while another call is outstanding. DECwindows is not reentrant. The user must ensure that at most one task is calling DECwindows at any one time. The simplest way to ensure this is to dedicate a single task to calling DECwindows. Whenever it is desirable to perform work in a different task, the "DECwindows task" must be made to suspend if there is any possibility that the other task may call any DECwindows function. This can be done by using a rendezvous to block the dedicated task and to transfer control to other tasks in the application.

B.1.4 Ada Examples

The VMS DECwindows examples directory (DECW\$EXAMPLES:) contains the following Ada language examples:

- HELLOWORLD.ADA is a simple example of using the XUI Toolkit and XUI Resource Manager (DRM). Example B-1 presents this program.
- DECBURGER.ADA is a more complex example that uses many of the widgets in the XUI Toolkit, demonstrates the use of callbacks, and illustrates how to use UIL and DRM to create a user interface.
- XLIBINTRO.ADA demonstrates the use of the Xlib interface and responding to events.

The first two example programs require that the appropriate UIL file from the directory DECW\$EXAMPLES be compiled using the UIL compiler before running the programs. See the command procedure DECW\$EXAMPLES:DEMO_BUILD.COM for details on compiling and linking the example applications. Example B-1 is the VAX Ada version of the Hello World! application.

Example B-1 Hello World! Application in VAX Ada

```
with DWT;
package HELLOWORLD_CALLBACKS is

procedure HELLOWORLD_BUTTON_ACTIVATE (
    WIDGET: in DWT.WIDGET_TYPE;
    TAG    : in INTEGER;
    REASON: in INTEGER);

pragma EXPORT_PROCEDURE (HELLOWORLD_BUTTON_ACTIVATE);

DONE: exception;

end HELLOWORLD_CALLBACKS;

with DWT,SYSTEM;
package body HELLOWORLD_CALLBACKS is
    PUSHED: BOOLEAN := FALSE;

    ① procedure HELLOWORLD_BUTTON_ACTIVATE (
        WIDGET: in DWT.WIDGET_TYPE;
        TAG    : in INTEGER;
        REASON: in INTEGER) is
```

(continued on next page)

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

Example B-1 (Cont.) Hello World! Application in VAX Ada

```
② ARG_LIST: DWT.ARG_ARRAY_TYPE (0..1);
   CSTRING: DWT.COMP_STRING_TYPE;
   CS_STATUS: INTEGER;

begin
  if PUSHED then
    raise DONE;
  else
    ③ DWT.LATIN1_STRING (
        STATUS => CS_STATUS,
        TEXT => "Goodbye"&ASCII.CR&"World!",
        COMPOUND_STRING => CSTRING);

    ④ DWT.VMS_SET_ARG (
        ARG => CSTRING,
        ARGLIST => ARG_LIST,
        ARGNUMBER => 0,
        ARGNAME => DWT.C_NLABEL);

        DWT.VMS_SET_ARG (
        ARG => 11,
        ARGLIST => ARG_LIST,
        ARGNUMBER => 1,
        ARGNAME => DWT.C_NX);

    ⑤ DWT.XT_SET_VALUES (
        WIDGET => WIDGET,
        ARGLIST => ARG_LIST,
        ARGCOUNT => ARG_LIST'LENGTH);

        DWT.XT_FREE (CSTRING);

        PUSHED := TRUE;

  end if;
  end HELLOWORLD_BUTTON_ACTIVATE;
end HELLOWORLD_CALLBACKS;
--
-- Main program
--
with DWT, HELLOWORLD_CALLBACKS, STARLET, SYSTEM, TEXT_IO;
use HELLOWORLD_CALLBACKS, SYSTEM;
procedure HELLOWORLD is

    ⑥ TOPLEVEL, HELLOWORLD_MAIN: DWT.WIDGET_TYPE;

    ⑦ HIERARCHY_FILE_NAME : constant STRING := "HELLOWORLD.UID";
      HIERARCHY_FILE_DESCR : DWT.DESRIPTOR_TYPE := (
        CLASS => STARLET.DSC_K_CLASS_S,
        DTYPE => STARLET.DSC_K_DTYPE_T,
        LENGTH => HIERARCHY_FILE_NAME'LENGTH,
        POINTER => HIERARCHY_FILE_NAME'ADDRESS);
      HIERARCHY_NAME_LIST : DWT.ADDRESS_ARRAY (0..0) :=
        (0=> HIERARCHY_FILE_DESCR'ADDRESS);

      CALLBACK_NAME : constant STRING := "helloworld_button_activate"&ASCII.NUL;

      CALLBACK_ARGLIST : DWT.DRMREG_ARG_ARRAY_TYPE (0..0) :=
        (0 => (DRMR_NAME => CALLBACK_NAME'ADDRESS,
              DRMR_VALUE => HELLOWORLD_BUTTON_ACTIVATE'ADDRESS));
```

(continued on next page)

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

Example B-1 (Cont.) Hello World! Application in VAX Ada

```
ARG_LIST : DWT.ARG_ARRAY_TYPE (0..0);
DRM_HIERARCHY: DWT.DRM_HIERARCHY_TYPE;
URLIST: UNSIGNED_LONGWORD_ARRAY(0..0) := (0=> 0);
ARGV : DWT.ADDRESS_ARRAY(0..0) := (0=> ADDRESS_ZERO);
ARGC : UNSIGNED_LONGWORD := 0;
HIERARCHY_STATUS, FETCH_STATUS, REGISTER_STATUS: DWT.CARDINAL_TYPE;
CLASS: DWT.DRM_TYPE_TYPE;
```

begin

```
⑧ DWT.INITIALIZE_DRM;
⑨ DWT.XT_INITIALIZE (
    WIDGET      => TOPLEVEL,
    NAME        => "Hi",
    CLASS_NAME  => "helloworldclass",
    URLIST      => URLIST,
    NUM_URLIST  => 0,
    ARGCOUNT   => ARGC,
    ARGVALUE    => ARGV);
DWT.VMS_SET_ARG (
    ARG      => DWT.C_TRUE,
    ARGLIST  => ARG_LIST,
    ARGNUMBER => ARG_LIST'FIRST,
    ARGNAME  => DWT.C_NALLOW_SHELL_RESIZE);
DWT.XT_SET_VALUES (
    WIDGET      => TOPLEVEL,
    ARGLIST     => ARG_LIST,
    ARGCOUNT   => ARG_LIST'LENGTH);
⑩ DWT.OPEN_HIERARCHY (
    STATUS      => HIERARCHY_STATUS,
    NUM_FILES   => HIERARCHY_NAME_LIST'LENGTH,
    FILE_NAMES_LIST => HIERARCHY_NAME_LIST,
    HIERARCHY_ID_RETURN => DRM_HIERARCHY);
if HIERARCHY_STATUS /= DWT.C_DRM_SUCCESS
then
    TEXT_IO.PUT_LINE ("Can't open hierarchy");
    raise DONE;
end if;
⑪ DWT.REGISTER_DRM_NAMES (
    STATUS      => REGISTER_STATUS,
    REGISTER_LIST => CALLBACK_ARGLIST,
    REGISTER_COUNT => CALLBACK_ARGLIST'LENGTH);
if REGISTER_STATUS /= DWT.C_DRM_SUCCESS
then
    TEXT_IO.PUT_LINE ("Can't register callback");
    raise DONE;
end if;
```

(continued on next page)

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

Example B-1 (Cont.) Hello World! Application in VAX Ada

```
12 DWT.FETCH_WIDGET (
    STATUS          => FETCH_STATUS,
    HIERARCHY_ID   => DRM_HIERARCHY,
    INDEX          => "helloworld_main",
    PARENT         => TOPLEVEL,
    W_RETURN       => HELLOWORLD_MAIN,
    CLASS_RETURN   => CLASS);
if FETCH_STATUS /= DWT.C_DRM_SUCCESS
then
    TEXT_IO.PUT_LINE ("Can't fetch interface");
    raise DONE;
end if;

13 DWT.XT_MANAGE_CHILD (
    WIDGET          => HELLOWORLD_MAIN);

14 DWT.XT_REALIZE_WIDGET (
    WIDGET          => TOPLEVEL);

15 DWT.XT_MAIN_LOOP;

exception
    when DONE =>
        NULL;

end HELLOWORLD;
```

- ① This procedure implements the callback routine used by the Hello World! application.
- ② The program creates an argument list by declaring an array of argument data structures. Note that the array begins with element zero.
- ③ The program creates the text string "Goodbye World!" as a compound string. This string will be placed in the push button widget when the callback routine executes. Because it appears on the display, the text string must be converted into a compound string.
- ④ The program uses the VMS SET ARG routine to assign values to attributes in the argument list.
- ⑤ The program calls the intrinsic routine SET VALUES to change the value of the text label in the push button widget.
- ⑥ This statement declares the variables that will hold the widget identifiers.
- ⑦ The sample program creates the DRM hierarchy in which all the UID files used by the application are listed. The Hello World! application uses only one UID file (included in the DECW\$EXAMPLES: directory).
- ⑧ The sample program initializes DRM in this call to the INITIALIZE DRM routine.
- ⑨ The sample program initializes the XUI Toolkit by calling the INITIALIZE intrinsic routine.

Using the VAX Bindings

B.2 Using the FORTRAN Bindings

Example B-2 (Cont.) Hello World! Application in VAX FORTRAN

```
11 CALL XT$MANAGE_CHILD (HELLOWORLD_MAIN)
12 CALL XT$REALIZE_WIDGET (TOPLEVEL)
   ! Main Input Loop
   !
13 CALL XT$MAIN_LOOP
   END
   ! Callback routine
   !
14 SUBROUTINE HELLOWORLD_BUTTON_ACTIVATE (
1   WIDGET, TAG, REASON)
   INCLUDE 'SYS$LIBRARY:DECW$DWTSTRUCT'
   INTEGER*4 WIDGET, TAG, REASON ! Only WIDGET is used
   RECORD /DWT$ARG/ ARG_LIST(0:1)
   INTEGER*4 CSTRING, CS_STATUS
   LOGICAL PUSHED /.FALSE./
   SAVE PUSHED ! Ensure that it is static
   IF (PUSHED) THEN
       STOP
   ELSE
15 CALL DWT$LATIN1_STRING ('Goodbye'//CHAR(13)//'World!',
1   CSTRING)
16 CALL DWT$VMS_SET_ARG (
1   CSTRING,
2   ARG_LIST,
3   0,
4   DWT$C_NLABEL)
   CALL DWT$VMS_SET_ARG (
1   11,
2   ARG_LIST,
3   1,
4   DWT$C_NX)
17 CALL XT$SET_VALUES (WIDGET,ARG_LIST,1)
       CALL XT$FREE (%VAL(CSTRING))
       PUSHED = .TRUE.
   END IF
   RETURN
   END
```

- ❶ The sample program includes the XUI Toolkit symbol definition file.
- ❷ The sample program declares variables to hold the widget identifiers.
- ❸ The sample program declares the descriptor for the DRM hierarchy file name.
- ❹ These statements declare the callback routine. Note that the name is declared as a case-sensitive, null-terminated string.

Using the VAX Bindings

B.2 Using the FORTRAN Bindings

- ⑤ The sample program declares the argument list used to assign values to widget attributes.
- ⑥ The sample program initializes DRM by calling the INITIALIZE DRM routine.
- ⑦ The sample program initializes the XUI Toolkit in this call to the intrinsic routine INITIALIZE.
- ⑧ The sample program opens the DRM hierarchy by calling the OPEN HIERARCHY routine.
- ⑨ The sample program registers the callback routines so that DRM can resolve references to the routines at widget creation time.
- ⑩ The sample program fetches the interface definition in the UID file with this call to the DRM routine FETCH WIDGET. Note that DRM creates the widgets in this call.
- ⑪ The sample program manages the topmost widget in the application widget hierarchy using the MANAGE CHILD intrinsic routine.
- ⑫ The sample program makes the interface appear on the display by realizing the application shell widget using the intrinsic routine REALIZE WIDGET.
- ⑬ The sample program enters an infinite loop in which it waits for events to process.
- ⑭ This is the callback routine used by the Hello World! application.
- ⑮ The sample program converts the text string the push button widget will contain into a compound string using the LATIN1 STRING routine.
- ⑯ The sample program uses the VMS SET ARG routine to assign a value to a widget attribute in an argument list.
- ⑰ The sample program changes the text string in the push button widget by calling the intrinsic routine SET VALUES.

B.3 Using the VAX Pascal Bindings

Example B-3 illustrates how to use the XUI Toolkit with the VAX Pascal programming language.

Using the VAX Bindings

B.1 Using the DECwindows Ada Programming Interfaces

- ⑩ The sample program opens the UID files identified in the DRM hierarchy using the OPEN HIERARCHY routine.
- ⑪ The sample program resolves the values of symbols used in the UIL module in this call to the DRM routine REGISTER NAMES.
- ⑫ The sample program fetches the interface definition in this call to the DRM routine FETCH WIDGET. DRM creates the widgets in the interface in this routine.
- ⑬ The sample program manages the topmost widget in the application interface in this call to the MANAGE CHILD intrinsic routine.
- ⑭ The sample program makes the interface appear on the display by calling the REALIZE WIDGET intrinsic routine.
- ⑮ The sample program enters an infinite loop in which it waits for events to process.

B.2 Using the FORTRAN Bindings

Example B-2 illustrates how to use the XUI Toolkit, UIL, and DRM in FORTRAN with the VAX binding. This example program uses the same UIL module in the DECW\$EXAMPLES: directory as the Ada and C versions of the Hello World! application.

Example B-2 Hello World! Application in VAX FORTRAN

```
PROGRAM HELLOWORLD
①  INCLUDE 'SYS$LIBRARY:DECW$DWTSTRUCT'
    INCLUDE 'SYS$LIBRARY:DECW$DWTENTRY'
    INCLUDE '($DSCDEF)'
②  INTEGER*4 TOPLEVEL, HELLOWORLD_MAIN
③  STRUCTURE /DESCRIPTOR_STRUCT/
    INTEGER*2 LENGTH
    BYTE DTYPE
    BYTE CLASS
    INTEGER*4 POINTER
END STRUCTURE !DESCRIPTOR_STRUCT
RECORD /DESCRIPTOR_STRUCT/ HIERARCHY_FILE_DESCR
INTEGER*4 HIERARCHY_NAME_LIST(0:0) ! Array of pointers
CHARACTER*(*) HIERARCHY_FILE_NAME
PARAMETER (HIERARCHY_FILE_NAME = 'HELLOWORLD.UID')
④  EXTERNAL HELLOWORLD_BUTTON_ACTIVATE
CHARACTER*(*) CALLBACK_NAME
PARAMETER (CALLBACK_NAME =
1  'helloworld_button_activate'//CHAR(0))
RECORD /DWT$DRMREG_ARG/ CALLBACK_ARGLIST(0:0)
⑤  RECORD /DWT$ARG/ ARG_LIST(0:0)
    INTEGER*4 DRM_HIERARCHY
    INTEGER*4 ARGC/0/, CLASS
```

(continued on next page)

Using the VAX Bindings

B.2 Using the FORTRAN Bindings

Example B-2 (Cont.) Hello World! Application in VAX FORTRAN

```
INTEGER*4 HIERARCHY_STATUS, FETCH_STATUS, REGISTER_STATUS

HIERARCHY_FILE_DESCR.LENGTH = LEN(HIERARCHY_FILE_NAME)
HIERARCHY_FILE_DESCR.DTYPE = DSC$K_DTYPE_T
HIERARCHY_FILE_DESCR.CLASS = DSC$K_CLASS_S
HIERARCHY_FILE_DESCR.POINTER = %LOC(HIERARCHY_FILE_NAME)
HIERARCHY_NAME_LIST(0) = %LOC(HIERARCHY_FILE_DESCR)
CALLBACK_ARGLIST(0).DWT$A_DRMR_NAME = %LOC(CALLBACK_NAME)
CALLBACK_ARGLIST(0).DWT$L_DRMR_VALUE =
1. %LOC(HELLOWORLD_BUTTON_ACTIVATE)

! Set up the User Interface
!
⑥ CALL DWT$INITIALIZE_DRM

⑦ TOPLEVEL = XT$INITIALIZE (
1 'Hi', ! NAME
2 'helloworldclass', ! CLASS
3 %VAL(0), ! URLIST (omitted)
4 0, ! URCOUNT
5 ARGV, ! ARGCOUNT
6 %VAL(0) ! ARGVALUE

CALL DWT$VMS_SET_ARG (
1 DWT$C_TRUE,
2 ARG_LIST,
3 0,
4 DWT$C_NALLOW_SHELL_RESIZE)

CALL XT$SET_VALUES (
1 TOPLEVEL,
2 ARG_LIST,
3 1)

⑧ HIERARCHY_STATUS = DWT$OPEN_HIERARCHY (
1 1,
2 HIERARCHY_NAME_LIST,
3 %VAL(0), ! ANCILLARY_STRUCTURES_LIST
3 DRM_HIERARCHY)
IF (HIERARCHY_STATUS .NE. DWT$C_DRM_SUCCESS) THEN
TYPE *, 'Can't open hierarchy, status = ', HIERARCHY_STATUS
STOP
END IF

⑨ REGISTER_STATUS = DWT$REGISTER_DRM_NAMES (
1 CALLBACK_ARGLIST, 1)
IF (REGISTER_STATUS .NE. DWT$C_DRM_SUCCESS) THEN
TYPE *, 'Can't register callback, status = ', REGISTER_STATUS
STOP
END IF

⑩ FETCH_STATUS = DWT$FETCH_WIDGET (
1 %VAL(DRM_HIERARCHY), ! HIERARCHY_ID
2 'helloworld_main', ! INDEX
3 TOPLEVEL, ! PARENT
4 HELLOWORLD_MAIN, ! W_RETURN
5 CLASS) ! CLASS_RETURN
IF (FETCH_STATUS .NE. DWT$C_DRM_SUCCESS) THEN
TYPE *, 'Can't fetch interface, status = ', FETCH_STATUS
STOP
END IF
```

(continued on next page)

Using the VAX Bindings

B.3 Using the VAX Pascal Bindings

Example B-3 Hello World! Application in VAX Pascal

```
①[inherit('decw$dwtdef.pen')]
program helloworld(input,output);
var
    toplevel, helloworldbutton,
    dialogbox          : dwt$widget;
    callback_list      : array[1..2] of dwt$callback;
    callback_list_null : array[1..1] of dwt$callback;
    latin              : dwt$comp_string;
    ② tag              : [static] integer;
    argc              : dwt$cardinal;
    arglist           : array[1..1] of dwt$arg;
function sys$exit(%immed status : integer) : integer; extern;
    ③procedure buttonpressed(var widget : dwt$widget;
                             var tag   : integer;
                             var struct : dwt$any_cb_st);
var arglist : array[1..1] of dwt$arg;
    latin   : dwt$comp_string;
begin
    writeln('Tag is ',tag);
    if tag = 123 then
    begin
        dwt$latin1_string('Goodbye World'(13),latin);
        ④ dwt$vms_set_arg(latin,arglist,0,dwt$c_nlabel);
          xt$set_values(widget,arglist,1);
        ⑤ xt$free(latin);
          tag := 0;
    end
    else
        sys$exit(1);
end;
begin
    argc := 0;
    callback_list_null[1].dwt$a_callback_proc := nil;
    callback_list_null[1].dwt$l_callback_tag  := 0;
    tag := 123;
    ⑥ callback_list[1].dwt$a_callback_proc::integer := iaddress(buttonpressed);
      callback_list[1].dwt$l_callback_tag          := iaddress(tag);
      callback_list[2] := callback_list_null[1];
    ⑦ toplevel := xt$initialize('Hi',
                               'helloworldclass',
                               0,0,argc,);
```

(continued on next page)

Using the VAX Bindings

B.3 Using the VAX Pascal Bindings

Example B-3 (Cont.) Hello World! Application in VAX Pascal

```
dwt$vms_set_desc_arg('TRUE', arglist, 0, dwt$c_nallow_shell_resize);
xt$set_values(toplevel, arglist, 1);

dwt$latin1_string('Box Title', latin);

⑧ dialogbox := dwt$dialog_box(toplevel,
                             'DialogBoxName',
                             dwt$c_false,
                             0, 0,
                             latin,
                             dwt$c_workarea,
                             callback_list_null,
                             callback_list_null);

xt$free(latin);
xt$manage_child(dialogbox);
dwt$latin1_string('HelloWorld' (13)'Once to change' (13)'twice to exit', latin);

⑨ helloworldbutton := dwt$push_button(dialogbox,
                                      'HelloWorldPushButton',
                                      0, 0,
                                      latin,
                                      callback_list,
                                      callback_list_null);

xt$free(latin);
xt$manage_child(helloworldbutton);
xt$realize_widget(toplevel);
xt$main_loop;

end.
```

- ① The program includes the XUI Toolkit symbol definition file for VAX Pascal.
- ② In this statement, the sample program declares a variable to store the tag. In VAX Pascal, this variable must be a static variable.
- ③ This is the callback routine used by the Hello World! application.
- ④ The sample program sets up an argument list using the VMS-specific XUI Toolkit routine VMS SET ARG. The XUI Toolkit identifies widget attributes as null-terminated strings. Because the VAX bindings pass text strings by descriptor, you must use the VMS SET ARG routine to create argument lists. This routine converts the attribute name into a null-terminated text string.
- ⑤ The sample program frees the storage obtained by the compound string routine in this statement.
- ⑥ The sample program sets up a callback list. This list associates the address of the callback routine with a tag.
- ⑦ The sample program initializes the XUI Toolkit in this call to the intrinsic routine INITIALIZE. This routine also returns the widget identifier of the application shell widget.

Using the VAX Bindings

B.3 Using the VAX Pascal Bindings

- ③ The sample program creates the dialog box widget using the high-level routine `DIALOG BOX`.
- ③ The sample program uses the high-level routine `PUSH BUTTON` to create the push button widget used in the application.

C

International Version of the DECburger Application

Example C-1 shows the complete C language program for the international version of the DECburger application. Differences between this version and the original version of DECburger are described in Section 3.4.1.

International Version of the DECburger Application

Example C-1 C Program for the International Version of DECburger

```
#include <stdio.h>
#ifdef VMS
#include <decw$include/DwtAppl.h>
#else
#include <X11/DwtAppl.h>
#endif

#define k_create_order      1
#define k_order_pdme       2
#define k_file_pdme        3
#define k_edit_pdme        4
#define k_nyi               5
#define k_apply             6
#define k_dismiss           7
#define k_noapply           8
#define k_cancel_order     9
#define k_submit_order     10
#define k_order_box        11
#define k_burger_min       12
#define k_burger_rare      12
#define k_burger_medium    13
#define k_burger_well      14
#define k_burger_ketchup   15
#define k_burger_mustard   16
#define k_burger_onion     17
#define k_burger_mayo      18
#define k_burger_pickle    19
#define k_burger_quantity  20
#define k_burger_max       20
#define k_fries_tiny       21
#define k_fries_small      22
#define k_fries_medium     23
#define k_fries_large      24
#define k_fries_huge       25
#define k_fries_quantity   26
#define k_drink_list       27
#define k_drink_add        28
#define k_drink_sub        29
#define k_drink_quantity   30
#define k_total_order      31
#define k_burger_label     32
#define k_fries_label      33
#define k_drink_label      34
#define k_menu_bar         35
#define k_file_menu        36
#define k_edit_menu        37
#define k_order_menu       38
#define k_max_widget       38
#define MAX_WIDGETS (k_max_widget + 1)
#define NUM_BOOLEAN (k_burger_max - k_burger_min + 1)
#define k_burger_index    0
#define k_fries_index     1
#define k_drinks_index    2
#define k_index_count     3
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
static Widget toplevel_widget,
    main_window_widget,
    widget_array[MAX_WIDGETS];
static char toggle_array[NUM_BOOLEAN];
static DwtCompString current_drink,
    current_fries,
    name_vector[k_index_count];
static char * welcome_text_ptr;
static int quantity_vector[k_index_count];
static DwtCompString latin_create;
static DwtCompString latin_dismiss;
static DwtCompString latin_space;
static DwtCompString latin_zero;
static DwtCompString latin_separator;
static DRMHierarchy s_DRMHierarchy;
static DRMResourceContextPtr resource_ctx;
static DRMTypedef *dummy_class;
static char *db_filename_vec[] =
    {"decburger$text",
     "decburger.uid",
    };
static int db_filename_num =
    (sizeof db_filename_vec / sizeof db_filename_vec [0]);
static void s_error();
static void get_something();
static void set_something();
static void activate_proc();
static void create_proc();
static void list_proc();
static void quit_proc();
static void pull_proc();
static void scale_proc();
static void show_hide_proc();
static void show_label_proc();
static void toggle_proc();
static DRMRegisterArg reglist[] = {
    {"activate_proc", (caddr_t) activate_proc},
    {"create_proc", (caddr_t) create_proc},
    {"list_proc", (caddr_t) list_proc},
    {"pull_proc", (caddr_t) pull_proc},
    {"quit_proc", (caddr_t) quit_proc},
    {"scale_proc", (caddr_t) scale_proc},
    {"show_hide_proc", (caddr_t) show_hide_proc},
    {"show_label_proc", (caddr_t) show_label_proc},
    {"toggle_proc", (caddr_t) toggle_proc}
};
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
static int reglist_num = (sizeof reglist / sizeof reglist [0]);
void get_literal (lit, ptr, compound)
char * lit;
char * * ptr;
int compound;
{
    if ( DwtDrmHGetIndexedLiteral (
        s_DRMHierarchy,
        lit,
        resource_ctx) !=DRMSuccess) {
        s_error (lit);
        (* ptr) = NULL;
        return;
    }
    if (compound)
        (* ptr) = DwtLatin1String( DwtDrmRCBuffer (resource_ctx) );
    else
        (* ptr) = DwtDrmRCBuffer (resource_ctx);
}

unsigned int main(argc, argv)
unsigned int argc;
char *argv[];
{
    DwtInitializedDRM();

    if (DwtDrmGetResourceContext (
        NULL,
        NULL,
        100,
        & resource_ctx ) !=DRMSuccess)
        s_error ("can't get resource context");

    if (DwtOpenHierarchy(db_filename_num,
        db_filename_vec,
        NULL,
        &s_DRMHierarchy)
        !=DRMSuccess)
        s_error("can't open hierarchy");

    get_literal ("k_welcome_text", & welcome_text_ptr, 0);
    toplevel_widget = XtInitialize(welcome_text_ptr,
        "example",
        NULL,
        0,
        &argc,
        argv);
    init_application();

    DwtRegisterDRMNames(reglist, reglist_num);

    if (DwtFetchWidget(s_DRMHierarchy, "S_MAIN_WINDOW", toplevel_widget,
        &main_window_widget, &dummy_class) != DRMSuccess)
        s_error("can't fetch main window");

    XtManageChild(main_window_widget);
    XtRealizeWidget(toplevel_widget);

    XtMainLoop();
}
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
static int init_application()
{
    int k;

    for (k = 0; k < MAX_WIDGETS; k++)
        widget_array[k] = NULL;
    for (k = 0; k < NUM_BOOLEAN; k++)
        toggle_array[k] = FALSE;

    toggle_array[k_burger_medium - k_burger_min] = TRUE;

    get_literal ("k_apple_juice_text", & current_drink, 1);
    get_literal ("k_tiny_label_text", & current_fries, 1);

    get_literal ("k_create_order_label_text", & latin_create, 1);
    get_literal ("k_dismiss_order_label_text", & latin_dismiss, 1);
    latin_space = DwtLatin1String(" ");
    latin_separator = DwtLatin1String(": ");
    latin_zero = DwtLatin1String(" 0 ");
}

static void s_error(problem_string)
    char *problem_string;
{
    printf("%s\n", problem_string);
    exit(0);
}

static void set_something(w, resource, value)
    Widget w;
    char *resource, *value;
{
    Arg al[1];
    XtSetArg(al[0], resource, value);
    XtSetValues(w, al, 1);
}

static void get_something(w, resource, value)
    Widget w;
    char *resource, *value;
{
    Arg al[1];
    XtSetArg(al[0], resource, value);
    XtGetValues(w, al, 1);
}

static void set_boolean(i, state)
    int i;
    int state;
{
    toggle_array[i - k_burger_min] = state;
    DwtToggleButtonSetState(widget_array[i],

        state,
        FALSE);
}
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
static void update_drink_display()
{
    char drink_txt[50];
    sprintf(drink_txt, " %d ", quantity_vector[k_drinks_index]);
    set_something(widget_array[k_drink_quantity], DwtNlabel,
        DwtLatin1String(drink_txt));
}

static void reset_values()
{
    int i;
    for (i = k_burger_min; i <= k_burger_max; i++) {
        set_boolean(i, (i == k_burger_medium));
    }

    set_something(widget_array[k_burger_quantity], DwtNvalue, 0);
    quantity_vector[k_burger_index] = 0;

    DwtSTextSetString(widget_array[k_fries_quantity], " 0 ");

    set_something(widget_array[k_drink_quantity], DwtNlabel, latin_zero);
    quantity_vector[k_drinks_index] = 0;
}

static void clear_order()
{
    Arg arglist[5];
    int ac = 0;
    XtSetArg(arglist[ac], DwtNitemsCount, 0);
    ac++;
    XtSetArg(arglist[ac], DwtNitems, NULL);
    ac++;
    XtSetArg(arglist[ac], DwtNselectedItemsCount, 0);
    ac++;
    XtSetValues(widget_array[k_total_order], arglist, ac);
}

static void activate_proc(w, tag, reason)
Widget w;
int *tag;
unsigned long *reason;
{
    int widget_num = *tag;
    int i, value, fries_num;
    char *txt, *fries_text, *list_txt, list_buffer[20];
    switch (widget_num) {
        case k_nyi:
            if (widget_array[k_nyi] == NULL)
            {
                if (DwtFetchWidget(s_DRMHierarchy, "nyi", toplevel_widget,
                    &widget_array[k_nyi], &dummy_class) != DRMSuccess) {
                    s_error("can't fetch nyi widget");
                }
            }

            XtManageChild(widget_array[k_nyi]);
            break;
    }
}
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
case k_submit_order:
    clear_order();
    break;
case k_cancel_order:
    clear_order();
    break;
case k_dismiss:
    XtUnmanageChild(widget_array[k_order_box]);
case k_noapply:
    reset_values();
    break;
case k_apply:
    if (quantity_vector[k_burger_index] > 0) {
        list_txt = name_vector[k_burger_index];
        list_txt = DwtCStrcat(list_txt, latin_separator);

        sprintf(list_buffer, "%d ", quantity_vector[k_burger_index]);
        list_txt = DwtCStrcat(list_txt, DwtLatin1String(list_buffer));

        for (i = k_burger_min; i <= k_burger_max; i++)
            if (toggle_array[i - k_burger_min]) {
                get_something(widget_array[i], DwtNlabel, &txt);
                list_txt = DwtCStrcat(list_txt, txt);
                list_txt = DwtCStrcat(list_txt, latin_space);
            }

        DwtListBoxAddItem(widget_array[k_total_order], list_txt, 0);
    }

    fries_text = DwtSTextGetString(widget_array[k_fries_quantity]);
    fries_num = 0;
    sscanf(fries_text, "%d", &fries_num);
    if (fries_num != 0) {
        list_txt = name_vector[k_fries_index];
        list_txt = DwtCStrcat(list_txt, latin_separator);

        sprintf(list_buffer, "%d ", fries_num);
        list_txt = DwtCStrcat(list_txt, DwtLatin1String(list_buffer));

        list_txt = DwtCStrcat(list_txt, current_fries);

        DwtListBoxAddItem(widget_array[k_total_order], list_txt, 0);
    }

    if (quantity_vector[k_drinks_index] > 0) {
        list_txt = name_vector[k_drinks_index];
        list_txt = DwtCStrcat(list_txt, latin_separator);

        sprintf(list_buffer, "%d ", quantity_vector[k_drinks_index]);
        list_txt = DwtCStrcat(list_txt, DwtLatin1String(list_buffer));

        list_txt = DwtCStrcat(list_txt, current_drink);

        DwtListBoxAddItem(widget_array[k_total_order], list_txt, 0);
    }
    break;
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
    case k_fries_tiny:
    case k_fries_small:
    case k_fries_medium:
    case k_fries_large:
    case k_fries_huge:
        get_something(w, DwtNlabel, &current_fries);
        break;
    case k_drink_add:
        quantity_vector[k_drinks_index]++;
        update_drink_display();
        break;
    case k_drink_sub:
        if (quantity_vector[k_drinks_index] > 0)
            quantity_vector[k_drinks_index]--;
        update_drink_display();
        break;
    default:
        break;
}
}

static void toggle_proc(w, tag, toggle)
    Widget w;
    int *tag;
    DwtTogglebuttonCallbackStruct *toggle;
{
    toggle_array[*tag - k_burger_min] = toggle->value;
}

static void list_proc(w, tag, list)
    Widget w;
    int *tag;
    DwtListBoxCallbackStruct *list;
{
    current_drink = list->item;
}

static void scale_proc(w, tag, scale)
    Widget w;
    int *tag;
    DwtScaleCallbackStruct *scale;
{
    quantity_vector[k_burger_index] = scale->value;
}

static void show_hide_proc(w, tag, reason)
    Widget w;
    int *tag;
    unsigned long *reason;
{
    if (XtIsManaged(widget_array[k_order_box]))
        XtUnmanageChild(widget_array[k_order_box]);
    else
        XtManageChild(widget_array[k_order_box]);
}
}
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
static void show_label_proc(w, tag, reason)
    Widget w;
    int *tag;
    unsigned long *reason;
{
    if (widget_array[k_order_box] == NULL)
    {
        if (DwtFetchWidget(s_DRMHierarchy, "control_box", toplevel_widget,
            &widget_array[k_order_box], &dummy_class) != DRMSuccess) {
            s_error("can't fetch order box widget");
        }
    }

    if (XtIsManaged(widget_array[k_order_box]))
        set_something(widget_array[k_create_order], DwtNlabel, latin_dismiss);
    else
        set_something(widget_array[k_create_order], DwtNlabel, latin_create);
}

static void create_proc(w, tag, reason)
    Widget w;
    int *tag;
    unsigned long *reason;
{
    int widget_num = *tag;
    widget_array[widget_num] = w;

    switch (widget_num) {
        case k_burger_label:
            get_something(w, DwtNlabel, &name_vector[k_burger_index]);
            break;
        case k_fries_label:
            get_something(w, DwtNlabel, &name_vector[k_fries_index]);
            break;
        case k_drink_label:
            get_something(w, DwtNlabel, &name_vector[k_drinks_index]);
            break;
        default:
            break;
    }
}

static void quit_proc(w, tag, reason)
    Widget w;
    int *tag;
    unsigned long *reason;
{
    if (tag != NULL)
        printf("%s", tag);
    exit(1);
}

static void pull_proc(w, tag, reason)
    Widget w;
    int *tag;
    unsigned long *reason;
```

(continued on next page)

International Version of the DECburger Application

Example C-1 (Cont.) C Program for the International Version of DECburger

```
{
    int widget_num = *tag;
    switch (widget_num) {
        case k_file_pdme:
            if (widget_array[k_file_menu] == NULL) {
                if (DwtFetchWidget(s_DRMHierarchy, "file_menu", widget_array[
                    k_menu_bar], &widget_array[k_file_menu], &dummy_class) !=
                    DRMSuccess)
                    s_error("can't fetch file pulldown menu widget");
                set_something(widget_array[k_file_pdme], DwtNsubMenuId,
                    widget_array[k_file_menu]);
            }
            break;
        case k_edit_pdme:
            if (widget_array[k_edit_menu] == NULL) {
                if (DwtFetchWidget(s_DRMHierarchy, "edit_menu", widget_array[
                    k_menu_bar], &widget_array[k_edit_menu], &dummy_class) !=
                    DRMSuccess)
                    s_error("can't fetch edit pulldown menu widget");
                set_something(widget_array[k_edit_pdme], DwtNsubMenuId,
                    widget_array[k_edit_menu]);
            }
            break;
        case k_order_pdme:
            if (widget_array[k_order_menu] == NULL) {
                if (DwtFetchWidget(s_DRMHierarchy, "order_menu", widget_array[
                    k_menu_bar], &widget_array[k_order_menu], &dummy_class) !=
                    DRMSuccess)
                    s_error("can't fetch order pulldown menu widget");
                set_something(widget_array[k_order_pdme], DwtNsubMenuId,
                    widget_array[k_order_menu]);
                if (DwtFetchWidget(s_DRMHierarchy, "control_box",
                    toplevel_widget, &widget_array[k_order_box], &dummy_class) !=
                    DRMSuccess)
                    s_error("can't fetch order box widget");
            }
            if ( widget_array[k_order_box] == NULL )
                if (DwtFetchWidget (
                    s_DRMHierarchy,
                    "control_box",
                    toplevel_widget,
                    &widget_array [k_order_box],
                    &dummy_class) != DRMSuccess)
                    s_error ("can't fetch order box widget");
            if (XtIsManaged(widget_array[k_order_box]))
                set_something(widget_array[k_create_order], DwtNlabel,
                    latin_dismiss);
            else
                set_something(widget_array[k_create_order], DwtNlabel,
                    latin_create);
            break;
    }
}
```

D Building Your Own Widgets

The XUI Toolkit includes a set of widgets you can use to create the user interface of your application. You also have the option of building your own widgets.

This appendix describes how to build widgets.

D.1 Overview of Widgets

As defined in Chapter 1, a widget is a window packaged with input and output capabilities. In an application program, a widget is a collection of data and procedures, gathered into predefined data structures. The data defines characteristics of a widget, such as the width and height of the widget window. The procedures define the basic set of operations that can be performed on the widget, such as initialization, creation, and destruction.

The data and procedures that define a widget are collected in two data structures, defined by the X Toolkit: the widget class data structure and the widget instance data structure. The widget class data structure contains all the data and procedures that define a particular type, or **class**, of widget. Section D.2.1.1 describes this data structure. The widget instance data structure contains all the data and procedures that define an **instance** of a widget of a particular class. Section D.2.1.2 describes this data structure.

For example, consider an application that contains dozens of push button widgets. For this application, one widget class data structure contains the data and procedures common to all the push button widgets. However, the application includes one widget instance data structure for each push button widget it contains. The widget instance data structure defines those aspects of the push button widget that vary with each instance, such as size, position, and text content.

D.1.1 Building a Widget

You build a widget by defining the data and procedures that characterize the widget's appearance and functions. The X Toolkit defines a fundamental widget, called the **core** widget, on which all other widgets can be based. The core widget definition contains the basic set of data and procedures common to all widgets. To build a new widget, add the data and procedures that define the new widget to the core widget definition. In this way, you can build a widget without having to rewrite existing data and procedures.

Building Your Own Widgets

D.1 Overview of Widgets

Thus, the core widget and all of its descendants are related in a hierarchical structure. A new widget is called a **subclass** of the core widget. The core widget is called the **superclass** of the new widget. A subclass is said to **inherit** all the data and procedures of its superclass. (A subclass can also override a data field or procedure of its superclass.)

Note that the superclass/subclass relationship between widgets is different from the parent/child widget relationship described in Chapter 1. The relationship of widget classes defines the characteristics of a type of widget. For example, in the XUI Toolkit, the push button widget is a subclass of the label widget. Therefore, the push button widget contains all the data and procedures of the label widget plus the data and procedures unique to a push button widget. The parent/child relationship describes how instances of widgets are arranged to create user interfaces. The parent controls aspects of the appearance of its children. For example, in the Hello World! application introduced in Chapter 1, the dialog box widget is the parent of the label widget and the push button widget.

D.1.2 Building a Sample Widget

To illustrate widget building, this section introduces a sample widget: the simple push button widget. The simple push button widget does not attempt to duplicate the appearance or function of the XUI Toolkit push button widget (described in Section 5.4). For example, the simple push button widget does not accept compound strings and does not flash when activated by a user. However, this sample widget does provide a useful illustration of the process of widget building.

The following summarizes the steps required to build and test the sample widget:

- 1 Define the data and procedures that must be added to the core widget class data structure to provide the push button functions.

Example D-1 shows the source code for the sample widget. The example shows how a widget is constructed of data declarations and procedures. You must initialize the widget class data structure when you declare it.

- 2 Compile the widget.

The widget data declarations and procedure definitions are usually segregated into a separately compilable module. Use the command syntax summarized in Example D-3.

- 3 Link the widget with an application that uses it.

To test the sample widget, use it in an application. For example, Example D-2 shows the changes you would make to the Hello World! application to replace the XUI Toolkit push button widget with the simple push button widget. Example D-3 presents the syntax to perform the link operation.

- 4 Run the test program.

Building Your Own Widgets

D.1 Overview of Widgets

Example D-1 is the source file for the sample widget.

Example D-1 Sample Widget

```
①#include "decw$include:dwtwidget.h"

②static void Notify(), Destroy(), Initialize(), Redisplay(), set_gcs();
static Boolean Setvalues();
Widget SimplePushCreate();

③#define forGCmask      GCForeground | GCBackground | GCLineWidth
#define TheLabel(w)    ((w)->simplepush.label)

④typedef struct _SimplePushClass {
    DwtOffsetPtr simplepushoffsets;
    int reserved;
} SimplePushClassPart;

⑤typedef struct _SimplePushClassRec {
    CoreClassPart core_class;
    SimplePushClassPart simplepush_class;
} SimplePushClassRec, *SimplePushWidgetClass;

⑥typedef struct {
    char *label;
    XtCallbackList callback_list;
    GC gc;
} SimplePushPart;

⑦typedef struct _SimplePushRec {
    CorePart core;
    SimplePushPart simplepush;
} SimplePushRec, *SimplePushWidget;

⑧static XtResource resources[] = {
    {XtNcallback, XtCCallback, XtRcallback, sizeof(caddr_t),
     XtOffset(SimplePushWidget, simplepush.callback_list),
     XtRcallback, (caddr_t) NULL},
    {XtNlabel, XtCstring, XtRstring, sizeof(char*),
     XtOffset(SimplePushWidget, simplepush.label),
     XtRstring, NULL},
};

⑨static char defaultTranslations[] = "<Btn1Up>: notify()";

⑩static XtActionsRec actionsList[] = { {"notify", Notify} };
```

(continued on next page)

Building Your Own Widgets

D.1 Overview of Widgets

Example D-1 (Cont.) Sample Widget

```
11 SimplePushClassRec simplepushClassRec = {
  { /* Core Class Part */
    /* superclass */ (WidgetClass) &widgetClassRec,
    /* class_name */ "SimplePush",
    /* widget_size */ sizeof(SimplePushRec),
    /* class_initialize */ NULL,
    /* class_part_initialize*/ NULL,
    /* class_inited */ FALSE,
    /* initialize */ Initialize,
    /* initialize_hook */ NULL,
    /* realize */ XtInheritRealize,
    /* actions */ actionsList,
    /* num_actions */ XtNumber(actionsList),
    /* resources */ (XtResource *) resources,
    /* num_resources */ XtNumber(resources),
    /* xrm_class */ NULLQUARK,
    /* compress_motion */ TRUE,
    /* compress_exposure */ TRUE,
    /* compress_enterleave*/ TRUE,
    /* visible_interest */ FALSE,
    /* destroy */ Destroy,
    /* resize */ XtInheritResize,
    /* expose */ Redisplay,
    /* set_values */ (XtSetValuesFunc) SetValues,
    /* set_values_hook */ NULL,
    /* set_values_almost */ XtInheritSetValuesAlmost,
    /* get_values_hook */ NULL,
    /* accept_focus */ NULL,
    /* version */ XtVersionDontCheck,
    /* callback offsets */ NULL,
    /* tm_table */ defaultTranslations,
    /* geometry */ NULL,
    /* disp accelerators */ NULL,
    /* extension */ NULL,
  },

  { /* simplepush class record */
    /* */ NULL,
    /* */ NULL,
  }
};

12 WidgetClass simplepushWidgetClass = (WidgetClass) &simplepushClassRec;

13 static void set_gcs(w)
    SimplePushWidget w;
{
    XGCValues values;

    values.font = XLoadFont (XtDisplay( w ), "fixed");
    w->simplepush.gc = XtGetGC (w, GCFont, &values);
}
```

(continued on next page)

Building Your Own Widgets

D.1 Overview of Widgets

Example D-1 (Cont.) Sample Widget

```
void UpdateCallback (r, rstruct, s, sstruct, argname)
    Widget r;                                /* the real widget*/
    DwtCallbackStructPtr rstruct;           /* the real callback list */
    Widget s;                                /* the scratch widget*/
    DwtCallbackStructPtr sstruct;          /* the scratch callback list */
    char      *argname;
{
    DwtCallbackPtr list;

    /*
     * if a new callback has been specified in the scratch widget,
     * remove and deallocate old callback and init new
     */
    if (rstruct->ecallback != sstruct->ecallback)
    {
        list = (DwtCallbackPtr)sstruct->ecallback;
        /*
         * Copy the old callback list into the new widget, since
         * XtRemoveCallbacks needs the "real" widget
         */
        *sstruct = *rstruct;
        XtRemoveAllCallbacks(s, argname);
        sstruct->ecallback = NULL;
        XtAddCallbacks(s, argname, list);
    }
}

14 static void Initialize(req, new)
    Widget req,          /* as built from arglist */
    new;                /* as modif by superclasses */
{
    SimplePushWidget w = (SimplePushWidget) new;

    /*
     * Handle the string and set widget gcs
     */
    if (TheLabel (w) == NULL)
        TheLabel (w) = strcpy (XtMalloc ((unsigned)
            strlen (w->core.name) + 1),
            w->core.name);
    else
        TheLabel (w) = strcpy (XtMalloc ((unsigned)
            strlen (TheLabel(w)) + 1),
            TheLabel (w));

    set_gcs(w);
}

15 static void Redisplay(w, event, region)
    SimplePushWidget w;
    XEvent *event;
    Region region;
{
    if (XtIsRealized (w))
```

(continued on next page)

Building Your Own Widgets

D.1 Overview of Widgets

Example D-1 (Cont.) Sample Widget

```
    XDrawString (XtDisplay (w),
                XtWindow (w),
                w->simplepush.gc,
                1,
                XtHeight (w)/2,
                TheLabel (w),
                strlen( TheLabel(w) ));
}

16static void Notify(w, event)
    SimplePushWidget w;
    XEvent *event;
{
    int data = 0;

    XtCallCallbacks(w, XtNcallback, data);
}

17static Boolean SetValues(old, request, new)
    SimplePushWidget old, request, new;
{
    Boolean redisplay = FALSE;

    UpdateCallback(old,
                  &(old->simplepush.callback_list),
                  new,
                  &(new->simplepush.callback_list),
                  XtNcallback);

    /*
     * if the address and/or contents of the label have been modified,
     * free old string and store new. Also must redisplay new label.
     */
    if (TheLabel(old) != TheLabel(new)
        ||
        strcmp(TheLabel(old), TheLabel(new)) != 0)
    {
        XtFree (TheLabel(old));
        TheLabel(old) = NULL;

        TheLabel(new) = strcpy (XtMalloc ((unsigned)
                                         strlen (TheLabel(new)) + 1),
                               TheLabel (new));

        redisplay = TRUE;
    }

    return (Boolean)(redisplay);
}

18static void Destroy(w)
    SimplePushWidget w;
{
    XtFree( TheLabel(w) );
    XtRemoveAllCallbacks(w, XtNcallback);
}
```

(continued on next page)

Building Your Own Widgets

D.1 Overview of Widgets

Example D-1 (Cont.) Sample Widget

```
Widget SimplePushCreate(parent, name, args, argCount)
Widget parent;
char *name;
ArgList args;
int argCount;
{
    return ( XtCreateWidget( name,
                             simplepushWidgetClass,
                             parent,
                             args,
                             argCount ) );
}
```

- ① Symbol definition file used by widget builders.
- ② Forward declarations of procedures defined in the module.
- ③ Constant definitions used by the widget.
- ④ This statement defines that part of the class record that is unique to the simple push button widget. This is called the class part definition. In the example, this includes two fields.
- ⑤ After defining the new fields of the simple push button widget class, the sample widget appends these new fields to the class part record of the core widget. Section D.2.1.1 describes this structure.
- ⑥ The sample widget defines the part of the widget instance record unique to the simple push button widget. The sample widget defines three new fields:
 - The text string the simple push button widget will contain
 - The list of callback routines
 - A graphics context data structure
- ⑦ After defining the new fields in the instance record, the sample widget appends these new fields to the core widget instance record. The core widget instance record is described in Section D.2.1.2.
- ⑧ This is a declaration of an array of XtResource data structures, with initialization of the data structures. The XtResource data structure is defined by the X Toolkit (see Section D.8.1). This array defines which aspects of the simple push button widget can be set at widget creation time.
- ⑨ The translation table for the simple push button widget defines the action the widget performs in response to user interaction. This translation table specifies that when the user clicks MB1 in the simple push button widget, the procedure *notify()* should be executed. Section D.9.3 describes the elements of a translation table.

Building Your Own Widgets

D.1 Overview of Widgets

⑩ The action table maps the name of the action procedure to the address of the action procedure. Here the *notify()* procedure is associated with the text string "notify". Section D.9.1 describes the elements of an action table.

⑪ The fields in the structure that define the simple push button widget class are initialized. This structure contains all the fields from the core widget class definition plus the new fields defined by the simple push button widget. Note that the first field in the structure points to the class definition of the superclass of this widget.

By initializing this structure, you determine which data and procedures are inherited from the core widget class and which data and procedures are new with the simple push button widget class. Section D.2.1 describes the core widget definition. The simple push button widget inherits most of the procedures except the initialize, destroy, and expose procedures.

⑫ This statement declares a static pointer to the simple push button widget class.

⑬ The next two procedures are local procedures used by the *initialize()* procedure. These procedures set the graphics context for displaying text.

⑭ This is the simple push button widget instance initialization procedure. It is called every time an instance of this widget is created. Section D.4.1 describes this procedure.

⑮ This is the simple push button widget expose procedure. This procedure is called whenever the widget becomes visible. Section D.7.2.1 describes this procedure.

⑯ This is the action routine for the widget, named *notify()*. This procedure is listed in the translation table for this widget.

⑰ This is the simple push button widget set values procedure. This procedure is called when an application program calls the intrinsic routine SET VALUES. The set values procedure, described in Section D.8.7.1, updates widget-specific attributes.

⑱ This is the simple push button widget destroy procedure. This procedure, described in Section D.4.5, destroys an instance of the widget.

⑲ The simple push button widget defines a low-level creation routine that you can use to create instances of the widget.

Example D-2 contains the changes you must make to the Hello World! sample application that appears in Example 2-14 to replace the XUI Toolkit push button widget with the sample widget.

Building Your Own Widgets

D.1 Overview of Widgets

Example D-2 Modifying the Hello World! Application to Use the Sample Widget

```
.  
. .  
① Arg arglist[15];  
. .  
XtSetArg (arglist[0], XtNx, 10) ;  
XtSetArg (arglist[1], XtNy, 40);  
XtSetArg (arglist[2], XtNcallback, callback_arg) ;  
② XtSetArg (arglist[3], XtNlabel, " Hello World!") ;  
③ XtSetArg (arglist[4], XtNwidth, 84);  
XtSetArg (arglist[5], XtNheight, 25);  
④ button = SimplePushCreate( helloworldmain, "button", arglist, 6 );  
. .  
⑤ XtSetArg( arglist[0], XtNlabel, "Goodbye World!");  
XtSetValues( widget, arglist, 1 );  
. .
```

- ① Increase the size of the array of argument data structures.
- ② The sample widget does not use compound strings.
- ③ The width and height of the sample widget are 0, by default, so you must supply explicit values for these attributes.
- ④ Create the sample widget using the low-level routine declared in the widget.
- ⑤ Change the compound string used in the callback routine to a standard text string. Also, because the size of the simple push button is fixed, the repositioning done in the original Hello World! example has been removed.

Example D-3 presents the command syntax to compile, link, and run the sample widget and test program.

Building Your Own Widgets

D.1 Overview of Widgets

Example D-3 Compiling and Linking the Sample Widget

```
$ CC SAMPLE_WIDGET.C
$ CC NEW_HELLOWORLD.C
$ LINK/NODEB NEW_HELLOWORLD, SAMPLE_WIDGET, SYSSINPUT/OPT
SYSSLIBRARY:DECW$DWTLIBSHR/SHARE
[Ctrl/Z]
$ RUN NEW_HELLOWORLD
```

D.2 Widget Class Definitions

Every widget belongs to exactly one widget class that is statically allocated and initialized and that contains the operations allowable on widgets of that class. Logically, a widget class is the procedures and data that is associated with all widgets belonging to that class. These procedures and data can be inherited by subclasses. Physically, a widget class is a pointer to a structure. The contents of this structure are constant for all widgets of the widget class but will vary from class to class. Here, constant means that the class structure is initialized at compile time and never changed, except for a one-time class initialization and in-place compilation of resource lists, which takes place when the first widget of the class or subclass is created.

The organization of the declarations and code for a new widget class between a public .h file, a private .h file, and the implementation .c file is described in Section D.3. The predefined widget classes adhere to these conventions.

A widget instance is composed of two parts:

- A data structure that contains instance-specific values
- A class structure that contains information applicable to all widgets of that class

Much of the input/output of a widget (for example, fonts, colors, sizes, border widths, and so on) is customizable by users.

The next three sections discuss the base widget classes:

- Core widgets
- Composite widgets
- Constraint widgets

D.2.1 Core Widgets

The core widget contains definitions of fields common to all widgets. All widgets are subclasses of the core widget, which is defined by the CoreClassPart and CorePart structures.

D.2.1.1 CoreClassPart Structure

The common fields for all widget classes are defined in the CoreClassPart structure, as follows:

```
typedef struct {
    WidgetClass superclass;
    String class_name;
    Cardinal widget_size;
    XtProc class_initialize;
    XtWidgetClassProc class_part_initialize;
    Boolean class_inited;
    XtInitProc initialize;
    XtArgsProc initialize_hook;
    XtRealizeProc realize;
    XtActionList actions;
    Cardinal num_actions;
    XtResourceList resources;
    Cardinal num_resources;
    XrmClass xrm_class;
    Boolean compress_motion;
    Boolean compress_exposure;
    Boolean compress_enterleave;
    Boolean visible_interest;
    XtWidgetProc destroy;
    XtWidgetProc resize;
    XtExposeProc expose;
    XtSetValuesFunc set_values;
    XtArgsFunc set_values_hook;
    XtAlmostProc set_values_almost;
    XtArgsProc get_values_hook;
    XtAcceptFocusProc accept_focus;
    XtVersionType version;
    _XtOffsetList callback_private;
    String tm_table;
    XtGeometryHandler query_geometry;
    XtStringProc display_accelerator;
    caddr_t extension;
} CoreClassPart;
```

All widget classes have the core class fields as their first component. The prototypical WidgetClass is defined with only this set of fields. Various routines can cast widget class pointers, as needed, to specific widget class types, as shown in the following example:

```
typedef struct {
    CoreClassPart core_class;
} WidgetClassRec, *WidgetClass;
```

The predefined class record and pointer for WidgetClassRec are:

```
extern WidgetClassRec widgetClassRec;
extern WidgetClass widgetClass;
```

The opaque types Widget and WidgetClass and the opaque variable widgetClass are defined for generic actions on widgets.

Building Your Own Widgets

D.2 Widget Class Definitions

D.2.1.2 CorePart Structure

The common fields for all widget instances are defined in the CorePart structure, as follows:

```
typedef struct _CorePart {
    Widget self;
    WidgetClass widget_class;
    Widget parent;
    XrmName xrm_name;
    Boolean being_destroyed;
    XtCallbackList destroy_callbacks;
    caddr_t constraints;
    Position x;
    Position y;
    Dimension width;
    Dimension height;
    Dimension border_width;
    Boolean managed;
    Boolean sensitive;
    Boolean ancestor_sensitive;
    XtEventTable event_table;
    XtTMRRec tm;
    XtTranslations accelerators;
    Pixel border_pixel;
    Pixmap border_pixmap;
    WidgetList popup_list;
    Cardinal num_popups;
    String name;
    Screen *screen;
    Colormap colormap;
    Window window;
    Cardinal depth;
    Pixel background_pixel;
    Pixmap background_pixmap;
    Boolean visible;
    Boolean mapped_when_managed;
} CorePart;
```

All widget instances have the core fields as their first component. The prototypical type Widget is defined with only this set of fields. Various routines can cast widget pointers, as needed, to specific widget types, as shown in the following example:

```
typedef struct {
    CorePart core;
} WidgetRec, *Widget;
```

D.2.1.3 CorePart Default Values

The default values for the core widget fields, which are filled in by the core widget resource list and the core widget initialize procedure, are listed in Table D-1.

Building Your Own Widgets

D.2 Widget Class Definitions

Table D-1 Default Values for the CorePart Structure

Field	Default Value
self	Address of the widget structure (cannot be changed)
widget_class	The widget_class argument to the CREATE WIDGET intrinsic routine (cannot be changed)
parent	The parent argument to the CREATE WIDGET intrinsic routine (cannot be changed)
xrm_name	Encoded name argument to the CREATE WIDGET intrinsic routine (cannot be changed)
being_destroyed	Parent's being_destroyed value
destroy_callbacks	Null
constraints	Null
x	0
y	0
width	0
height	0
border_width	1
managed	False
sensitive	True
ancestor_sensitive	Bitwise AND of sensitive and ancestor_sensitive fields of the parent widget's CorePart structure
event_table	Initialized by the event manager
tm	Initialized by the translation manager
accelerators	Null
border_pixel	XtDefaultForeground
border_pixmap	Null
popup_list	Null
num_popups	0
name	The name argument to the CREATE WIDGET intrinsic routine (cannot be changed)
screen	Parent's screen; top-level widget gets it from display specifier (cannot be changed)
colormap	Default color map for the screen
window	Null
depth	Parent's depth; top-level widget gets root window depth
background_pixel	XtDefaultBackground
background_pixmap	Null
visible	True
map_when_managed	True

Building Your Own Widgets

D.2 Widget Class Definitions

D.2.2 Composite Widgets

Composite widgets are a subclass of the core widget (see Section D.5). They are intended to be containers for other widgets and are defined by the CompositeClassPart and CompositePart structures.

D.2.2.1 CompositeClassPart Structure

In addition to the core widget class fields, composite widgets have the following class fields:

```
typedef struct {
    XtGeometryHandler geometry_manager;
    XtWidgetProc change_managed;
    XtWidgetProc insert_child;
    XtWidgetProc delete_child;
    caddr_t extension;
} CompositeClassPart;
```

Composite widget classes have the composite fields immediately following the core fields, as shown in the following example:

```
typedef struct {
    CoreClassPart          core_class;
    CompositeClassPart     composite_class;
} CompositeClassRec, *CompositeWidgetClass;
```

The predefined class record and pointer for CompositeClassRec are:

```
extern CompositeClassRec compositeClassRec;
extern WidgetClass compositeWidgetClass;
```

The opaque types CompositeWidget and CompositeWidgetClass and the opaque variable compositeWidget are defined for generic operations on widgets that are a subclass of CompositeWidget.

D.2.2.2 CompositePart Structure

In addition to the CorePart fields, composite widgets have the following fields defined in the CompositePart structure:

```
typedef struct {
    WidgetList children;
    Cardinal num_children;
    Cardinal num_slots;
    XtOrderProc insert_position;
} CompositePart;
```

Composite widgets have the composite fields immediately following the core fields, as shown in the following example:

```
typedef struct {
    CorePart core;
    CompositePart composite;
} CompositeRec, *CompositeWidget;
```


D.2.2.3 CompositePart Default Values

The default values for the composite fields, which are filled in by the composite widget resource list and the composite widget initialize procedure, are listed in Table D-2.

Table D-2 Default Values for the CompositePart Structure

Field	Default Value
children	Null
num_children	0
num_slots	0
insert_position	Internal function InsertAtEnd

D.2.3 Constraint Widgets

Constraint widgets are a subclass of the Composite widget (see Section D.5.4) that maintain additional state data for each child, such as client-defined constraints on the child's geometry. They are defined by the ConstraintClassPart and ConstraintPart structures.

D.2.3.1 ConstraintClassPart Structure

In addition to the composite class fields, constraint widgets have the following class fields:

```
typedef struct {
    XtResourceList resources;
    Cardinal num_resources;
    Cardinal constraint_size;
    XtInitProc initialize;
    XtWidgetProc destroy;
    XtSetValuesFunc set_values;
    caddr_t extension;
} ConstraintClassPart;
```

Constraint widget classes have the constraint fields immediately following the composite fields, as follows:

```
typedef struct {
    CoreClassPart          core_class;
    CompositeClassPart     composite_class;
    ConstraintClassPart     constraint_class;
} ConstraintClassRec, *ConstraintWidgetClass;
```

The predefined class record and pointer for ConstraintClassRec are:

```
extern ConstraintClassRec constraintClassRec;
extern WidgetClass constraintWidgetClass;
```

The opaque types ConstraintWidget and ConstraintWidgetClass and the opaque variable constraintWidgetClass are defined for generic operations on widgets that are a subclass of ConstraintWidgetClass.

Building Your Own Widgets

D.2 Widget Class Definitions

D.2.3.2 ConstraintPart Structure

In addition to the CompositePart fields, Constraint widgets have the following fields defined in the ConstraintPart structure:

```
typedef struct { int empty; } ConstraintPart;
```

Constraint widgets have the constraint fields immediately following the composite fields, as follows:

```
typedef struct {  
    CorePart core;  
    CompositePart composite;  
    ConstraintPart constraint;  
} ConstraintRec, *ConstraintWidget;
```

D.3 Widget Classing

The widget_class field of a widget points to its widget class structure, which contains information that is constant across all widgets of that class. As a consequence, widget classes usually do not implement directly callable procedures; rather, they implement procedures that are available through their widget class structure. These methods are invoked by generic procedures that envelop common actions around the procedures implemented by the widget class. Such procedures are applicable to all widgets of that class and also to widgets that are subclasses of that class.

All widget classes are a subclass of the core widget. These widget classes can, in turn, have subclasses. Subclasses reduce the amount of code and declarations you write to make a new widget class that is similar to an existing class. For example, you do not have to describe every resource your widget uses in an XtResourceList. Instead, you describe only the resources your widget has that its superclass does not. Subclasses usually inherit many of their superclass's procedures (for example, the expose procedure or geometry handler).

Subclassing, however, can be taken too far. If you create a subclass that inherits none of the procedures of its superclass, you should consider whether or not you have chosen the most appropriate superclass.

To make good use of subclassing, widget declarations and naming conventions are highly stylized. A widget declaration consists of three files:

- A public .h file that is used by client widgets or applications
- A private .h file that is used by widgets that are subclasses of the widget
- A .c file that implements the widget class

D.3.1 Widget Naming Conventions

Intrinsics routines provide a vehicle by which programmers can create new widgets and organize a collection of widgets into an application. To ensure that applications need not deal with as many styles of capitalization and spelling as the number of widget classes it uses, use the following guidelines when writing new widgets:

- Use the X naming conventions that are applicable (see the *VMS DECwindows Guide to Xlib Programming: MIT C Binding*). For example, a record component name is all lowercase and uses underscores (`_`) for compound words (for example, `background_pixmap`). Type and procedure names start with a capital letter and use initial capitalization for compound words (for example, `ArgList` or `XtSetValues`).
- A resource name string is spelled identically to the field name except that compound names use initial capitalization rather than an underscore. To let the compiler catch spelling errors, each resource name should have a macro definition prefixed with `XtN`. For example, the `background_pixmap` field has the corresponding resource name identifier `XtNbackgroundPixmap`, which is defined as the string "backgroundPixmap". Many predefined names are listed in `<X11/StringDefs.h>`. Before you create a new name, make sure that your proposed name is not already defined, or that there is not already a name that you can use.
- A resource class string starts with a capital letter and uses initial capitalization for compound names (for example, `BorderWidth`). Each resource class string should have a macro definition prefixed with `XtC` (for example, `XtCBorderWidth`).
- A resource representation string is spelled identically to the type name (for example, `TranslationTable`). Each representation string should have a macro definition prefixed with `XtR` (for example, `XtRTranslationTable`).
- New widget classes start with a capital letter and use initial capitalization for compound words. For example, given a new class name `AbcXyz`, you should derive several names:
 - Partial widget instance structure name `AbcXyzPart`
 - Complete widget instance structure names `AbcXyzRec` and `_AbcXyzRec`
 - Widget instance pointer type name `AbcXyzWidget`
 - Partial class structure name `AbcXyzClassPart`
 - Complete class structure names `AbcXyzClassRec` and `_AbcXyzClassRec`
 - Class structure variable `abcXyzClassRec`
 - Class pointer variable `abcXyzWidgetClass`

Building Your Own Widgets

D.3 Widget Classing

- Action procedures available to translation specifications should follow the same naming conventions as procedures. That is, they start with a capital letter, and compound names use initial capitalization (for example, Highlight and NotifyClient).

D.3.2 Widget Subclassing in Public .h Files

The public .h file for a widget class is imported by clients and contains the following:

- A reference to the public .h files for the superclass
- The names and classes of the new resources that this widget adds to its superclass
- The class record pointer that you use to create widget instances
- The C type that you use to declare widget instances of this class
- Entry points for new class methods

For example, the following is the public .h file for a possible implementation of a label widget:

```
#ifndef LABEL_H
#define LABEL_H

/* New resources */
#define XtNjustify"justify"
#define XtNforeground"foreground"
#define XtNlabel "label"
#define XtNfont "font"
#define XtNinternalWidth"internalWidth"
#define XtNinternalHeight"internalHeight"

/* Class record pointer */
extern WidgetClass labelWidgetClass;

/* C Widget type definition */
typedef struct _LabelRec *LabelWidget;

/* New class method entry points */
extern void Label SetText();
    /* Widget w */
    /* String text */

extern String Label GetText();
    /* Widget w */

#endif LABEL_H
```

The conditional inclusion of the text allows the application to include header files for different widgets without being concerned that they already may be included as a superclass of another widget.

To accommodate operating systems with file name length restrictions, the name of the public .h file is the first 10 characters of the widget class. For example, the public .h file for the Constraint widget is Constraint.h.

D.3.3 Widget Subclassing in Private .h Files

The private .h file for a widget is imported by widget classes that are subclasses of the widget and contains the following:

- A reference to the public .h file for the class
- A reference to the private .h file for the superclass
- The new fields that the widget instance adds to its superclass's widget structure
- The complete widget instance structure for this widget
- The new fields that this widget class adds to its superclass's Constraint structure if the widget is a subclass of Constraint
- The complete Constraint structure if the widget is a subclass of Constraint
- The new fields that this widget class adds to its superclass's widget class structure
- The complete widget class structure for this widget
- The name of a constant of the generic widget class structure
- An inherit procedure for subclasses that want to inherit a superclass operation for each new procedure in the widget class structure

For example, the following is the private .h file for a label widget:

```
#ifndef LABELP_H
#define LABELP_H

#include <X11/Label.h>

/* New fields for the Label widget record */
typedef struct {
/* Settable resources */
    Pixel foreground;
    XFontStruct *font;
    String label;           /* text to display */
    XtJustify justify;
    Dimension internal_width; /* # of pixels horizontal border */
    Dimension internal_height; /* # of pixels vertical border */

/* Data derived from resources */
    GC normal_GC;
    GC gray_GC;
    Pixmap gray_pixmap;
    Position label_x;
    Position label_y;
    Dimension label_width;
    Dimension label_height;
    Cardinal label_len;
    Boolean display_sensitive;
} LabelPart;

/* Full instance record declaration */
typedef struct _LabelRec {
    CorePart core;
    LabelPart label;
} LabelRec;
```

Building Your Own Widgets

D.3 Widget Classing

```
/* Types for label class methods */
typedef void (*LabelSetTextProc)();
/* Widget w */
/* String text */

typedef String (*LabelGetTextProc)();
/* Widget w */

/* New fields for the Label widget class record */
typedef struct {
    LabelSetTextProc set_text;
    LabelGetTextProc get_text;
    caddr_t extension;
} LabelClassPart;

/* Full class record declaration */
typedef struct _LabelClassRec {
    CoreClassPart core_class;
    LabelClassPart label_class;
} LabelClassRec;

/* Class record variable */
extern LabelClassRec labelClassRec;

#define LabelInheritSetText ((LabelSetTextProc) _XtInherit)
#define LabelInheritGetText ((LabelGetTextProc) _XtInherit)
#endif LABELP_H
```

To accommodate operating systems with file name length restrictions, the name of the private .h file is the first nine characters of the widget class followed by an uppercase P. For example, the private .h file for the Constraint widget is ConstraintP.h.

D.3.4 Widget Subclassing in .c Files

The .c file for a widget contains the structure initializer for the class record variable, which contains the following types of fields:

- Class information (for example, superclass, class_name, widget_size, class_initialize, and class_init)
- Data constants (for example, resources and num_resources, actions and num_actions, visible_interest, compress_motion, compress_exposure, and version)
- Widget operations (for example, initialize, realize, destroy, resize, expose, set_values, accept_focus, and any operations specific to the widget)

The superclass field points to the superclass WidgetClass record. For direct subclasses of the generic core widget, the superclass should be initialized to the address of the widgetClassRec structure. The superclass is used for class chaining operations (see Section D.3.5) and for inheriting or enveloping a superclass's operations.

The class_name field contains the text name for this class (used by the resource manager). For example, the label widget has the string "Label". More than one widget class can share the same text class name.

The widget_size field is the size of the corresponding widget structure (not the size of the class structure).

Building Your Own Widgets

D.3 Widget Classing

The version field indicates the X Toolkit version number and is used for run-time consistency checking of the intrinsics and widgets in an application. Widget writers must set the version field to the symbolic value `XtVersion` in the widget class initialization. Widget writers who know that their widgets are backward compatible with previous versions of the intrinsics can put the special value `XtVersionDontCheck` in the version field to turn off version checking for those widgets.

The extension field is for future upward compatibility. If you add additional fields to class parts, all subclass structure layouts change, requiring complete recompilation. To allow clients to avoid recompilation, an extension field at the end of each class part can point to a record that contains any additional class information required.

All other fields are described in their respective sections.

Example D-4 is an abbreviated version of the `.c` file for the label widget. (The resource table is described in Section D.8.)

Example D-4 The `.c` File for a Label Widget

```
/* Resources specific to Label */
#define XtRJustify      "Justify"
static XtResource resources[] = {
    {XtNforeground, XtCForeground, XtRPixel, sizeof(Pixel),
     XtOffset(LabelWidget, label.foreground), XtRString, XtDefaultForeground},
    {XtNfont, XtCFont, XtRFontStruct, sizeof(XFontStruct *),
     XtOffset(LabelWidget, label.font), XtRString, XtDefaultFont},
    {XtNlabel, XtCLabel, XtRString, sizeof(String),
     XtOffset(LabelWidget, label.label), XtRString, NULL},
    .
    .
}

/* Forward declarations of procedures */
static void ClassInitialize();
static void Initialize();
static void Realize();
static void SetText();
static void GetText();
.
.
.
```

(continued on next page)

Building Your Own Widgets

D.3 Widget Classing

Example D-4 (Cont.) The .c File for a Label Widget

```
/* Class record constant */
LabelClassRec labelClassRec = {
    {
        /* core_class fields */
        /* superclass */      (WidgetClass) &widgetClassRec,
        /* class_name */      "Label",
        /* widget_size */     sizeof(LabelRec),
        /* class_initialize */ /*ClassInitialize,
        /* class_part_initialize *//NULL,
        /* class_inited */    /*False,
        /* initialize */      Initialize,
        /* initialize_hook */ /*NULL,
        /* realize */         Realize,
        /* actions */        NULL,
        /* num_actions */    0,
        /* resources */      resources,
        /* num_resources */  /*XtNumber(resources),
        /* xrm_class */      NULLQUARK,
        /* compress_motion */ /*True,
        /* compress_exposure *//True,
        /* compress_enterleave *//True,
        /* visible_interest *//False,
        /* destroy */        NULL,
        /* resize */         Resize,
        /* expose */         Redisplay,
        /* set_values */     SetValue,
        /* set_values_hook */ /*NULL,
        /* set_values_almost *//XtInheritSetValuesAlmost,
        /* get_values_hook */ /*NULL,
        /* accept_focus */   /*NULL,
        /* version */        XtVersion,
        /* callback_offsets *//NULL,
        /* tm_table */       NULL,
        /* query_geometry */ /*XtInheritQueryGeometry,
        /* display_accelerator *//NULL,
        /* extension */      NULL
    },
    {
        /* Label_class fields */
        /* get_text */       GetText,
        /* set_text */       SetText,
        /* extension */      NULL
    }
};

/* Class record pointer */
WidgetClass labelWidgetClass = (WidgetClass) &labelClassRec;
```

(continued on next page)

Example D-4 (Cont.) The .c File for a Label Widget

```
/* New method access routines */
void Label SetText(w, text)
    Widget w;
    String text;
{
    Label WidgetClass lwc = (Label WidgetClass)XtClass(w);
    XtCheckSubclass(w, labelWidgetClass, NULL);
    *(lwc->label_class.set_text)(w, text)
}
/* Private procedures */
.
.
.
```

D.3.5 Superclass Chaining

While most fields in a widget class structure are self-contained, some fields are linked to their corresponding field in their superclass or subclass structures. With a linked field, the intrinsics access the value in the widget class field only after accessing the value in the corresponding superclass field (called downward superclass chaining) or before accessing the value in the corresponding superclass field (called upward superclass chaining). The self-contained fields in a widget class are:

- The class_name field
- The class_initialize field
- The widget_size field
- The realize field
- The visible_interest field
- The resize field
- The expose field
- The accept_focus field
- The compress_motion field
- The compress_exposure field
- The compress_enterleave field
- The set_values_almost field
- The tm_table field
- The version field

With downward superclass chaining, the invocation of an operation first accesses the field from the Core class structure, then the subclass structure, and so on down the class chain to that widget's class structure.

Building Your Own Widgets

D.3 Widget Classing

These superclass-to-subclass fields are:

- The `class_part_initialize` field
- The `get_values_hook` field
- The `initialize` field
- The `initialize_hook` field
- The `set_values` field
- The `set_values_hook` field
- The `resources` field

In addition, for subclasses of the constraint widget, the `resources` field of the `ConstraintClassPart` structure is chained from the `Constraint` class down to the subclass.

With upward superclass chaining, the invocation of an operation first accesses the field from the widget class structure, then the field from the superclass structure, and so on up the class chain to the `Core` class structure. The subclass-to-superclass fields are:

- The `destroy` field
- The `actions` field

D.3.6 Class Initialization

Many class records can be initialized completely at compile time. In some cases, however, a class may need to register type converters or perform other sorts of one-time initialization.

Because the C language does not have initialization procedures that are invoked automatically when a program starts up, a widget class can declare a class initialize procedure that will be automatically called exactly once by the X Toolkit. A class initialization procedure is of type `XtProc`, as follows:

```
typedef void (*XtProc)();
```

A widget class indicates that it has no class initialization procedure by specifying null in the `class_initialize` field.

In addition to having class initializations done exactly once, some classes need to perform additional initialization for fields in their part of the class record. These are performed for the particular class and for subclasses as well. This is done in the class part initialization procedure, which is stored in the `class_part_initialize` field and is of type `XtWidgetClassProc`, as follows:

```
typedef void (*XtWidgetClassProc)(WidgetClass);  
WidgetClass widgetClass;
```

During class initialization, the class part initialization procedures for the class and all its superclasses are called in superclass-to-subclass order on the class record. These procedures have the responsibility of doing any dynamic initializations necessary to their class's part of the

record. The most common task is the resolution of any inherited methods defined in the class. For example, if a widget class C has superclasses Core, Composite, A, and B, the class record for C is first passed to Core's `class_part_initialize` record. This resolves any inherited core methods and compiles the textual representations of the resource list and action table that are defined in the class record. Next, the Composite's class part initialize procedure is called to initialize the composite part of C's class record. Finally, the class part initialize procedures for A, B, and C (in order) are called. Classes that do not define any new class fields or that need no extra processing can specify null in the `class_part_initialize` field.

All widget classes, whether they have a class initialization procedure or not, must start with their `class_inited` field set to false.

The first time a widget of a class is created, the `CREATE WIDGET` intrinsic routine ensures that the widget class and all superclasses are initialized, in superclass to subclass order, by checking each `class_inited` field and, if it is set to false, by then calling the class initialize and the class part initialize procedures for the class and all its superclasses. The intrinsics then set the `class_inited` field to true. After the one-time initialization, a class structure is constant.

The following provides the class initialization procedure for the label widget described in Example D-4.

```
static void ClassInitialize()
{
    XtQEleft   = XrmStringToQuark("left");
    XtQEcenter = XrmStringToQuark("center");
    XtQEright  = XrmStringToQuark("right");

    XtAddConverter(XtRString, XtRJustify, CvtStringToJustify, NULL, 0);
}
```

A class is initialized the first time a widget of that class or any subclass is created. If the class initialization procedure registers type converters, they are not available until this first widget is created.

D.3.7 Inheritance of Superclass Operations

A widget class is free to use any of its superclass's self-contained operations rather than implementing its own code. The most frequently inherited operations are:

- The `expose` operation
- The `realize` operation
- The `insert_child` operation
- The `delete_child` operation
- The `geometry_manager` operation
- The `set_values_almost` operation

For example, to inherit an `expose` operation, specify the constant `XtInheritExpose` in your class record.

Building Your Own Widgets

D.3 Widget Classing

Every class that declares a new procedure in its widget class part must provide for inheriting the procedure in its class part initialize procedure. (The special chained operations initialize, set values, and destroy declared in the Core record do not have inherited procedures. Widget classes that do nothing beyond what their superclass does specify null for chained procedures in their class records.)

Inheriting works by comparing the value of the field with a known, special value. If a match occurs, the superclass's value for that field is copied. This special value is usually the internal value `_XtInherit` cast to the appropriate type. (`_XtInherit` is a procedure that issues an error message if it is called.)

For example, the Composite class's private include file contains the following definitions:

```
#define XtInheritGeometryManager ((XtGeometryHandler) _XtInherit)
#define XtInheritChangeManaged ((XtWidgetProc) _XtInherit)
#define XtInheritInsertChild ((XtArgsProc) _XtInherit)
#define XtInheritDeleteChild ((XtWidgetProc) _XtInherit)
```

The composite widget's class part initialize procedure begins as follows:

```
static void CompositeClassPartInitialize(widgetClass)
WidgetClass widgetClass;
{
    register CompositeWidgetClass wc = (CompositeWidgetClass) widgetClass;
    CompositeWidgetClass super = (CompositeWidgetClass) wc->core_class.superclass;
    if (wc->composite_class.geometry_manager == XtInheritGeometryManager) {
        wc->composite_class.geometry_manager =
            super->composite_class.geometry_manager;
    }
    if (wc->composite_class.change_managed == XtInheritChangeManaged) {
        wc->composite_class.change_managed = super->composite_class.change_managed;
    }
    .
    .
    .
}
```

The inherit constants defined for the core widget are:

- `XtInheritRealize`
- `XtInheritResize`
- `XtInheritExpose`
- `XtInheritSetValuesAlmost`
- `XtInheritAcceptFocus`
- `XtInheritDisplayAccelerator`

The inherit constants defined for the composite widget are:

- `XtInheritGeometryManager`
- `XtInheritChangeManaged`
- `XtInheritInsertChild`
- `XtInheritDeleteChild`

D.3.8 Invocation of Superclass Operations

A widget class sometimes explicitly needs to call a superclass operation that usually is not chained. For example, a widget's `expose` procedure might call its superclass's `expose` and then perform more work of its own. Composite classes with fixed children can implement the `insert_child` procedure by first calling their superclass's `insert_child` procedure and then calling `XtManageChild` to add the child to the managed list.

Note that a method should call its own superclass method, not the widget's superclass method. That is, it should use its own class pointers only, not the widget's class pointers. This technique is referred to as enveloping the superclass's operation.

D.4 Creating Instances of Widgets to Build a User Interface

A collection of widget instances constitutes an application widget hierarchy. The shell widget returned by `APPLICATION CREATE SHELL` is the root of the application widget hierarchy. Widgets with one or more children are the intermediate nodes of the hierarchy; widgets with no children of any kind are the leaves of the hierarchy. The application widget hierarchy defines the associated X Window hierarchy.

Widgets can be either composite or primitive. Using composite widgets to build an application widget hierarchy is advisable for many reasons. While both types of widgets can contain children, the intrinsics provide a set of management mechanisms for building and communicating between composite widgets, their children, and other clients.

Composite widgets, subclasses of `Composite`, are containers for an arbitrary but implementation-defined collection of children, which may be created by the composite widget itself, by other clients, or by a combination of the two. Composite widgets also contain methods for managing the geometry (layout) of any child widget. Under unusual circumstances, a composite widget may have no children, but it usually has at least one. By contrast, primitive widgets that contain children typically create specific children of a known class themselves and do not expect external clients to do so. Primitive widgets also do not have general geometry management methods.

In addition, the intrinsics recursively perform many operations (for example, realization and destruction) on composite widgets and all of their children. Primitive widgets that have children must be prepared to perform the recursive operations themselves on behalf of their children.

The application widget hierarchy is manipulated by several intrinsics routines. For example, the `REALIZE WIDGET` intrinsic routine traverses the hierarchy downward and recursively realizes all pop-up widgets and children of composite widgets. `DESTROY WIDGET` traverses the hierarchy downward and destroys all pop-up widgets and children of composite widgets. The functions that fetch and modify resources traverse the hierarchy upward and determine the inheritance of resources from a widget's ancestors. The `MAKE GEOMETRY REQUEST` intrinsic routine traverses the hierarchy up one level and calls the geometry manager that is responsible for a widget child's geometry.

Building Your Own Widgets

D.4 Creating Instances of Widgets to Build a User Interface

To facilitate traversing up the application widget hierarchy, each widget has a pointer to its parent widget. The shell widget that the APPLICATION CREATE SHELL intrinsic routine returns, however, has a parent pointer of null.

To facilitate traversing down the application widget hierarchy, each composite widget has a pointer to an array of children widgets, which includes all normal children created, not just the subset of children that are managed by the composite widget's geometry manager. Primitive widgets that create children are entirely responsible for all operations that require downward traversal below themselves. In addition, every widget has a pointer to an array of pop-up children widgets.

D.4.1 Widget Instance Initialization

The initialize procedure for a widget class is of type XtInitProc, as follows:

```
typedef void (*XtInitProc) (Widget, Widget);  
Widget request, new;
```

Argument	Function
request	Specifies the widget with resource values as requested by the argument list, the resource database, and the widget defaults
new	Specifies a widget with the new values, both resource and nonresource, that are actually allowed

An initialization procedure does the following:

- Allocates space for and copies any resources that are referenced by address. For example, if a widget has a field that is a string, it cannot depend on the characters at that address remaining constant but must dynamically allocate space for the string and copy it to the new space. (Do not allocate space for or copy callback lists.)
- Computes values for unspecified resource fields. For example, if width and height are 0, the widget should compute an appropriate width and height based on other resources. This is the only time that a widget should ever directly assign its own width and height.
- Computes values for uninitialized nonresource fields that are derived from resource fields. For example, graphics contexts (GCs) that the widget uses are derived from resources like background, foreground, and font.

An initialization procedure can also check certain fields for internal consistency. For example, it makes no sense to specify a color map for a depth that does not support that color map.

Initialization procedures are called in superclass-to-subclass order. Most of the initialization code for a specific widget class deals with fields defined in that class and not with fields defined in its superclasses.

If a subclass does not need an initialization procedure because it does not need to perform any initialization operations, it can specify null for the initialize field in the class record.

Building Your Own Widgets

D.4 Creating Instances of Widgets to Build a User Interface

Sometimes a subclass may want to overwrite values filled in by its superclass. In particular, size calculations of a superclass are often incorrect for a subclass and, in this case, the subclass must modify or recalculate fields declared and computed by its superclass.

As an example, a widget subclass can require more display area than its superclass. In this case, the width and height calculated by the superclass initialize procedure are too small and need to be incremented. The subclass widget needs to know if its superclass's size was calculated by the superclass or was specified explicitly. All widgets must place themselves into whatever size is explicitly given, but they should compute a reasonable size if no size is requested.

The **request** and **new** arguments provide the necessary information for how a subclass knows the difference between a specified size and a size computed by a superclass. The request widget is the widget as originally requested. The new widget starts with the values in the request, but it has been updated by all superclass initialization procedures called so far. A subclass initialize procedure can compare these two to resolve any potential conflicts.

In the previous example, the widget subclass that is larger than its superclass can see if the width and height in the request widget are 0. If so, the subclass widget adds its size to the width and height fields in the new widget. If not, it must make do with the size originally specified.

The new widget will become the actual widget instance record. Therefore, the initialization procedure should do all its work on the new widget (the request widget should never be modified). If the initialization procedure needs to call any routines that operate on a widget, it should specify new as the widget instance.

D.4.2 Constraint Widget Instance Initialization

The constraint widget initialization procedure is of type `XtInitProc`. The values passed to the parent constraint initialization procedure are the same as those passed to the child's class widget initialization procedure.

The constraint initialization procedure should compute any constraint fields derived from constraint resources. It can make further changes to the widget to make the widget conform to the specified constraints, such as changing the widget's size or position.

If a constraint class does not need a constraint initialization procedure, it can specify null for the initialize field of the `ConstraintClassPart` in the class record.

Building Your Own Widgets

D.4 Creating Instances of Widgets to Build a User Interface

D.4.3 Nonwidget Data Initialization

The initialize hook procedure is of type `XtArgsProc`, as follows:

```
typedef void (*XtArgsProc) (Widget, ArgList, Cardinal *);  
Widget w;  
ArgList args;  
Cardinal *num_args;
```

Argument	Function
w	Specifies the widget
args	Specifies the argument list to override the resource defaults
num_args	Specifies the number of arguments in the argument list

If this procedure is not null, it is called immediately after the corresponding initialize procedure or in its place if the initialize procedure is null.

The initialize hook procedure allows a widget instance to initialize nonwidget data using information from the specified argument list. For example, the text widget has subparts that are not widgets, yet these subparts have resources that can be specified by means of the resource file or an argument list.

D.4.4 Widget Instance Window Creation

The realize procedure for a widget class is of type `XtRealizeProc`, as follows:

```
typedef void (*XtRealizeProc) (Widget, XtValueMask *, XSetWindowAttributes *);  
Widget w;  
XtValueMask *value_mask;  
XSetWindowAttributes *attributes;
```

Argument	Function
w	Specifies the widget
value_mask	Specifies which fields in the attributes structure to use
attributes	Specifies the window attributes to use in the <code>XCreateWindow</code> call

The realize procedure must create the widget's window.

The `REALIZE WIDGET` intrinsic routine fills in a mask and a corresponding `XSetWindowAttributes` structure. It sets the following fields based on information in the widget Core structure:

- The `background_pixmap` field (or `background_pixel` if `background_pixmap` is null) is filled in from the corresponding field.
- The `border_pixmap` field (or `border_pixel` if `border_pixmap` is null) is filled in from the corresponding field.
- The `event_mask` field is filled in based on the event handlers registered, the event translations specified, whether `expose` is not null, and whether `visible_interest` is true.

D.4 Creating Instances of Widgets to Build a User Interface

- The `bit_gravity` field is set to `NorthWestGravity` if the `expose` field is null.
- The `do_not_propagate_mask` field is set to propagate all pointer and keyboard events up the window tree. A composite widget can implement functionality caused by an event anywhere inside it (including on top of children widgets) as long as children do not specify a translation for the event.

All other fields in the `XSetWindowAttributes` structure (and the corresponding bits in the `value_mask` argument of the `CREATE WINDOW` Xlib routine) can be set by the `realize` procedure.

Note that, because `realize` is not a chained operation, the widget class `realize` procedure must update the `XSetWindowAttributes` structure with all the appropriate fields from non-Core superclasses.

A widget class can inherit its `realize` procedure from its superclass during class initialization. The `realize` procedure defined for the core widget calls the `CREATE WINDOW` intrinsic routine with the passed `value_mask` and `attributes` arguments, and with `window_class` and `visual` arguments set to `CopyFromParent`. Both `CompositeWidgetClass` and `ConstraintWidgetClass` inherit this `realize` procedure, and most new widget subclasses can do the same.

The most common noninherited `realize` procedures set the `bit_gravity` field in the mask and in the `XSetWindowAttributes` structure to the appropriate value and then create the window. For example, depending on its justification, the label widget sets the `bit_gravity` field to `WestGravity`, `CenterGravity`, or `EastGravity`. Consequently, shrinking the label widget just moves the bits appropriately, and no `Expose` event is needed for repainting.

If a composite widget's children should be realized in a particular order (typically to control the stacking order), the composite widget should call the `REALIZE WIDGET` intrinsic routine on its children in the appropriate order from within its own `realize` procedure.

Widgets that have children and that are not a subclass of `compositeWidgetClass` are responsible for calling the `REALIZE WIDGET` intrinsic routine on their children, usually from within the `realize` procedure.

D.4.5 Dynamic Data Deallocation

The `destroy` procedure is retrieved from the `destroy` field of the `CoreClassPart` structure and is of type `XtWidgetProc`, as follows:

```
typedef void (*XtWidgetProc) (Widget);  
Widget w;
```

Argument	Function
w	Specifies the widget

Building Your Own Widgets

D.4 Creating Instances of Widgets to Build a User Interface

The destroy procedures are called in subclass-to-superclass order. Therefore, a widget's destroy procedure should only deallocate storage that is specific to the subclass and should not deallocate the storage allocated by any of its superclasses. If a widget does not need to deallocate any storage, the destroy procedure entry in its widget class record can be null.

Deallocating storage includes but is not limited to:

- Calling `XtFree` on dynamic storage allocated with `XtMalloc`, `XtCalloc`, and so on
- Calling `XFreePixmap` on pixmaps created with direct X calls
- Calling `XtDestroyGC` on GCs allocated with `XtGetGC`
- Calling `XFreeGC` on GCs allocated with direct X calls
- Calling `XtRemoveEventHandler` on event handlers added with `XtAddEventHandler`
- Calling `XtRemoveTimeOut` on timers created with `XtAppAddTimeOut`
- Calling `XtDestroyWidget` for each child if the widget has children and is not a subclass of `compositeWidgetClass`

D.4.6 Dynamic Constraint Data Deallocation

The constraint destroy procedure is retrieved from the `destroy` field of the `ConstraintClassPart` structure, is called for a widget whose parent is a subclass of `constraintWidgetClass`, and is of type `XtWidgetProc`. The constraint destroy procedures are called in subclass-to-superclass order, starting at the widget's parent and ending at `constraintWidgetClass`. Therefore, a parent's constraint destroy procedure only should deallocate storage that is specific to the constraint subclass and not the storage allocated by any of its superclasses.

If a parent does not need to deallocate any constraint storage, the constraint destroy procedure entry in its class record can be null.

D.5 Composite Widgets and Their Children

Composite widgets (widgets that are a subclass of `compositeWidgetClass`) can have any number of children. Consequently, they are responsible for much more than primitive widgets. Their responsibilities (either implemented directly by the widget class or indirectly by intrinsic routines) include:

- Overall management of children from creation to destruction
- Destruction of descendants when the composite widget is destroyed
- Physical arrangement (geometry management) of a displayable subset of children (that is, the managed children)
- Mapping and unmapping of a subset of the managed children

Building Your Own Widgets

D.5 Composite Widgets and Their Children

Overall management is handled by the `CREATE WIDGET` and `DESTROY WIDGET` intrinsic routines. `CREATE WIDGET` adds children to the parent by calling the parent's insert child procedure. `DESTROY WIDGET` removes children from the parent by calling the parent's delete child procedure and ensures that all children of a destroyed composite widget also get destroyed.

Only a subset of the total number of children is actually managed by the geometry manager and, hence, possibly visible. For example, a multibuffer composite editor widget might allocate one child widget for each file buffer, but it might display only a small number of the existing buffers. Windows that are in this displayable subset are called managed windows and enter into geometry manager calculations. The other children are not managed and, by definition, are not mapped.

Children are added to and removed from the managed set by using the `MANAGE CHILD`, `MANAGE CHILDREN`, `UNMANAGE CHILD`, and `UNMANAGE CHILDREN` intrinsic routines, which notify the parent to recalculate the physical layout of its children by calling the parent's change managed procedure. The `CREATE MANAGED WIDGET` convenience routine calls `CREATE WIDGET` and `MANAGE CHILD` on the result.

Most managed children are mapped, but some widgets can be in a state where they take up physical space but do not show anything. Managed widgets are not mapped automatically if their `map_when_managed` field is false. The default is true and is changed by using the `SET MAPPED WHEN MANAGED` intrinsic routine.

Each composite widget class has a geometry manager, which is responsible for figuring out where the managed children should appear within the composite widget's window. Geometry management techniques fall into four classes:

- **Managing fixed boxes**

Fixed boxes have a fixed number of children that are created by the parent. All of these children are managed, and none ever makes a geometry manager request.

- **Managing homogeneous boxes**

Homogeneous boxes treat all children equally and apply the same geometry constraints to each child. Many clients insert and delete widgets freely.

- **Managing heterogeneous boxes**

Heterogeneous boxes have a specific location where each child is placed. This location usually is not specified in pixels because the window may be resized. Rather, it is expressed in terms of the relationship between a child and the parent or between the child and other specific children. Heterogeneous boxes are usually subclasses of the constraint widget.

Building Your Own Widgets

D.5 Composite Widgets and Their Children

- Managing shell boxes

Shell boxes have only one child, which is exactly the size of the shell widget. The geometry manager must communicate with the window manager if it exists. The box must also accept `ConfigureNotify` events when the window size is changed by the window manager.

D.5.1 Addition of Children to a Composite Widget

To add a child to the parent's list of children, the `CREATE_WIDGET` intrinsic routine calls the parent's class `insert child` procedure. The `insert child` procedure for a composite widget is of type `XtWidgetProc`, as follows:

```
typedef void (*XtWidgetProc) (Widget);  
Widget w;
```

Argument	Function
w	Specifies the child that is being added to the parent's list of children

Most composite widgets inherit their superclass's operation. The composite widget's `insert child` procedure calls the `insert position` procedure and inserts the child at the specified position.

Some composite widgets define their own `insert child` procedure so that they can order their children in some convenient way, create companion controller widgets for a new widget, or limit the number or type of their children widgets.

If there is not enough room to insert a new child in the children array (that is, the value of the `num_children` field equals the value of the `num_slots` field), the `insert child` procedure must first reallocate the array and update the `num_slots` field. The `insert child` procedure then places the child wherever it wants and increments the `num_children` field.

D.5.2 Insertion Order of Children

Instances of composite widgets need to specify the order in which their children are placed. For example, an application may want a set of command button widgets in some logical order grouped by function, and it may want the command button widgets that represent file names to be kept in alphabetical order.

The `insert position` procedure for a composite widget instance is of type `XtOrderProc`, as follows:

```
typedef Cardinal (*XtOrderProc) (Widget);  
Widget w;
```

Argument	Function
w	Specifies the widget

Building Your Own Widgets

D.5 Composite Widgets and Their Children

Composite widgets that allow clients to order their children (usually homogeneous boxes) can call their widget instance's insert position procedure from the class's insert child procedure to determine where a new child should go in its children array. Thus, a client of a composite class can apply different sorting criteria to widget instances of the class, passing in a different insert position procedure when it creates each composite widget instance.

The return value of the insert position procedure indicates how many children should go before the widget. Returning 0 means placement before all other children. Returning the value of the num_children field means placement after all other children. The default insert position procedure returns the value of the num_children field and can be overridden by a specific composite widget's resource list or by the argument list provided when the composite widget is created.

D.5.3 Deleting Children

To remove the child from the parent's children array, the DESTROY WIDGET routine eventually causes a call to the composite parent's class delete child procedure. A delete child procedure is of type XtWidgetProc, as follows:

```
typedef void (*XtWidgetProc) (Widget);  
Widget w;
```

Argument	Function
w	Specifies the widget

Most widgets inherit the delete child procedure from their superclass. Composite widgets that create companion widgets define their own delete child procedure to remove these companion widgets.

D.5.4 Constrained Composite Widgets

Constrained composite widgets are a subclass of compositeWidgetClass. Their name is derived from the fact that they manage the geometry of their children based on constraints associated with each child. These constraints can be as simple as the maximum width and height the parent will allow the child to occupy, or can be as complicated as how other children should change if this child is moved or resized. Constraint widgets let a parent define resources that are supplied for their children. For example, if the Constraint parent defines the maximum sizes for its children, these new size resources are retrieved for each child as if they were resources that were defined by the child widget. Accordingly, constraint resources may be included in the argument list or resource file just like any other resource for the child.

Constraint widgets have all the responsibilities of normal composite widgets and, in addition, must process and act upon the constraint information associated with each of their children.

Building Your Own Widgets

D.5 Composite Widgets and Their Children

To make it easy for widgets and the intrinsics to keep track of the constraints associated with a child, every widget has a constraints field, which is the address of a parent-specific structure that contains constraint information about the child. If a child's parent is not a subclass of `constraintWidgetClass`, the child's constraints field is null.

Subclasses of a constraint widget can add additional constraint fields to their superclass. To allow this, widget writers should define the constraint records in their private .h file by using the same conventions used for widget records. For example, a widget that needs to maintain a maximum width and height for each child might define its constraint record as follows:

```
typedef struct {
    Dimension max_width, max_height;
} MaxConstraintPart;

typedef struct {
    MaxConstraintPart max;
} MaxConstraintRecord, *MaxConstraint;
```

A subclass of this widget that also needs to maintain a minimum size would define its constraint record as follows:

```
typedef struct {
    Dimension min_width, min_height;
} MinConstraintPart;

typedef struct {
    MaxConstraintPart max;
    MinConstraintPart min;
} MaxMinConstraintRecord, *MaxMinConstraint;
```

Constraints are allocated, initialized, deallocated, and otherwise maintained as much as possible by the intrinsics. The constraint class record part has several entries that facilitate this. All entries in `ConstraintClassPart` are information and procedures that are defined and implemented by the parent, but they are called whenever actions are performed on the parent's children.

The `CREATE WIDGET` intrinsic routine uses the `constraint_size` field to allocate a constraint record when a child is created. The `constraint_size` field gives the number of bytes occupied by a constraint record. `CREATE WIDGET` also uses the constraint resources to fill in resource fields in the constraint record associated with a child. It then calls the constraint initialize procedure so that the parent can compute constraint fields that are derived from constraint resources and can possibly move or resize the child to conform to the given constraints.

The `GET VALUES` and `SET VALUES` intrinsic routines use the constraint resources to get the values or set the values of constraints associated with a child. The `SET VALUES` intrinsic routine then calls the constraint set values procedures so that a parent can recompute derived constraint fields and move or resize the child as appropriate.

The `DESTROY WIDGET` intrinsic routine calls the constraint destroy procedure to deallocate any dynamic storage associated with a constraint record. The constraint record itself must not be deallocated by the constraint destroy procedure; `DESTROY WIDGET` does this automatically.

D.6 Geometry Management

A widget does not directly control its size and location; rather, its parent is responsible for controlling its size and location. Although the position of children is usually left up to the parent, widgets themselves often have the best idea of their optimal sizes and, possibly, preferred locations.

To resolve physical layout conflicts between sibling widgets and between a widget and its parent, the intrinsics provide the geometry management mechanism. Almost all composite widgets have a geometry manager (`geometry_manager` field in the widget class record) that is responsible for the size, position, and stacking order of the widget's children. The only exception are fixed box widgets, which create their children themselves and can ensure that their children will never make a geometry request.

D.6.1 Initiating Geometry Changes

Parents, children, and clients all cause geometry changes differently. Because a parent has absolute control of its children's geometry, it changes the geometry directly by calling the `MOVE WIDGET`, `RESIZE WIDGET`, or `CONFIGURE WIDGET` intrinsic routines. A child must ask its parent for a geometry change by calling the `MAKE GEOMETRY REQUEST` or `MAKE RESIZE REQUEST` intrinsic routines. An application or other client code initiates a geometry change by calling the `SET VALUES` intrinsic routine on the appropriate geometry fields, thereby giving the widget the opportunity to modify or reject the client request before it gets propagated to the parent.

When a widget that needs to change its size, position, border width, or stacking depth asks its parent's geometry manager to make the desired changes, the geometry manager can do one of the following:

- Allow the request
- Disallow the request
- Suggest a compromise

When the geometry manager is asked to change the geometry of a child, the geometry manager may also rearrange and resize any or all of the other children that it controls. The geometry manager can move children around freely using `MOVE WIDGET`. When it resizes a child (that is, changes width, height, or border width) other than the one making the request, it should do so by calling `XtResizeWidget`. It can simultaneously move and resize a child with a single call to the `CONFIGURE WIDGET` intrinsic routine.

Often, geometry managers find that they can satisfy a request only if they can reconfigure a widget that they are not in control of (in particular, when the composite widget wants to change its own size). In this case, the geometry manager makes a request to its parent's geometry manager. Geometry requests can cascade this way to arbitrary depth.

Because such cascaded arbitration of widget geometry can involve extended negotiation, windows are not actually allocated to widgets at application startup until all widgets are satisfied with their geometry.

Building Your Own Widgets

D.6 Geometry Management

Users should be aware of the following:

- The intrinsic treatment of stacking requests is deficient in several areas. Stacking requests for unrealized widgets are granted but will have no effect. In addition, there is no way to use the SET VALUES intrinsic routine to generate a stacking geometry request.
- After a successful geometry request (one that returned XtGeometryYes), a widget does not know whether or not its resize procedure has been called. Therefore, widgets should have resize procedures that can be called more than once without negative effects.

D.6.2 General Geometry Manager Requests

To make a general geometry manager request from a widget, use the MAKE GEOMETRY REQUEST intrinsic routine.

The return codes from geometry managers are:

```
typedef enum _XtGeometryResult {
    XtGeometryYes,
    XtGeometryNo,
    XtGeometryAlmost,
    XtGeometryDone
} XtGeometryResult;
```

The XtWidgetGeometry structure, shown in the following example, is quite similar but not identical to the corresponding Xlib structure:

```
typedef unsigned long XtGeometryMask;

typedef struct {
    XtGeometryMask request_mode;
    Position x, y;
    Dimension width, height;
    Dimension border_width;
    Widget sibling;
    int stack_mode;
} XtWidgetGeometry;
```

The following request_mode definitions are from the X Window System symbol definition file (<X11/X.h>):

```
#define CWX (1<<0)
#define CWY (1<<1)
#define CWidth (1<<2)
#define CWHeight (1<<3)
#define CWBorderWidth (1<<4)
#define CWSibling (1<<5)
#define CWStackMode (1<<6)
#define CWQueryOnly (1<<7)
```

The additional mode XtCWQueryOnly indicates that the corresponding geometry request is only a query as to what would happen if this geometry request were made. No widgets should actually be changed.

The `MAKE GEOMETRY REQUEST` intrinsic routine, like the `XConfigureWindow Xlib` routine, uses bits in the `request_mode` field mask to determine which fields in the `XtWidgetGeometry` structure you want to specify.

The following `stack_mode` definitions are from the X Window System symbol definition file (`<X11/X.h>`):

```
#define      Above          0
#define      Below          1
#define      TopIf          2
#define      BottomIf       3
#define      Opposite       4
#define      XtSMDontChange 5
```

For definition and behavior of `Above`, `Below`, `TopIf`, `BottomIf`, and `Opposite`, see the *VMS DECwindows Xlib Programming Volume*. `XtSMDontChange` indicates that the widget wants its current stacking order preserved.

D.6.3 Resize Requests

To make a simple resize request from within a widget, you can use the `MAKE RESIZE REQUEST` intrinsic routine as an alternative to the `MAKE GEOMETRY REQUEST` intrinsic routine.

D.6.4 Potential Geometry Changes

Sometimes a geometry manager cannot respond to a geometry request from a child without first making a geometry request to the widget's own parent (the requestor's grandparent). If the request to the grandparent would allow the parent to satisfy the original request, the geometry manager can make the intermediate geometry request as if it were the originator. However, if the geometry manager already has determined that the original request cannot be completely satisfied (for example, if it always denies position changes), it needs to tell the grandparent to respond to the intermediate request without actually changing the geometry, because it does not know if the child will accept the compromise. To accomplish this, the geometry manager uses the `XtCWQueryOnly` mode in the `request_mode` field in the intermediate request.

When the `XtCWQueryOnly` mode is used, the geometry manager needs to cache enough information to reconstruct exactly the intermediate request. If the grandparent's response to the intermediate query was `XtGeometryAlmost`, the geometry manager needs to cache the entire reply geometry in the event the child accepts the parent's compromise.

If the grandparent's response was `XtGeometryAlmost`, it may also be necessary to cache the entire reply geometry from the grandparent when `XtCWQueryOnly` is not used. If the geometry manager is still able to satisfy the original request, it may immediately accept the grandparent's compromise and then act on the child's request. If the grandparent's compromise geometry is insufficient to allow the child's request and if the geometry manager is willing to offer a different compromise to the child,

Building Your Own Widgets

D.6 Geometry Management

the grandparent's compromise should not be accepted until the child has accepted the new compromise.

Note that a compromise geometry returned with `XtGeometryAlmost` is guaranteed only for the next call to the same widget; therefore, a cache of size one is sufficient.

D.6.5 Child Geometry Management

The geometry manager procedure for a composite widget class is of type `XtGeometryHandler`, as follows:

```
typedef XtGeometryResult (*XtGeometryHandler)(Widget, XtWidgetGeometry *,
                                             XtWidgetGeometry *);

Widget w;
XtWidgetGeometry *request;
XtWidgetGeometry *geometry_return;
```

A class can inherit its superclass's geometry manager during class initialization.

A bit set to 0 in the request's mask field means that the child widget does not care about the value of the corresponding field. The geometry manager can change it as it wishes. A bit set to 1 means that the child wants that geometry element changed to the value in the corresponding field.

If the geometry manager can satisfy all changes requested and if `XtCWQueryOnly` is not specified, it updates the widget's x, y, width, height, and border_width values appropriately. Then, it returns `XtGeometryYes`, and the value of the **geometry_return** argument is undefined. The widget's window is moved and resized automatically by the `MAKE GEOMETRY REQUEST` routine.

Homogeneous composite widgets often find it convenient to treat the widget making the request the same as any other widget, possibly reconfiguring it as part of its layout process, unless `XtCWQueryOnly` is specified. If it does this, it should return `XtGeometryDone` to inform `MAKE GEOMETRY REQUEST` that it does not need to do the configuration itself. Although `MAKE GEOMETRY REQUEST` resizes the widget's window, it does not call the widget class's resize procedure if the geometry manager returns `XtGeometryYes`. The requesting widget must perform whatever resizing calculations are needed explicitly.

If the geometry manager chooses to disallow the request, the widget cannot change its geometry. The value of the **geometry_return** argument is undefined, and the geometry manager returns `XtGeometryNo`.

Sometimes the geometry manager cannot satisfy the request exactly, but it may be able to satisfy a similar request. That is, it could satisfy only a subset of the requests (for example, size but not position) or a lesser request (for example, it cannot make the child as big as the request, but it can make the child bigger than its current size). In such cases, the geometry manager fills in the **geometry_return** argument with the actual changes it is willing to make, including an appropriate mask, and returns `XtGeometryAlmost`. If a bit in request_mode field of the **geometry_return** argument is 0, the geometry manager does not change the corresponding value if the **geometry_return** argument is

used immediately in a new request. If a bit is set to 1, the geometry manager does change that element to the corresponding value in the **geometry_return** argument. More bits may be set in the **geometry_return** argument than in the original request if the geometry manager intends to change other fields if the child accepts the compromise.

When **XtGeometryAlmost** is returned, the widget must decide if the compromise suggested in the **geometry_return** argument is acceptable. If it is, the widget must not change its geometry directly; rather, it must make another call to the **MAKE GEOMETRY REQUEST** intrinsic routine.

If the next geometry request from this child uses the **geometry_return** argument structure filled in by an **XtGeometryAlmost** return and if there have been no intervening geometry requests on either its parent or any of its other children, the geometry manager must grant the request, if possible. That is, if the child asks immediately with the returned geometry, it should get an answer of **XtGeometryYes**. However, the user's window manager may affect the final outcome.

To return an **XtGeometryYes** answer, the geometry manager frequently rearranges the position of other managed children by calling the **MOVE WIDGET** intrinsic routine. However, a few geometry managers may sometimes change the size of other managed children by calling the **RESIZE WIDGET** or **CONFIGURE WIDGET** intrinsic routines. If **XtCWQueryOnly** is specified, the geometry manager must return how it would react to this geometry request without actually moving or resizing any widgets.

Geometry managers must not assume that the **request** and **geometry_return** arguments point to independent storage. The caller is permitted to use the same field for both, and the geometry manager must allocate its own temporary storage if necessary.

D.6.6 Widget Placement and Sizing

To move a sibling widget of the child making the geometry request, use the **MOVE WIDGET** intrinsic routine.

To resize a sibling widget of the child making the geometry request, use the **RESIZE WIDGET** intrinsic routine.

To move and resize the sibling widget of the child making the geometry request, use the **CONFIGURE WIDGET** intrinsic routine.

To resize a child widget that already has the new values of its width, height, and border width fields, use the **RESIZE WINDOW** intrinsic routine.

Building Your Own Widgets

D.6 Geometry Management

D.6.7 Obtaining the Preferred Geometry

Some parents may be willing to adjust their layouts to accommodate the preferred geometries of their children. To obtain the preferred geometry and, as they see fit, use or ignore any portion of the response, these parents can use the QUERY GEOMETRY intrinsic routine. The syntax of the QUERY GEOMETRY routine is as follows:

```
XtGeometryResult XtQueryGeometry(w, intended, preferred_return)
Widget w;
XtWidgetGeometry *intended, *preferred_return;
```

Argument	Function
w	Specifies the widget.
intended	Specifies any changes the parent plans to make to the child's geometry or NULL.
preferred_return	Returns the child widget's preferred geometry.

To discover a child's preferred geometry, the child's parent sets any changes that it intends to make to the child's geometry in the corresponding fields of the **intended** structure, sets the corresponding bits in `intended.request_mode`, and calls the QUERY GEOMETRY intrinsic routine.

The QUERY GEOMETRY intrinsic routine clears all bits in the `preferred_return->request_mode` and checks the `query_geometry` field of the specified widget's class record. If the `query_geometry` field is not specified as null, the QUERY GEOMETRY intrinsic routine calls the `query_geometry` procedure and passes as arguments the specified widget, the **intended** structure, and the **preferred_return** structure. If the **intended** argument is specified as null, the QUERY GEOMETRY intrinsic routine replaces it with a pointer to an `XtWidgetGeometry` structure with the `request_mode` field set to 0 before calling the `query_geometry` procedure.

The query geometry procedure is of type `XtGeometryHandler`, as follows:

```
typedef XtGeometryResult (*XtGeometryHandler)(Widget, XtWidgetGeometry *,
                                             XtWidgetGeometry *);
Widget w;
XtWidgetGeometry *request;
XtWidgetGeometry *geometry_return;
```

The query geometry procedure is expected to examine the bits set in the `request_mode` field of the **intended** argument, evaluate the preferred geometry of the widget, and store the result in the **preferred_return** argument (setting the bits in the request mode field of the **preferred_return** argument corresponding to those geometry fields that it cares about). If the proposed geometry change is acceptable without modification, the query geometry procedure should return `XtGeometryYes`. If at least one field in the **preferred_return** argument is different from the corresponding field in the **intended** argument or if a bit was set in the **preferred_return** argument that was not set in the **intended** argument, the query geometry procedure should return `XtGeometryAlmost`. If the preferred geometry is identical to the current geometry, the query geometry procedure should return `XtGeometryNo`.

After calling the query geometry procedure or if the `query_geometry` field is null, the `QUERY GEOMETRY` intrinsic routine examines all the unset bits in `preferred_return->request_mode` and sets the corresponding fields in the **preferred_return** argument to the current values from the widget instance. If `CWStackMode` is not set, the `stack_mode` field is set to `XtSMDontChange`. The `QUERY GEOMETRY` intrinsic routine returns the value returned by the query geometry procedure or `XtGeometryYes` if the `query_geometry` field is null.

Therefore, the caller can interpret a return of `XtGeometryYes` as not needing to evaluate the contents of the reply and, more importantly, not needing to modify its layout plans. A return of `XtGeometryAlmost` means either that both the parent and the child expressed interest in at least one common field and the child's preference does not match the parent's intentions, or that the child expressed interest in a field that the parent might need to consider. A return value of `XtGeometryNo` means that both the parent and the child expressed interest in a field and that the child suggests that the field's current value is its preferred value. In addition, whether or not the caller ignores the return value or the reply mask, it is guaranteed that the reply structure contains complete geometry information for the child.

Parents are expected to call the `QUERY GEOMETRY` intrinsic routine in their layout routine and wherever other information is significant after the change managed procedure has been called. The changed managed procedure may assume that the child's current geometry is its preferred geometry. Thus, the child is still responsible for storing values into its own geometry during its initialize procedure.

D.6.8 Managing Size Changes

A child can be resized by its parent at any time. Widgets usually need to know when they have changed size so that they can lay out their displayed data again to match the new size. When a parent resizes a child, it calls the `RESIZE WIDGET` intrinsic routine, which updates the geometry fields in the widget, configures the window if the widget is realized, and calls the child's resize procedure to notify the child. The resize procedure is of type `XtWidgetProc`, as follows:

```
typedef void (*XtWidgetProc) (Widget);  
Widget w;
```

Argument	Function
w	Specifies the widget

If a class need not recalculate anything when a widget is resized, it can specify null for the `resize` field in its class record. This is an unusual case and should occur only for widgets with very trivial display semantics. The `resize` procedure takes a widget as its only argument. The `x`, `y`, `width`, `height` and `border_width` fields of the widget contain the new values. The `resize` procedure should recalculate the layout of internal data as needed. (For example, a centered label in a window that changes size should recalculate the starting position of the text.) The widget must obey `resize`

Building Your Own Widgets

D.6 Geometry Management

as a command and must not treat it as a request. A widget must not call the MAKE GEOMETRY REQUEST or MAKE RESIZE REQUEST intrinsic routines from its resize procedure.

D.7 Event Management

While X allows the reading and processing of events anywhere in an application, widgets in the X Toolkit neither directly read events nor grab the server or pointer. Widgets register procedures that are to be called when an event or class of events occurs in that widget.

A typical application consists of startup code followed by an event loop that reads events and dispatches them by calling the procedures that widgets have registered. The default event loop provided by the intrinsics is the APPLICATION MAIN LOOP intrinsic routine.

The event manager is a collection of functions to perform the following tasks:

- Add or remove event sources other than X server events (in particular, timer interrupts and file input).
- Query the status of event sources.
- Add or remove procedures to be called when an event occurs for a particular widget.
- Enable and disable the dispatching of user-initiated events (keyboard and pointer events) for a particular widget.
- Constrain the dispatching of events to a cascade of pop-up widgets.
- Call the appropriate set of procedures currently registered when an event is read.

Most widgets do not need to call any of the event handler functions explicitly. The normal interface to X events is through the higher-level translation manager, which maps sequences of X events (with modifiers) into procedure calls. Applications rarely use any of the event manager routines besides the APPLICATION MAIN LOOP intrinsic routine.

D.7.1 X Event Filters

The event manager provides filters that can be applied to X user events. The filters, which screen out events that are redundant or are temporarily unwanted, handle the following:

- Pointer motion compression
- Enter/leave compression
- Exposure compression

D.7.1.1 Pointer Motion Compression

Widgets can have a hard time keeping up with pointer motion events. Further, they usually do not actually care about every motion event. To disregard redundant motion events, the widget class field `compress_motion` should be true. When a request for an event would return a motion event, the intrinsics check if there are any other motion events immediately following the current one and, if so, skip all but the last of them.

D.7.1.2 Enter/Leave Compression

To disregard pairs of enter and leave events that have no intervening events, as can happen when the user moves the pointer across a widget without stopping in it, the widget class field `compress_enterleave` should be true. These enter and leave events are never delivered to the client.

D.7.1.3 Exposure Compression

Many widgets prefer to process a series of exposure events as a single expose region rather than as individual rectangles. Widgets with complex displays might use the expose region as a clip list in a graphics context; widgets with simple displays might ignore the region entirely and redisplay their whole window, or they might get the bounding box from the region and redisplay only that rectangle.

In either case, these widgets do not care about getting partial expose events. If the `compress_exposure` field in the widget class structure is true, the event manager calls the widget's expose procedure only once for each series of exposure events. In this case, all expose events are accumulated into a region. When the expose event with count zero is received, the event manager replaces the rectangle in the event with the bounding box for the region and calls the widget's expose procedure, passing the modified exposure event and the region.

If the `compress_exposure` field is false, the event manager calls the widget's expose procedure for every exposure event, passing it the event and a region argument of null.

D.7.2 Widget Exposure and Visibility

Every primitive widget and some composite widgets display data on the screen by means of Xlib calls. Widgets cannot simply write to the screen and forget what they have done. They must keep enough state information to redisplay the window or parts of it if a portion is obscured and then reexposed.

D.7.2.1 Redisplay of a Widget

The expose procedure for a widget class is of type `XtExposeProc`, as follows.

Building Your Own Widgets

D.7 Event Management

```
typedef void (*XtExposeProc) (Widget, XEvent *, Region);
Widget w;
XEvent *event;
Region region;
```

Argument	Function
w	Specifies the widget instance requiring redisplay
event	Specifies the exposure event giving the rectangle requiring redisplay
region	Specifies the union of all rectangles in this exposure sequence

The redisplay of a widget upon exposure is the responsibility of the expose procedure in the widget's class record. If a widget has no display semantics, it can specify null for the expose field. Many composite widgets serve only as containers for their children and have no expose procedure.

If the expose procedure is null, the `REALIZE_WIDGET` intrinsic routine fills in a default bit gravity of `NorthWestGravity` before it calls the widget's realize procedure.

If the widget's `compress_exposure` field is set to false, the region argument is always null. If the widget's `compress_exposure` field is set to true, the event contains the bounding box for region.

A small simple widget (for example, a label widget) can ignore the bounding box information in the event and redisplay the entire window. A more complicated widget (for example, a text widget) can use the bounding box information to minimize the amount of calculation and redisplaying of the widget it does. A very complex widget uses the region as a clip list in a graphics context and ignores the event information. The expose procedure is responsible for exposure of all superclass data as well as its own.

However, it is often possible to anticipate the display needs of several levels of subclassing. For example, rather than separate display procedures for the label, command, and toggle widgets, you could write a single display procedure in the label widget that uses the following display state fields:

```
Boolean invert
Boolean highlight
Dimension highlight_width
```

The label widget would have the `invert` and `highlight` fields always set to false and the `highlight_width` field set to 0. The command widget would dynamically set `highlight` and `highlight_width`, but it would leave `invert` always false. Finally, the toggle widget would dynamically set all three. In this case, the expose procedures for the command and toggle widgets inherit their superclass's expose procedure.

D.7.2.2 Widget Visibility

Some widgets may use substantial computing resources to display data. However, this effort is wasted if the widget is not actually visible on the screen, such as when the widget is obscured by another application or is made into an icon.

The `visible` field in the core widget structure provides a hint to the widget that it need not display data. This field is guaranteed true by the time an `Expose` event is processed if the widget is visible, but is usually false if the widget is not visible.

Widgets can use or ignore the visible hint. If they ignore it, they should have the `visible_interest` field in their widget class record set to false. In such cases, the `visible` field is initialized to true and never changes. If `visible_interest` is set to true, the event manager asks for `VisibilityNotify` events for the widget and updates the `visible` field accordingly.

D.7.3 X Event Handlers

Event handlers are procedures that are called when specified events occur in a widget. Most widgets need not use event handlers explicitly. Instead, they use the intrinsics translation manager. Event handlers are of the type `XtEventHandler`, as follows:

```
typedef void (*XtEventHandler)(Widget, caddr_t, XEvent *);
Widget w;
caddr_t client_data;
XEvent *event;
```

Argument	Function
<code>w</code>	Specifies the widget for which to handle events
<code>client_data</code>	Specifies the client-specific information registered with the event handler, which is usually null if the event handler is registered by the widget itself
<code>event</code>	Specifies the triggering event

To register an event handler procedure with the dispatch mechanism, use the `ADD EVENT HANDLER` intrinsic routine.

To remove a previously registered event handler, use the `REMOVE EVENT HANDLER` intrinsic routine.

To stop a procedure from receiving any events, which will remove it from the widget's event table entirely, call the `REMOVE EVENT HANDLER` intrinsic routine with the `event_mask` argument set to `XtAllEvents` and with the `nonmaskable` set to true.

On occasion, clients need to register an event handler procedure with the dispatch mechanism without causing the server to select for that event. To do this, use the `ADD RAW EVENT HANDLER` intrinsic routine.

To remove a previously registered raw event handler, use the `REMOVE RAW EVENT HANDLER` intrinsic routine.

Building Your Own Widgets

D.7 Event Management

To retrieve the event mask for a given widget, use the `BUILD EVENT MASK` intrinsic routine.

D.8 Resource Management

A resource is a field in the widget record with a corresponding resource entry in the resource list of the widget or any of its superclasses. This means that the field is settable by the `CREATE WIDGET` intrinsic routine (by naming the field in the argument list), by an entry in the default resource files (by using either the name or class), and by using the `SET VALUES` intrinsic routine. In addition, it is readable by the `GET VALUES` routine. Not all fields in a widget record are resources. Some are for bookkeeping use by the generic routines (like `managed` and `being_destroyed`). Others can be for local bookkeeping, and still others are derived from resources (many graphics contexts and pixmap).

Writers of widgets need to obtain a large set of resources at widget creation time. Some of the resources come from the argument list supplied in the call to the `CREATE WIDGET` intrinsic routine, some from the resource database, and some from the internal defaults specified for the widget. Resources are obtained first from the argument list, then from the resource database for all resources not specified in the argument list, and finally from the internal default, if needed.

D.8.1 Resource Lists

A resource entry specifies a field in the widget, the text name and class of the field that argument lists and external resource files use to refer to the field, and a default value that the field should get if no value is specified. The declaration for the `XtResource` structure is as follows:

```
typedef struct {
    String resource_name;
    String resource_class;
    String resource_type;
    Cardinal resource_size;
    Cardinal resource_offset;
    String default_type;
    caddr_t default_address;
} XtResource, *XtResourceList;
```

The `resource_name` field contains the name used by clients to access the field in the widget. By convention, it starts with a lowercase letter and is spelled identically to the field name, except all underscores (`_`) are deleted, and the next letter is replaced by its uppercase counterpart. For example, the resource name for `background_pixel` becomes `backgroundPixel`. Widget header files typically contain a symbolic name for each resource name. All resource names, classes, and types used by the intrinsics are named in `<X11/StringDefs.h>`. The intrinsics symbolic resource names begin with `XtN` and are followed by the string name (for example, `XtNbackgroundPixel` for `backgroundPixel`).

A resource class provides two functions:

- It isolates an application from different representations that widgets can use for a similar resource.

Building Your Own Widgets

D.8 Resource Management

- It lets you specify values for several actual resources with a single name. A resource class should be chosen to span a group of closely related fields.

For example, a widget can have several pixel resources: background, foreground, border, block cursor, pointer cursor, and so on. Typically, the background defaults to white and everything else to black. The resource class for each of these resources in the resource list should be chosen so that it takes the minimal number of entries in the resource database to make background off-white and everything else dark blue.

In this case, the background pixel should have a resource class of Background and all the other pixel entries a resource class of Foreground. Then the resource file needs only two lines to change all pixels to off-white or dark blue:

```
*Background:  offwhite
*Foreground:  darkblue
```

Similarly, a widget may have several resource fonts (such as normal and bold), but all fonts should have the class Font. Thus, changing all fonts requires only a single line in the default resource file:

```
*Font:        6x13
```

By convention, resource classes always start with an initial capital. Their symbolic names are preceded with XtC (for example, XtCBackground).

The resource_type field is the physical representation type of the resource. By convention, it starts with an uppercase letter and is spelled identically to the type name of the field. The resource type is used when resources are fetched to convert from the resource database format (usually a text string) or the default resource format (almost anything, but often a text string) to the desired physical representation. Table D-3 lists the resource types defined by the intrinsics.

Table D-3 Resource Types

Resource Type	Structure or Field Type
XtRAcceleratorTable	XtAccelerators
XtRBool	Bool
XtRBoolean	Boolean
XtRCallback	XtCallbackList
XtRColor	XColor
XtRCursor	Cursor
XtRDimension	Dimension
XtRDisplay	Pointer to a Display structure
XtRFile	Pointer to a FILE structure
XtRFont	Font

(continued on next page)

Building Your Own Widgets

D.8 Resource Management

Table D-3 (Cont.) Resource Types

Resource Type	Structure or Field Type
XtRFontStruct	Pointer to an XFontStruct structure
XtRFunction	Pointer to a procedure
XtRInt	int
XtRPixel	Pixel
XtRPixmap	Pixmap
XtRPointer	caddr_t
XtRPosition	Position
XtRShort	short
XtRString	Pointer to a string of characters
XtRTranslationTable	XtTranslations
XtRUnsignedChar	unsigned char
XtRWidget	Widget
XtRWindow	Window

The `resource_size` field is the size of the physical representation in bytes; you should specify it as `sizeof(type)` so that the compiler fills in the value. The `resource_offset` field is the offset in bytes of the field within the widget. Use the `XtOffset` macro to retrieve this value. The `default_type` field is the representation type of the default resource value. If `default_type` is different from `resource_type` and the `default_type` is needed, the resource manager invokes a conversion procedure from `default_type` to `resource_type`. Whenever possible, the default type should be identical to the resource type in order to minimize widget creation time. However, there are sometimes no values of the type that the program can easily specify. In this case, it should be a value that the converter is guaranteed to work for (for example, `XtDefaultForeground` for a pixel resource). The `default_address` field is the address of the default resource value. The default is used if a resource is not specified in the argument list or in the resource database, or if the conversion from the representation type stored in the resource database fails, which can happen for various reasons (for example, a misspelled entry in a resource file).

Two special representation types (`XtRImmediate` and `XtRCallProc`) are usable only as default resource types. `XtRImmediate` indicates that the value in the `default_address` field is the actual value of the resource rather than the address of the value. The value must be in the correct representation type for the resource. No conversion is possible since there is no source representation type. `XtRCallProc` indicates that the value in the `default_address` field is a procedure variable. This procedure is automatically invoked with the widget, `resource_offset`, and a pointer to the `XrmValue` in which to store the result, and is an `XtResourceDefaultProc`, as follows:

Building Your Own Widgets

D.8 Resource Management

```
typedef void (*XtResourceDefaultProc)(Widget, int, XrmValue *)
Widget w;
int offset;
XrmValue *value;
```

Argument	Function
w	Specifies the widget whose resource is to be obtained
offset	Specifies the offset of the field in the widget record
value	Specifies the resource value to fill in

The `XtResourceDefaultProc` procedure should fill in the address field (`addr`) of the value with a pointer to the default data in its correct type.

The `default_address` field in the resource structure is declared as `caddr_t`. On some machine architectures, this may be insufficient to hold procedure variables.

To get the resource list structure for a particular class, use the `GET RESOURCE LIST` intrinsic routine.

The following is an abbreviated version of the resource list in a label widget:

```
/* Resources specific to Label */
static XtResource resources[] = {
    {XtNforeground, XtCforeground, XtRPixel, sizeof(Pixel),
     XtOffset(LabelWidget, label.foreground), XtRString, XtDefaultForeground},
    {XtNfont, XtCfont, XtRFontStruct, sizeof(XFontStruct *),
     XtOffset(LabelWidget, label.font), XtRString, XtDefaultFont},
    {XtNlabel, XtClabel, XtRString, sizeof(String),
     XtOffset(LabelWidget, label.label), XtRString, NULL},
    .
    .
    .
}
```

The complete resource name for a field of a widget instance is the concatenation of the application name (from `argv[0]`) or the name command-line option, the instance names of all the widget's parents, the instance name of the widget itself, and the resource name of the specified field of the widget. Likewise, the full resource class of a field of a widget instance is the concatenation of the application class (from the `APPLICATION CREATE SHELL` intrinsic routine), the widget class names of all the widget's parents (not the superclasses), the widget class name of the widget itself, and the resource name of the specified field of the widget.

To determine the byte offset of a field within a structure, use the `OFFSET` intrinsic routine.

Building Your Own Widgets

D.8 Resource Management

D.8.2 Superclass to Subclass Chaining of Resource Lists

The `CREATE WIDGET` intrinsic routine gets resources as a superclass-to-subclass operation. That is, the resources specified in the core widget resource list are fetched, then those in the subclass, and so on down to the resources specified for this widget's class. Within a class, resources are fetched in the order they are declared.

In general, if a widget resource field is declared in a superclass, that field is included in the superclass's resource list and need not be included in the subclass's resource list. For example, the core widget class contains a resource entry for the background pixel, called `background_pixel`. Consequently, the implementation of a label widget need not also have a resource entry for the `background_pixel` field. However, a subclass, by specifying a resource entry for that field in its own resource list, can override the resource entry for any field declared in a superclass. This is most often done to override the defaults provided in the superclass with new ones. At class initialization time, resource lists for that class are scanned from the superclass down to the class to look for resources with the same offset. A matching resource in a subclass will be reordered to override the superclass entry. (A copy of the superclass resource list is made to avoid affecting other subclasses of the superclass.)

D.8.3 Retrieving Subresources

A widget does not do anything to get its own resources; instead, the `CREATE WIDGET` intrinsic routine does this automatically before calling the class initialize procedure.

Some widgets have subparts that are not widgets but for which the widget would like to fetch resources. For example, the text widget fetches resources for its source and destination. Such widgets call the `GET SUBRESOURCES` intrinsic routine to accomplish this.

D.8.4 Obtaining Application Resources

To retrieve resources that are not specific to a widget but that apply to the overall application, use the `GET APPLICATION RESOURCES` intrinsic routine.

D.8.5 Resource Conversions

The intrinsics provide a mechanism for registering representation converters that are automatically invoked by the resource fetching routines. The intrinsics additionally provide and register several commonly used converters. This resource conversion mechanism serves several purposes:

- It permits user and application resource files to contain ASCII representations of nontextual values.

Building Your Own Widgets

D.8 Resource Management

- It allows textual or other representations of default resource values that are dependent on the display, screen, or color map, and thus must be computed at run time.
- It caches all conversion source and result data. Conversions that require much computation or space (for example, string to translation table) or that require round trips to the server (for example, string to font or color) are performed only once.

D.8.5.1 Predefined Resource Converters

The intrinsics define all the representations used in the core, composite, constraint, and shell widgets. Furthermore, the intrinsics register resource converters that convert resources from their text string representation to all the following representations.

For XtRString, the intrinsics register the following representations:

- XtRAcceleratorTable
- XtRBool
- XtRBoolean
- XtRCursor
- XtRDimension
- XtRDisplay
- XtRFile
- XtRFloat
- XtRFont
- XtRFontStruct
- XtRInt
- XtRPixel
- XtRPosition
- XtRShort
- XtRTranslationTable
- XtRUnsignedChar

For XtRColor, the intrinsics register the following representations:

- XtRPixel

For XRInt, the intrinsics register the following representations:

- XtRBool
- XtRBoolean
- XtRColor
- XtRDimension
- XtRFloat

Building Your Own Widgets

D.8 Resource Management

- XtRFont
- XtRPixel
- XtRPixmap
- XtRPosition
- XtRShort
- XtRUnsignedChar

For XtRPixel, the intrinsics register the following representation:

- XtRColor

The string-to-pixel conversion has two predefined constants that are guaranteed to work and contrast with each other (XtDefaultForeground and XtDefaultBackground). They evaluate the black and white pixel values of the widget's screen, respectively. For applications that run with reverse video, however, they evaluate the white and black pixel values of the widget's screen, respectively. Similarly, the string-to-font and font structure converters recognize the constant XtDefaultFont and evaluate this to the font in the screen's default graphics context.

D.8.5.2 New Resource Converters

Type converters use pointers to XrmValue structures for input and output values. The following is the definition of the XrmValue structure in the X Window System symbol definition file (<X11/Xresource.h>):

```
typedef struct {
    unsigned int size;
    caddr_t addr;
} XrmValue, *XrmValuePtr;
```

A resource converter is a procedure of type XtConverter, as follows:

```
typedef void (*XtConverter)(XrmValue *, Cardinal *, XrmValue *, XrmValue *);
XrmValue *args;
Cardinal *num_args;
XrmValue *from;
XrmValue *to;
```

Argument	Function
args	Specifies a list of additional XrmValue arguments to the converter if additional context is needed to perform the conversion or null. For example, if the string-to-font converter needs the widget's screen, or if the string-to-pixel converter needs the widget's screen and color map.
num_args	Specifies the number of additional XrmValue arguments or 0.
from	Specifies the value to convert.
to	Specifies the descriptor to use to return the converted value.

Type converters should perform the following actions:

- Check to see that the number of arguments passed is correct.
- Attempt the type conversion.

Building Your Own Widgets

D.8 Resource Management

- If successful, return a pointer to the data in the **to** argument; otherwise, call `XtWarningMsg` and return without modifying the **to** argument.

Most type converters just take the data described by the specified **from** argument and return data by writing into the specified **to** argument. A few need other information, which is available in the specified argument list. A type converter can invoke another type converter, which allows differing sources that may convert into a common intermediate result to make maximum use of the type converter cache.

Note that the address written in the address field (`addr`) of the **to** argument cannot be that of a local variable of the converter because this is not valid after the converter returns. It should be a pointer to a static variable, as in the following example where `screenColor` is returned.

The following is an example of a converter that takes a string and converts it to a pixel:

```
static void CvtStringToPixel(args, num_args, fromVal, toVal)
    XrmValuePtr    args;
    Cardinal       *num_args;
    XrmValuePtr    fromVal;
    XrmValuePtr    toVal;
{
    static XColor screenColor;
    XColor    exactColor;
    Screen    *screen;
    Colormap  colormap;
    Status    status;
    char      message[1000];
    XrmQuark  q;
    String    params[1];
    Cardinal  num_params = 1;

    if (*num_args != 2)
        XtErrorMsg("cvtStringToPixel", "wrongParameters", "XtToolkitError",
            "String to pixel conversion needs screen and colormap arguments",
            (String *)NULL, (Cardinal *)NULL);

    screen = *((Screen **) args[0].addr);
    colormap = *((Colormap *) args[1].addr);

    LowerCase((char *) fromVal->addr, message);
    q = XrmStringToQuark(message);

    if (q == XtQExtdefaultbackground)
    {
        done(&screen->white_pixel, Pixel);
        return;
    }
    if (q == XtQExtdefaultforeground)
    {
        done(&screen->black_pixel, Pixel);
        return;
    }
    if ((char) fromVal->addr[0] == '#') /* some color rgb definition */
    {
        status = XParseColor(DisplayOfScreen(screen), colormap,
            (String) fromVal->addr, &screenColor);
    }
}
```

Building Your Own Widgets

D.8 Resource Management

```
    if (status != 0)
    {
        status = XAllocColor(DisplayOfScreen(screen), colormap,
                               &screenColor);
    }
} else /* some color name */
    status = XAllocNamedColor(DisplayOfScreen(screen), colormap,
                               (String) fromVal->addr, &screenColor,
                               &exactColor);

if (status == 0) {
    params[0]=(String) fromVal->addr;
    XtWarningMsg("cvtStringToPixel", "noColormap", "XtToolkitError",
                "Cannot allocate colormap entry for \"%s\"", params, &num_params);
} else {
    done(&(screenColor.pixel), Pixel)
}
};
```

All type converters should define some set of conversion values that they are guaranteed to succeed on so these can be used in the resource defaults. This issue arises only with conversions, such as fonts and colors, where there is no string representation that all server implementations will necessarily recognize. For resources like these, the converter should define a symbolic constant (for example, `XtDefaultForeground`, `XtDefaultBackground`, or `XtDefaultFont`).

The `STRING CONVERSION WARNING` intrinsic routine issues a warning message with name "conversionError", type "string", class "XtToolkitError", and the default message string "Cannot convert src to type dst_type" for new resource converters that convert from strings.

To register a new converter, use the `APPLICATION ADD CONVERTER` intrinsic routine.

All resource-fetching routines (for example, `GET RESOURCES`, `GET APPLICATION RESOURCES`, and so on) call resource converters if the user specifies a resource that is a different representation from the desired representation or if the widget's default resource value representation is different from the desired representation.

To invoke resource conversions, use the `CONVERT` or `DIRECT CONVERT` intrinsic routines.

D.8.6 Reading and Writing Widget Resource Fields

Any resource field in a widget can be read or written by a client. On a write operation, the widget decides what changes it will actually allow and updates all derived fields appropriately.

To retrieve the current value of a resource associated with a widget instance, use the `GET VALUES` intrinsic routine.

D.8.6.1 Widget Subpart Resource Data

Widgets that have subparts can return resource values from them for the GET VALUES routine by supplying a get values hook procedure. The get values hook procedure is of type XtArgsProc, as follows:

```
typedef void (*XtArgsProc)(Widget, ArgList, Cardinal *);
Widget w;
ArgList args;
Cardinal *num_args;
```

Argument	Function
w	Specifies the widget whose nonwidget resource values are to be retrieved
args	Specifies the argument list that was passed to XtCreateWidget
num_args	Specifies the number of arguments in the argument list

The widget should call the GET SUBVALUES intrinsic routine and pass in its subresource list and the **arg** and **num_args** arguments.

To retrieve the current value of a nonwidget resource data associated with a widget instance, use the GET SUBVALUES routine. For a description of nonwidget subclass resources, see Section D.8.3.

D.8.7 Setting Widget Resource Fields

To modify the current value of a resource associated with a widget instance, use the SET VALUES intrinsic routine.

D.8.7.1 Specifying Widget State

The set values procedure for a widget class is of type XtSetValuesFunc, as follows:

```
typedef Boolean (*XtSetValuesFunc)(Widget, Widget, Widget);
Widget current;
Widget request;
Widget new;
```

Argument	Function
current	Specifies a copy of the widget as it was before the XtSetValues call
request	Specifies a copy of the widget with all values changed as asked for by the XtSetValues call before any class set values procedures have been called
new	Specifies the widget with the new values that are actually allowed

The set values procedure should recompute any field derived from resources that are changed (for example, many graphics contexts depend on foreground and background). If no recomputation is necessary and if none of the resources specific to a subclass require the window to be redisplayed when their values are changed, you can specify null for the set_values field in the class record.

Building Your Own Widgets

D.8 Resource Management

Like the initialize procedure, the set values procedure deals mostly with the fields defined in the subclass, but it has to resolve conflicts with its superclass, especially conflicts over width and height.

The new widget is the actual widget instance record. Therefore, the set values procedure should do all its work on the new widget (the request widget should never be modified), and if it needs to call any routines that operate on a widget, it should specify the **new** argument as the widget instance.

The widget specified in the **new** argument starts with the values of that specified by request but has been modified by any superclass set values procedures. A widget need not refer to the request widget, unless it must resolve conflicts between the current and new widgets. Any changes that the widget needs to make, including geometry changes, should be made in the new widget.

Finally, the set values procedure must return a Boolean value that indicates whether the widget needs to be redisplayed. Note that a change in the geometry fields alone does not require the set values procedure to return true; the X server will eventually generate an expose event, if necessary. After calling all the set values procedures, the SET VALUES intrinsic routine forces a redisplay by calling the Xlib CLEAR AREA routine if any of the set values procedures returned true. Therefore, a set values procedure should not try to do its own redisplaying.

Set values procedures should not do any work in response to changes in geometry because the SET VALUES intrinsic routine will eventually perform a geometry request, and that request might be denied. If the widget actually changes size in response to the SET VALUES intrinsic routine, its resize procedure is called. Widgets should do any geometry-related work in their resize procedures.

Note that it is permissible to call the SET VALUES intrinsic routine before a widget is realized. Therefore, the set values procedure must not assume that the widget is realized.

D.8.7.2 Specifying Widget Geometry Values

The set values almost procedure for a widget class is of type XtAlmostProc, as follows:

```
typedef void (*XtAlmostProc)(Widget, Widget, XtWidgetGeometry *, XtWidgetGeometry *);
Widget w;
Widget new_widget_return;
XtWidgetGeometry *request;
XtWidgetGeometry *reply;
```

Argument	Function
w	Specifies the widget on which the geometry change is requested
new_widget_return	Specifies the new widget into which the geometry changes are to be stored
request	Specifies the original geometry request that was sent to the geometry manager that returned XtGeometryAlmost

Argument	Function
reply	Specifies the compromise geometry that was returned by the geometry manager that returned XtGeometryAlmost

Most classes inherit this operation from their superclass by specifying XtInheritSetValuesAlmost in the class initialization. The core widget set values almost procedure accepts the compromise suggested.

The set values almost procedure is called when a client tries to set a widget's geometry by means of a call to the SET VALUES routine, and the geometry manager cannot satisfy the request but instead returns XtGeometryAlmost and a compromise geometry. The set values almost procedure takes the original geometry and the compromise geometry and determines whether the compromise is acceptable or whether a different compromise might work better. It returns its results in the **new_widget** argument, which is then sent back to the geometry manager for another try.

D.8.7.3 Specifying Widget Constraint Information

The constraint set values procedure is of type XtSetValuesFunc. The values passed to the parent's constraint set values procedure are the same as those passed to the child's class set values procedure. A class can specify null for the set_values field of the ConstraintPart if it need not compute anything.

The constraint set values procedure should recompute any constraint fields derived from constraint resource that are changed. Further, it should modify the widget fields as appropriate. For example, if a constraint for the maximum height of a widget is changed to a value smaller than the widget's current height, the constraint set values procedure should reset the height field in the widget.

D.8.7.4 Specifying the Widget Subpart Resources

To set the current value of a nonwidget resource associated with a widget instance, use the SET SUBVALUES intrinsic routine. For a discussion of nonwidget subclass resources, see Section D.8.3.

Widgets that have a subpart can set the resource values by using the SET VALUES routine and supplying a set values hook procedure. The set values hook procedure for a widget class is of type XtArgsFunc, as follows:

```
typedef Boolean (*XtArgsFunc) (Widget, Arglist, Cardinal *);
Widget w;
ArgList args;
Cardinal *num_args;
```

Argument	Function
w	Specifies the widget whose nonwidget resource values are to be changed
args	Specifies the argument list that was passed to XtCreateWidget
num_args	Specifies the number of arguments in the argument list

Building Your Own Widgets

D.9 Translation Management

D.9 Translation Management

Except under unusual circumstances, widgets do not specify the mapping of user events into widget behavior by using the event manager. Instead, they provide a default mapping of events that you can override.

The translation manager provides an interface to specify and manage the mapping of X Event sequences into widget-supplied functionality, such as calling procedure `Abc` when the `Y` key is pressed.

The translation manager uses two kinds of tables to perform translations:

- The action tables, which are in the widget class structure, specify the mapping of externally available procedure name strings to the corresponding procedure implemented by the widget class.
- A translation table, which is in the widget class structure, specifies the mapping of event sequence to procedure name strings.

You can override the translation table in the class structure for a specific widget instance by supplying a different translation table for the widget instance. The resource name is `XtNtranslations`.

D.9.1 Action Tables

All widget class records contain an action table. An action table is made up of action records, defined as follows:

```
typedef struct _XtActionsRec {
    char      *string;
    XtActionProc  proc;
} XtActionsRec;
```

The `action_name` field of the action record is the name that you use in translation tables to access the procedure. The `action_proc` field is a pointer to a procedure that implements the functionality.

An application can register its own action tables with the translation manager so that the translation tables it provides to widget instances can access application functionality. The action procedure pointer in the translation table is of type `XtActionProc`, as follows:

```
typedef void (*XtActionProc)(Widget, XEvent *, String *, Cardinal *);
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
```

Argument	Function
<code>w</code>	Specifies the widget that caused the action to be called.
<code>event</code>	Specifies the event that caused the action to be called. If the action is called after a sequence of events, the last event in the sequence is used.

Building Your Own Widgets

D.9 Translation Management

Argument	Function
params	Specifies a pointer to the list of strings that were specified in the translation table as arguments to the action.
num_params	Specifies the number of arguments specified in the translation table.

For example, the command widget has procedures to take the following actions:

- Set the command button to indicate it is activated
- Unset the button back to its normal mode
- Highlight the button borders
- Unhighlight the button borders
- Notify any callbacks that the button has been activated

The action table for the command widget class makes these functions available to translation tables written for the command widget or any subclass. The string entry is the name used in translation tables. The procedure entry (often spelled identically to the string) is the name of the procedure that implements that function. The following is the action table for the command widget:

```
XtActionsRec actionTable[] = {
    {"Set", Set},
    {"Unset", Unset},
    {"Highlight", Highlight},
    {"Unhighlight", Unhighlight},
    {"Notify", Notify},
};
```

To declare an action table and register it with the translation manager, use the ADD ACTIONS intrinsic routine.

D.9.2 Translating Action Names to Procedures

The translation manager uses a simple algorithm to convert the name of a procedure specified in a translation table into the actual procedure specified in an action table. When the widget is realized, the translation manager performs a search for the name in the following tables:

- The widget's class action table for the name
- The widget's superclass action table and up the superclass chain
- The action tables registered with the ADD ACTIONS routine (from the most recently added table to the oldest table)

As soon as it finds a name, the translation manager stops the search. If it cannot find a name, the translation manager generates an error message.

Building Your Own Widgets

D.9 Translation Management

D.9.3 Translation Tables

All widget instance records contain a translation table, which is a resource with no default value. A translation table specifies what action procedures are invoked for an event or a sequence of events. A translation table is a string containing a list of translations from an event sequence into one or more action procedure calls. The translations are separated by new line characters (ASCII LF).

As an example, the default behavior of the command widget is:

- Highlight on enter window
- Unhighlight on exit window
- Invert on left button down
- Call callbacks and reinvert on left button up

The following illustrates the command widget's default translation table:

```
static String defaultTranslations =
    "<EnterWindow>:Highlight()\n\
    <LeaveWindow>:Unhighlight()\n\
    <Btn1Down>:Set()\n\
    <Btn1Up>:Notify() Unset()";
```

The `tm_table` field of the `CoreClass` record should be filled in at static initialization time with the string containing the class's default translations. If a class wants to inherit its superclass's translations, it can store the special value `XtInheritTranslations` into the `tm_table` field. After the class initialization procedures have been called, the intrinsics compile this translation table into an efficient internal form. Then, at widget creation time, this default translation table is used for any widgets that have not had their core translations field set by the resource manager or the initialize procedures.

The resource conversion mechanism automatically compiles string translation tables that are resources. If a client uses translation tables that are not resources, it must compile them itself using the `PARSE_TRANSLATIONS_TABLE` intrinsic routine.

The intrinsics use the compiled form of the translation table to register the necessary events with the event manager. Widgets need do nothing other than specify the action and translation tables for events to be processed by the translation manager.

D.9.3.1 Event Sequences

An event sequence is a comma-separated list of X event descriptions that describes a specific sequence of X events to map to a set of program actions. Each X event description consists of three parts:

- The X event type
- A prefix consisting of the X modifier bits
- An event-specific suffix

Various abbreviations are supported to make translation tables easier to read.

D.9.3.2 Action Sequences

Action sequences specify what program or widget actions to take in response to incoming X events. An action sequence is a sequence of action procedure call specifications. Each action procedure call consists of the name of an action procedure and a parenthesized list of string parameters to pass to that procedure.

D.9.4 Translation Table Syntax

The following Extended Backus Naur Form (EBNF) notation describes the syntax of a translation table file. The description uses the following conventions:

```
[ a ]   Means either nothing or "a"
{ a }   Means zero or more occurrences of "a"
```

All terminals are enclosed in quotation marks. Informal descriptions are enclosed in angle brackets (<>).

```
translationTable = [ directive ] { production }
directive       = ( "#replace" | "#override" | "#augment" ) "\n"
production     = lhs ":" rhs "\n"
lhs             = ( event | keyseq ) { "," (event | keyseq) }
keyseq         = "" keychar {keychar} ""
keychar        = [ "^" | "$" | "\" ] <ISO Latin 1 character>
event          = [modifier_list] "<event_type>" [ "(" count["+"] ")" ] {detail}
modifier_list  = ( [ "!" | ":" ] {modifier} ) | "None"
modifier       = [ "~" ] modifier_name
count          = ("1" | "2" | "3" | "4" | ...)
modifier_name  = "@" <keysym> | <see ModifierNames table below>
event_type     = <see Event Types table below>
detail         = <event specific details>
rhs            = { name "(" [params] ")" }
name           = namechar { namechar }
namechar      = { "a"-"z" | "A"-"Z" | "0"-"9" | "$" | "_" }
params        = string { "," string }.
string        = quoted_string | unquoted_string
quoted_string  = "" {<Latin 1 character>} ""
unquoted_string = {<Latin 1 character except space, tab, ",", newline, ">}
```

It is often convenient to include new lines in a translation table to make it more readable. In C, indicate a new line as \n, for example:

```
"<Btn1Down>:DoSomething()\n\
<Btn2Down>:DoSomethingElse()"
```

D.9.4.1 Modifier Names in a Translation Table

The modifier field is used to specify normal X keyboard and button modifier mask bits. Modifiers are legal on event types KeyPress, KeyRelease, ButtonPress, ButtonRelease, MotionNotify, EnterNotify, LeaveNotify, and their abbreviations. An error is generated when a translation table that contains modifiers for any other events is parsed.

The following rules are applicable to modifier lists:

- If the modifier list has no entries and has not been specified as None, it means any modifier is acceptable.

Building Your Own Widgets

D.9 Translation Management

- If an exclamation point (!) is specified at the beginning of the modifier list, it means that the listed modifiers must be in the correct state and no other modifiers can be asserted.
- If any modifiers are specified and an exclamation point is not specified, it means that the listed modifiers must be in the correct state; any unlisted modifiers can be in any order.
- If a modifier is preceded by a tilde (~), it means that the modifier must not be asserted.
- If None is specified, it means no modifiers can be asserted.
- A colon (:) specified at the beginning of the modifier list directs the intrinsics to apply any standard modifiers in the event to map the event keycode into a key symbol. The default standard modifiers are Shift and Lock, with the interpretation as defined in the X Window System. The resulting key symbol must exactly match the specified key symbol, and the nonstandard modifiers in the event must match the modifier_list. For example, :<Key>a is distinct from :<Key>A, and :Shift<Key>A is distinct from :<Key>A.
- If a colon is not specified, no standard modifiers are applied. Then, for example, <Key>A and <Key>a are equivalent.

In key sequences, a circumflex (^) is an abbreviation for the Control modifier, a dollar sign (\$) is an abbreviation for Meta, and a backslash (\) can be used to quote any character, in particular a quotation mark ("). A circumflex, a dollar sign, and another backslash. The following table shows how a translation table interprets the presence or absence of modifiers:

Modifier Specification	Interpretation
No modifiers	None <event> detail
Any modifiers	<event> detail
Only these modifiers	! mod1 mod2 <event> detail
These modifiers and any others	mod1 mod2 <event> detail

The use of None for a modifier list is identical to the use of an exclamation point with no modifiers.

Table D-4 lists the modifiers with their abbreviations and meanings.

Table D-4 Translation Table Modifiers

Modifier	Abbreviation	Meaning
Ctrl	c	Control modifier bit
Shift	s	Shift modifier bit

(continued on next page)

Table D-4 (Cont.) Translation Table Modifiers

Modifier	Abbreviation	Meaning
Lock	l	Lock modifier bit
Meta	m	Meta key modifier
Hyper	h	Hyper key modifier
Super	su	Super key modifier
Alt	a	Alt key modifier
Mod1		Mod1 modifier bit
Mod2		Mod2 modifier bit
Mod3		Mod3 modifier bit
Mod4		Mod4 modifier bit
Mod5		Mod5 modifier bit
Button1		Button1 modifier bit
Button2		Button2 modifier bit
Button3		Button3 modifier bit
Button4		Button4 modifier bit
Button5		Button5 modifier bit
ANY		Any combination

A key modifier is any modifier bit whose corresponding key code contains the corresponding left or right key symbol. For example, m or Meta means any modifier bit mapping to a key code whose key symbol list contains XK_Meta_L or XK_Meta_R. Note that this interpretation is for each display, not global or even for each application context. The Control, Shift, and Lock modifier names refer explicitly to the corresponding modifier bits; there is no additional interpretation of key symbols for these modifiers.

Because it is possible to associate arbitrary key symbols with modifiers, the set of modifier key modifiers is extensible. The @<keysym> syntax means any modifier bit whose corresponding key code contains the specified key symbol.

A modifier_list/key symbol combination in a translation matches a modifiers/key code combination in an event in the following:

- If a colon (:) is used, the intrinsics call the display's XtKeyProc with the key code and modifiers. To match, (modifiers & ~modifiers_return) must equal modifier_list, and keysym_return must equal the given key symbol.
- If a colon is not used, the intrinsics mask off all unspecified modifier bits from the modifiers. This value must be equal to the modifier_list field. Then, for each possible combination of the unspecified modifiers in the modifier_list field, the intrinsics call the display's XtKeyProc with the key code and that combination in a bitwise OR with the specified modifier bits from the event. The keysym_return field must match the key symbol in the translation.

Building Your Own Widgets

D.9 Translation Management

D.9.4.2 Event Types

The EventType field describes XEvent types. Table D-5 lists the currently defined EventType values.

Table D-5 Event Types

Type	Meaning
Key	KeyPress
KeyDown	KeyPress
KeyUp	KeyRelease
BtnDown	ButtonPress
BtnUp	ButtonRelease
Motion	MotionNotify
PtrMoved	MotionNotify
MouseMoved	MotionNotify
Enter	EnterNotify
EnterWindow	EnterNotify
Leave	LeaveNotify
LeaveWindow	LeaveNotify
FocusIn	FocusIn
FocusOut	FocusOut
Keymap	KeymapNotify
Expose	Expose
GrExp	GraphicsExpose
NoExp	NoExpose
Visible	VisibilityNotify
Create	CreateNotify
Destroy	DestroyNotify
Unmap	UnmapNotify
Map	MapNotify
MapReq	MapRequest
Reparent	ReparentNotify
Configure	ConfigureNotify
ConfigureReq	ConfigureRequest
Grav	GravityNotify
ResReq	ResizeRequest
Circ	CirculateNotify
CircReq	CirculateRequest
Prop	PropertyNotify
SelClr	SelectionClear

(continued on next page)

Table D-5 (Cont.) Event Types

Type	Meaning
SelReq	SelectionRequest
Select	SelectionNotify
Clrmap	ColormapNotify
Message	ClientMessage
Mapping	MappingNotify

Table D-6 lists the supported abbreviations.

Table D-6 Event Type Abbreviations for Translation Tables

Abbreviation	Meaning
Ctrl	KeyPress with control modifier
Meta	KeyPress with meta modifier
Shift	KeyPress with shift modifier
Btn1Down	ButtonPress with Btn1 detail
Btn1Up	ButtonRelease with Btn1 detail
Btn2Down	ButtonPress with Btn2 detail
Btn2Up	ButtonRelease with Btn2 detail
Btn3Down	ButtonPress with Btn3 detail
Btn3Up	ButtonRelease with Btn3 detail
Btn4Down	ButtonPress with Btn4 detail
Btn4Up	ButtonRelease with Btn4 detail
Btn5Down	ButtonPress with Btn5 detail
Btn5Up	ButtonRelease with Btn5 detail
BtnMotion	MotionNotify with any button modifier
Btn1Motion	MotionNotify with Button1 modifier
Btn2Motion	MotionNotify with Button2 modifier
Btn3Motion	MotionNotify with Button3 modifier
Btn4Motion	MotionNotify with Button4 modifier
Btn5Motion	MotionNotify with Button5 modifier

The detail field is event specific and normally corresponds to the detail field of an X Event, such as <Key>A. If no detail field is specified, then ANY is assumed.

A key symbol can be specified as any of the standard key symbol names, a hexadecimal number prefixed with 0x or 0X, an octal number prefixed with 0 or a decimal number. A key symbol expressed as a single digit is interpreted as the corresponding Latin1 key symbol; for example, 0 is the key symbol XK_0. Other single character key symbols are treated as literal constants from Latin1; for example, ! is treated as 0x21. Standard key symbol names are as defined in <X11/keysymdef.h> with the XK_ prefix removed.

Building Your Own Widgets

D.9 Translation Management

D.9.4.3 Canonical Representation

Every translation table has a unique, canonical text representation. This representation is passed to a widget's `display_accelerator` method to describe the accelerators installed on that widget. The canonical representation of a translation table file is the following:

```
translationTable = { production }
production      = lhs ":" rhs "\n"
lhs             = event { "," event }
event           = [modifier_list] "<event_type>" [ "(" count["+"] ")" ] {detail}
modifier_list  = ["!" | ":"] {modifier}
modifier        = ["~"] modifier_name
count          = ("1" | "2" | "3" | "4" | ...)
modifier_name  = "@<keysym> | <see canonical modifier names below>"
event_type     = <see canonical event types below>
detail         = <event specific details>
rhs            = { name "(" [params] ")" }
name           = namechar { namechar }
namechar       = { "a"-"z" | "A"-"Z" | "0"-"9" | "$" | "_" }
params         = string {"," string}.
string         = quoted_string
quoted_string  = "" {<Latin 1 character>} ""
```

The canonical modifier names are:

- Ctrl
- Shift
- Lock
- Mod1
- Mod2
- Mod3
- Mod4
- Mod5
- Button1
- Button2
- Button3
- Button4
- Button5

The canonical event types are:

- KeyPress
- KeyRelease
- ButtonPress
- ButtonRelease
- MotionNotify
- EnterNotify
- LeaveNotify

Building Your Own Widgets

D.9 Translation Management

- FocusIn
- FocusOut
- KeymapNotify
- Expose
- GraphicsExpose
- NoExpose
- VisibilityNotify
- CreateNotify
- DestroyNotify
- UnmapNotify
- MapNotify
- MapRequest
- ReparentNotify
- ConfigureNotify
- ConfigureRequest
- GravityNotify
- ResizeRequest
- CirculateNotify
- CirculateRequest
- PropertyNotify
- SelectionClear
- SelectionRequest
- SelectionNotify
- ColormapNotify
- ClientMessage

The following examples illustrate common translation table entries:

- Always put specific events in the table before general ones, as follows:

```
Shift <Btn1Down> : proc()\n<Btn1Down> : proc()
```

- For double click on Button 1 Up with Shift, use the following specification:

```
Shift<Btn1Up>(2) : proc()
```

This is equivalent to the following line with appropriate timers set between events:

```
Shift<Btn1Down>,Shift<Btn1Up>,  
Shift<Btn1Down>,Shift<Btn1Up>: proc()
```

Building Your Own Widgets

D.9 Translation Management

- For double click on Button 1 Down with Shift, use the following specification:

```
Shift<Btn1Down>(2) : proc()
```

This is equivalent to the following line with appropriate timers set between events:

```
Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down> : proc()
```

- Mouse motion is always discarded when it occurs between events in a table where no motion event is specified, as follows:

```
<Btn1Down>,<Btn1Up> : proc()
```

This translation is taken, even if the pointer moves a bit between the down and up events. Similarly, any motion event specified in a translation matches any number of motion events. If the motion event causes an action procedure to be invoked, the procedure is invoked after each motion event.

- If an event sequence is part of another event sequence (but not the first part), the event sequence is not taken if it occurs in the context of the longer sequence. This occurs mostly in sequences like the following:

```
<Btn1Down>,<Btn1Up> : proc()\n<Btn1Up> : proc()
```

The second translation is taken only if the button release is not preceded by a button press or if there are intervening events between the press and the release. Be particularly aware of this when using repeat notation with buttons and keys (because their expansion includes additional events) and when specifying motion events because they are implicitly included between any two other events. In particular, pointer motion and double click translations cannot coexist in the same translation table.

- For single click on Button 1 Up with Shift and Meta, use the following specification:

```
Shift Meta <Btn1Down>, Shift Meta<Btn1Up>: proc()
```

- You can use a plus sign (+) to indicate *for any number of clicks greater than or equal to count*. For example:

```
Shift <Btn1Up>(2+) : proc()
```

- To indicate EnterNotify with any modifiers, use the following specification:

```
<Enter> : proc()
```

- To indicate EnterNotify with no modifiers, use the following specification:

```
None <Enter> : proc()
```

- To indicate EnterNotify with Button 1 Down and Button 2 Up and other modifiers unspecified, use the following specification:

```
Button1 ~Button2 <Enter> : proc()
```


- To indicate EnterNotify with Button1 Down and Button2 Down exclusively, use the following specification:

```
! Button1 Button2 <Enter> : proc()
```

You do not need to use a tilde (~) with an exclamation point (!).

D.9.5 Translation Table Management

Sometimes an application needs to destructively or nondestructively add its own translations to a widget's translation. For example, a window manager provides functions to move a window. It can usually move the window when any pointer button is pressed down in a title bar, but it allows the user to specify other translations for MB2 or MB3 pressed down in the title bar and ignores any user translations for MB1 pressed down.

To accomplish this, the window manager first should create the title bar and then should merge the two translation tables into the title bar's translations. One translation table contains the translations that the window manager wants only if the user has not specified a translation for a particular event (or event sequence). The other translation table contains the translations that the window manager wants regardless of what the user has specified.

Three intrinsics routines support this merging:

- **PARSE TRANSLATION TABLE**—Compiles a translation table
- **AUGMENT TRANSLATIONS**—Nondestructively merges a compiled translation table into a widget's compiled translation table
- **OVERRIDE TRANSLATIONS**—Destructively merges a compiled translation table into a widget's compiled translation table

D.9.6 Using Accelerators

It is often convenient to be able to bind events in one widget to actions in another. In particular, it is often useful to be able to invoke menu actions from the keyboard. The intrinsics provide a facility, called accelerators, that let you accomplish this. An accelerator is a translation table that is bound with its actions in the context of a particular widget. The accelerator table can then be installed on some destination widget. When an action in the destination widget would cause an accelerator action to be taken, rather than causing an action in the context of the destination, the actions are executed as though triggered by an action in the accelerator widget.

Each widget instance contains that widget's exported accelerator table. Each class of widget exports a method that takes a displayable string representation of the accelerators so that widgets can display their current accelerators. The representation is the accelerator table in canonical translation table form.

Building Your Own Widgets

D.9 Translation Management

The display accelerator procedure pointer is of type `XtStringProc`, as follows:

```
typedef void (*XtStringProc)(Widget, String);
Widget w;
String string;
```

Argument	Function
w	Specifies the widget on which the accelerators are installed
string	Specifies the string representation of the accelerators for the widget

Accelerators can be specified in default files, and the string representation is the same as for a translation table. However, the interpretation of the augment and override directives apply to what will happen when the accelerator is installed, that is, whether or not the accelerator translations will override the translations in the destination widget. The default is augment, which means that the accelerator translations have lower priority than the destination translations. The replace directive is ignored for accelerator tables.

To parse an accelerator table, use the `PARSE ACCELERATOR TABLE` intrinsic routine.

To install accelerators from a widget on another widget, use the `INSTALL ACCELERATORS` intrinsic routine. As a convenience for installing all accelerators from a widget and all its descendants onto one destination, use the `INSTALL ALL ACCELERATORS` intrinsic routine.

D.9.7 Key Code to Key Symbol Conversions

The translation manager provides support for automatically translating key codes in incoming key events into key symbols. `KeyCode-to-KeySym`-translator procedure pointers are of type `XtKeyProc`, as follows:

```
typedef void (*XtKeyProc)(Display *, KeyCode, Modifiers, Modifiers *, KeySym *);
Display *display;
KeyCode keycode;
Modifiers modifiers;
Modifiers *modifiers_return;
KeySym *keysym_return;
```

Argument	Function
display	Specifies the display that the key code is from
keycode	Specifies the key code to translate
modifiers	Specifies the modifiers to the key code
modifiers_return	Returns a mask that indicates the subset of all modifiers that are examined by the key translator
keysym_return	Returns the resulting key symbol

This procedure takes a key code and modifiers and produces a key symbol. For any given key translator function, the `modifiers_return` argument

Building Your Own Widgets

D.9 Translation Management

will be a constant that indicates the subset of all modifiers that are examined by the key translator.

To register a key translator, use the `SET KEY TRANSLATOR` intrinsic routine.

To invoke the currently registered KeyCode-to-KeySym translator, use the `TRANSLATE KEYCODE` intrinsic routine.

To handle capitalization of nonstandard key symbols, the intrinsics allow clients to register case conversion routines. Case converter procedure pointers are of type `XtCaseProc`, as follows:

```
typedef void (*XtCaseProc) (KeySym *, KeySym *,
                             KeySym *);
    KeySym *keysym;
    KeySym *lower_return;
    KeySym *upper_return;
```

Argument	Function
<code>keysym</code>	Specifies the key symbol to convert
<code>lower_return</code>	Specifies the lowercase equivalent for the key symbol
<code>upper_return</code>	Specifies the uppercase equivalent for the key symbol

If there is no case distinction, this procedure should store the `KeySym` into both return values.

To register a case converter, use the `REGISTER CASE CONVERTER` intrinsic routine.

To determine uppercase and lowercase equivalents for a key symbol, use the `CONVERT CASE` intrinsic routine.

Glossary

accelerator: A shortcut that allows users to work more quickly by eliminating steps needed to invoke a command. Accelerators include key bindings or pointing-device-button bindings, pop-up menus, and double clicks.

active: Ready to accept user input, as in *active insertion point*, *active window*.

active grab: The condition that exists when a client has exclusive use of the pointer or keyboard.

active insertion point: An insertion point, indicated by a blinking cursor, that is ready to accept user input.

active window: The window that is ready to accept input from the keyboard (that is, that currently has *input focus*).

ancestor: In a hierarchy of windows, a window is an ancestor if it is logically closer to the root window than to another window.

application title: The part of the window that identifies the window and the primary file, if any, associated with an application. The application title is located to the right of the shrink-to-icon button.

arguments list: In UIL, the sequence of argument name/argument value pairs that describes the initial state of a widget. The UIL compiler and DRM translate the pairs into a data structure that can be passed as an argument to a low-level widget creation routine.

atom: A unique identifier for a string.

attribute: A characteristic of a window other than those related to its size and shape.

background: The default contents of a window.

bit gravity: The location of the contents of a window after it has been resized and repainted.

bitmap: A two-dimensional array of bits that represent the pixels of an image in which the 1 bits represent foreground pixel values and the 0 bits represent background pixel values.

In UIL, a bitmap can have one of two values: FOREGROUND or BACKGROUND. See also *pixmap*.

bitmap text insertion pointer: The pointer that specifies the point where text can be entered in a bitmapped image. This pointer includes a horizontal baseline to help align text.

Glossary

- bitmask:** A sequence of 8 (byte), 16 (word), or 32 (longword) unsigned bits that can be combined with a variable of the same size by a binary operator (AND, OR, XOR, and so on) to either set or check the value of individual bits in the variables.
- border:** The margin of a window, defined by either a pixel value or a pixmap.
- button:** An on-screen control that allows users to choose actions or operations and set states.
- byte order:** The order in which data is organized in a bitmap or pixmap.
- callback:** A mechanism for allowing the XUI Toolkit widget routines to notify an application that a particular event has occurred. Every widget has a callback list that connects the user's manipulation of that widget with the functions of the application. For example, when the user clicks on (activates) a widget, every procedure on the widget's `ACTIVATE` callback list is called. See also *callbacks list*.
- callback reason:** A constant, defined by the XUI Toolkit, that describes a state change in a widget. For example, when the user clicks on a toggle button, the state changes. The callback reason for this state change is called *value_changed*.
- callback routine:** An application-specified routine to be invoked by a widget when a widget changes state. See also *callbacks list*.
- callbacks list:** In UIL, the sequence of reason/procedure pairs that describes the state changes in a widget under which an application is to receive control. Each reason describes a state change, and the corresponding procedure is the procedure the application invokes. The UIL compiler and DRM translate the pairs into a data structure that can be passed as an argument to a low-level widget creation routine. Procedures are specified as case-sensitive names (strings), which must be registered in DRM for this translation to work correctly. See also *registration*.
- cancel:** To remove a modal dialog box without applying changes.
- case-sensitivity clause:** The part of the UIL module that specifies whether or not names in a UIL module are case sensitive.
- caution dialog box:** A standard dialog box that informs the user of the consequences of carrying out an action. When the box appears, application activity stops and user input is required for application activity to proceed.
- character set:** A code that identifies the character encoding and writing direction of a primitive string. For example, the UIL character set name `ISO_LATIN1` states that the hexadecimal value 5C is the backslash character (`\`), while the `DEC_KATAKANA` character set states that the value 5C is the yen symbol (`¥`).
- child:** A first-level subwindow of a window.
In an application widget hierarchy, a *child* is a widget that is controlled by another widget; a descendant of a widget.
- choose:** To pick an operation by clicking on a control or dragging to a menu item.

- class:** The type of a window. Possible types are input-only and input-output.
- click:** To press and release a mouse button.
- click on:** To press and release a mouse button when the pointer is positioned on an active object.
- click rate:** The speed with which one click follows another in a double click.
- client:** An application program connected to the server.
- clip:** To restrict drawing to a specified area of a window.
- clip region:** A region to which screen output is restricted.
- clipboard:** The storage area (buffer) for the most recently cut or copied information (text or graphics).
- close:** To remove a window associated with an application.
- color:** In UIL, the text string naming a color. The DRM converts this string into a usable internal form. The terms FOREGROUND and BACKGROUND are also acceptable color names in UIL and DRM for mapping colors to monochrome displays. See also *color table*.
- color cell:** An entry in a color map that defines one color or shade of gray.
- color dialog box:** A standard dialog box that displays colors and color attributes from which a user can choose.
- color map:** A resource that associates colors with pixel values.
- color table:** In UIL, a map associating defined colors and single characters that may be used to construct icons. FOREGROUND and BACKGROUND are defined colors in the UIL color tables, in addition to named colors. See also *color*.
- command box:** A subdivision of the main window within which users can enter commands and receive messages in response to those commands.
- command item:** A choice on a menu that initiates an action or operation directly, without calling a dialog box or submenu.
- compound string:** A string stored with character set and writing direction information. A compound string can consist of multiple segments. Each segment in the string can have a different character set and writing direction properties.
- connection:** The network path between a client and server.
- control:** A screen object that allows users to provide input to applications.
- control panel:** A subregion or dialog box containing controls often used during a work session. The control panel can remain on the screen during a work session.
- controls list:** In UIL, the sequence of widgets that defines the children controlled by a particular widget.

Glossary

conversion: In DRM, the translation of some information from its representation in a UID file into the internal form required by the XUI Toolkit or X Window System. Some conversions rely on information supplied through registration (for example, procedure names to addresses, identifier names to values). Others depend on information available through low-level XUI Toolkit procedure calls and other mapping transformations (for example, compound strings, font lists, colors, color tables, and icons).

creation callback: A callback that is invoked directly by DRM at the moment it creates a widget instance during fetch operations. It differs from most other callbacks in that the widget's change of state is not caused by an input event. Creation callbacks provide a mechanism for applications to discover the identifiers of widgets created by fetch operations. Creation callbacks are specified in UIL exactly like other callbacks; the callback reason is **create**.

cursor: An image that displays either the movement of the pointer or the current insertion point of text on the screen.

default character set: The character set, specified by a character set clause in a UIL module, used to interpret a string literal if no character set is specified for that literal.

default color table: The color table used by the UIL compiler when a color table is omitted from an icon definition.

default push button: The default option that provides the user with the most reasonable and least destructive response to a dialog box query. The default push button is selected when a user presses the Return key.

depth: The number of bits associated with a pixel.

dialog box: A special window that is displayed in response to user action. Usually, the user must take an appropriate action (as indicated by the alternatives presented in the dialog box) to continue application activity.

See also *caution dialog box*, *color dialog box*, *file dialog box*, *font attributes dialog box*, *message dialog box*, *modal dialog box*, *modeless dialog box*, *print dialog box*, *standard dialog box*, *work-in-progress dialog box*.

dim: To give an object a faded appearance, indicating that the object is inactive or disabled.

direct manipulation: Use of the pointing device to issue commands.

disable: To make inaccessible to the user.

discontinuous selection: Selection of two or more nonadjacent text or graphic objects.

dismiss: To remove a modeless dialog box.

display: The connection between a client and server.

do not propagate mask: Values that define which events should not be reported to the ancestors of a window.

- double-click:** A type of accelerator in which the user presses and releases a mouse button twice quickly without moving the mouse.
- double click:** To press and release a mouse button twice quickly without moving the mouse.
- drag:** To press and hold a mouse button, move the mouse, and then release the button when the pointer is in the desired position.
- drawable:** A window or pixmap.
- DRM (XUI Resource Manager):** A collection of XUI Toolkit routines for creating a set of widgets or retrieving values from a hierarchy of UID files. See also *DRM hierarchy*.
- DRM hierarchy:** The ordered list of UID files that DRM searches to find public resources. All DRM fetch or read operations have a scope of exactly one hierarchy. Applications may open more than one hierarchy.
- event:** An asynchronous report, sent by the server to a client, of either a change in the state of a device or the execution of a routine by another client.
- event mask:** Values that define which events associated with a window that the server should report to a client.
- exit:** To leave an application, automatically saving the file.
- exported resource:** A public resource defined in a given UID file; can be referenced by name in some other UID file.
- exposure:** A report that the server has made either a window or part of a window visible on the screen.
- extended selection:** An existing selection that has been altered by pressing and holding the Shift key in conjunction with MB1.
- fetch:** In DRM, the action of retrieving the description of a widget from a UID file and creating an instance of that widget. If the widget has any descendants (children), DRM also retrieves and creates these descendants in the same fetch operation. See also *widget hierarchy*.
- file dialog box:** A standard dialog box that solicits and accepts a file name from the user.
- font:** An array of glyphs, specifying the height, width, and shape of each symbol (uppercase and lowercase letters, numbers, punctuation marks, and so on) in a typeface.
- Fonts are stored in files. In UIL, a font consists of the name of a file (that holds the array of glyphs) and a character set (that specifies the expected character encoding for the font).
- font attributes dialog box:** A standard dialog box that displays fonts and font attributes from which the user can choose.
- font path:** A server-dependent description of where to find fonts.

Glossary

font table: A sequence of fonts often called a font list in the XUI Toolkit. When the XUI Toolkit needs to display a primitive string, it selects the correct font by searching its font table for the first font whose character set matches that of the primitive string.

foreground: The pixel value written by graphic objects.

gadget: A functionally limited widget; there is no window associated with a gadget. Gadgets use less memory than widgets and therefore provide better performance to applications. See also *widget*.

ghost image: An outline image of an object.

glyph: An image, typically of a character, in a font.

grab: A request by a client to get exclusive control of the keyboard, the pointer, or the server.

graphics context: A resource used to define the characteristics of graphic objects.

graphics exposure: A report by the server that part of the source for a copy operation is unavailable.

handle: A rectangular symbol on the border of a screen object that appears when that object is selected for operations such as moving, sizing, copying, or deleting.

help menu: Allows a user to access a help facility associated with a specific application.

help pointer: The pointer used with DECwindows Help.

hierarchy: The genealogical relationship of two or more windows.

horizontal pane pointer: The pointer used to reposition a vertical boundary between panes by moving the boundary left or right.

host: The system on which the client is operating.

hotspot: The point within the pointer that corresponds to the coordinate location of the pointing device.

icon: In the user environment, an icon is a symbol that represents an application, object, process, or window.

In the programming environment, an icon is a simple pixmap that can be described directly in UIL. A common use of an icon is to replace the text label of a button with a graphic symbol.

icon box: A special window that contains icons representing the applications available to the user.

identifier: In UIL, a placeholder for a value that is not known until run time. Not all values of arguments or tag values for callback procedures are known when the UID file is created. Identifiers receive values through registration of their names. See also *registration*.

- imported resource:** An object referenced in a particular UID file but defined in another UID file in the DRM hierarchy.
- inactive:** Not ready to accept user input.
- inactive insertion point:** An insertion point, indicated by a dimmed cursor, associated with text in an inactive window.
- inactive pointer:** The pointer that appears in regions of the application window that the application has rendered temporarily inactive, as when waiting for user input to a dialog box.
- inactive window:** A window that is not ready to accept input from the keyboard.
- index window:** A window attached to the scroll bar that offers a guide to the material to be displayed on the screen when the mouse button is released.
- indicator:** A symbol that designates the status of a radio or toggle button, or a radio or toggle item.
See also *radio indicator*, *toggle indicator*.
- inferior:** A subwindow of a specified window.
- input device:** A keyboard, mouse, tablet, track-ball, button, key, or other source of input to the workstation.
- input focus:** The ability to accept user input from the keyboard.
- insertion point:** The point on the screen where data will be inserted using the keyboard, the clipboard, or functions for creating graphic objects.
- interface module:** A list of all the top-level widgets defined in a UIL module. This list is stored in UID files in order to allow DRM to fetch an entire interface in a single fetch operation.
- key map:** The mapping between key symbols and key codes.
- key vector:** A list of the values of keys.
- keyboard:** Primary device for text insertion.
- label:** An inactive symbol or text that identifies a control.
- list box:** A dialog box component that displays a list of options, such as available files, from which the user can select one or more options. List boxes include scroll bars to allow users to move through lists that are too big to display in the dialog box.
- literal:** In UIL, a constant value. Common examples include ASCII strings, compound strings, or pixmaps. Exported literals may be fetched by DRM for arbitrary use by an application.
- main window:** A single window that is the starting point for all user interaction with an application.

Glossary

- map:** To make one or more windows visible on the screen.
- maximum slider:** A slider that fills the entire scroll region.
- MB1, MB2, MB3, ..., MB*n*:** Mouse button 1, mouse button 2, mouse button 3, mouse button *n*. Usually, MB1 is the left mouse button, MB2 the center button, and MB3 the right button; however, users can redefine the setup to make MB1 the right mouse button—usually because the particular user is left-handed. Functionally, MB1 is usually the primary selector, MB2 is often used to call pop-up menus, and any others are usually application defined.
- menu:** A list from which users can choose one or more items.
- menu bar:** A horizontal subdivision of a window that contains the names of pull-down menus.
- menu item:** A choice on any type of menu, including pop-up menus, pull-down menus, and submenus.
- menu name:** The title of a menu listed in the menu bar.
- message dialog box:** A standard dialog box that appears in response to a user action to report system or application information.
- minimum slider:** A slider that is a square based on the width of the scroll region.
- modal dialog box:** A dialog box that requires a user response before application activity can continue.
- modeless dialog box:** A dialog box that does not require a user response before application activity can continue.
- mouse:** A pointing device that, when moved across any surface, causes a corresponding movement of the pointer. A mouse can have one or more buttons.
- mouse button:** A button on a mouse.
- name:** An active symbol or text that identifies a control or menu. In UIL, a user-defined string identifying a value or object.
- null-terminated string:** A sequence of characters terminated by an ASCII null (\n).
- object:** An entity on the screen, such as a button, control, graphic, icon, menu, pointer, text, and so on.
In UIL, a construct that can be named. Widgets, identifiers, values, procedures, and lists are example of objects.
- object selection:** Selection of graphic screen entities, such as pictures.
- object text insertion pointer:** The pointer that specifies the point where text can be entered.
- obscure:** Window A obscures window B if both are viewable input-output windows, if A is higher in the stacking order than B, and if the rectangle defined by the outside edges of A intersects the rectangle defined by the outside edges of B.

occlude: Window A occludes window B if both are mapped, if A is higher in the stacking order than B, and if the rectangle defined by the outside edges of A intersects the rectangle defined by the outside edges of B.

option box: A control within a dialog box, consisting of a label and an active region, that shows the available options and the current option selected.

option menu: Control that consists of a label and an active area that shows the current option selected. Clicking on the active area produces a pop-up menu.

origin: The reference point (0,0) in a window, pixmap, or graphic object that is used to determine the position of other windows, pixmaps, and graphic objects.

override redirect: A request that the server ignore redirect instructions for a window.

pane: A special type of region whose size can be adjusted.

parent: The first-level ancestor of a window.

passive grab: A request to get control of the pointer or keyboard when specific keys or buttons are pressed, or when the pointer is in a specific window.

PB1, PB2, PB3, PB4, ..., PB n : Puck button 1, puck button 2, puck button 3, puck button 4, puck button n . Usually, PB1 is the left puck button, PB2 the top center button, PB3 the right button, and PB4 the bottom center button; however, users can redefine the setup to make PB1 the right puck button—usually because the particular user is left-handed. Functionally, PB1 is usually the primary selector, PB2 is often used to call pop-up menus, and any others are usually application defined.

pending delete: A state in which selected text will disappear when a key is pressed; the selected text is replaced with keyboard input.

pixel: The smallest definable graphic image on a screen.

pixel value: Each pixel on the screen has a pixel value that controls the visual effect of that pixel. On monochrome displays, only two pixel values exist: FOREGROUND and BACKGROUND. However, on color displays, the number of pixel values supported determines the range of colors that can be displayed.

UIL does not directly support pixel values because the individual values are server dependent. Instead, UIL supports colors that are mapped to pixel values at run time by DRM. See also *color*, *color table*.

pixmap: A two-dimensional array of nonnegative integers that represents an image. Each integer contains the value of the corresponding pixel in the image.

In the XUI Toolkit, a pixmap is represented using a data structure that holds the height and width of the image in pixels, the depth of the image (range of values each pixel can take), and the pixel value of each pixel.

In UIL, a pixmap is represented as an icon. UIL pixmaps are device independent; therefore, they are described by specifying the color of each pixel rather than its pixel value. See also *color*, *color table*.

Glossary

- plane:** An allocation of memory in which there is a one-to-one correspondence between pixels and bits.
- pointer:** An on-screen symbol that specifies position by reflecting the motion of the mouse. The pointer shape indicates the type of operation being performed.
- pointer shape:** The form of the pointer. Different pointer shapes indicate the type or state of the operation the user has chosen.
- pointer speed:** The relationship between the distance that the user moves the mouse and the distance that the pointer moves on the screen.
- pointing device:** The hardware used to control pointer position on the screen.
See also *mouse, puck, stylus, tablet*.
- pointing device button:** A button on the hardware used to control pointer position on the screen.
- pop-up menu:** A context-sensitive menu that appears at the current pointer position when the user presses the mouse button defined for this function.
- prefill:** The first text entry field in a dialog box that is filled with the current selection.
- primary selector:** The pointing device button that initiates actions, such as menu manipulation and selection or cut and paste operations.
- print dialog box:** A standard dialog box with which the user can choose attributes related to the printing of a file.
- private resource:** A resource whose scope is the UID file in which it is defined. Private resources can only be referenced by other resources in the same UID file.
- procedure name:** The case-sensitive name of a callback routine stored by UIL in callbacks lists in UID files. DRM converts procedure names to procedure addresses using information registered by the application. See also *callbacks list, conversion, registration*.
- progressive text selection hierarchy:** A selection hierarchy that increases during successive clicks of MB1.
- property:** A collection of named, typed data associated with a window.
- pseudomotion:** The apparent movement of the cursor when a client grabs the pointer, causing the server to reposition the cursor from one window to another, even though the pointer has not moved.
- public resource:** A resource whose scope is the entire DRM hierarchy of UID files. Public resources can be referenced by other resources anywhere in the DRM hierarchy.
- puck:** A pointing device used with a tablet. Moving a puck across the tablet causes a corresponding movement of the pointer. A puck can have one or more buttons.
See also *PB1, PB2, PB3, PB4, ..., PBn; pointing device; tablet*.

- pull-down menu:** A menu that is displayed when the user presses a mouse button when the pointer is positioned on a menu name in the menu bar.
- push button:** A control consisting of a rectangular box and a label that indicates the command to be performed.
- push-to-back button:** A button within the window title bar that allows users to move the window to the back of the window stack.
- quit:** To leave an application. The user will be prompted as to whether to save changes to a file.
- radio button:** An on-screen control consisting of a button name with an indicator next to it. Radio buttons interact in a special way: only one can be chosen at a time; choosing one turns off any previously chosen button.
- radio icon:** One of a set of icons, from which only one icon can be chosen at a time.
- radio indicator:** A circle, which is part of a radio button or a radio item, that designates which button or item from a set of radio buttons or items has been chosen.
- radio item:** One item in a set of items on a menu, consisting of a name and an indicator for the chosen item. Radio items interact in a special way: only one can be chosen at a time; choosing one turns off any previously chosen item.
- Redo:** A type of operation that reverses the effects of the last Undo operation performed.
- region:** An area of a window or pixmap.
- registration:** The provision of information to DRM at run time that is needed to correctly interpret information contained in a DRM hierarchy of UID files.
- DRM supports two registration operations. Class registration provides the information necessary to interpret definitions of user-defined widgets. Name registration provides the mapping between names such as procedures and identifiers stored as strings in a UID file and the run-time values associated with the names (for example, the addresses of callback routines).
- reparent:** To change the location of a window in the hierarchy. Reparenting windows is limited to the window manager.
- reply:** A message sent synchronously from the server to a client.
- resize:** To change the size of a screen object.
- resize button:** A control used to resize a window.
- resize pointer:** The pointer used for resizing operations.
- resource:** A data object stored in a UID file that is a component of an interface specification (for example, a widget, a color, a font, a string). DRM retrieves resources from a UID file and then interprets them as required.

Glossary

resource context: A data structure used by the DRM to retrieve resources from UID files, providing flexible memory management. Applications must use resource contexts to retrieve literals from UID files.

resource database: A static, binary-encoded, read-only database containing the specification of a DECwindows application user interface, or some components of an interface (for example, a given resource database may contain only literals). See also *UID file*.

root window: The initial window on a screen.

save set: A list of windows that will not be destroyed if the client-server connection is broken. Save sets are typically used by window managers only.

save under: Instructions to the server regarding whether to save the screen contents when a window hides them.

SB1, SB2, ..., SB n : Stylus button 1, stylus button 2, stylus button n . SB1 is the button on the barrel of the stylus, SB2 the button on the tip of the stylus. Functionally, SB1 is usually the primary selector, SB2 is often used to call pop-up menus, and any others are usually application defined.

See also *primary selector, stylus*.

scale: Control that allows the user to enter a numerical value by adjusting a pointer to a specific position along a line.

scope: The visibility of a UIL name within the DRM hierarchy of UID files. An exported object in the UIL has a scope of the entire hierarchy and so can be referenced by other objects in any of the UID files in the hierarchy. A private object has scope only within an individual UID file and so cannot be referenced by objects in other UID files.

scroll bar: A subregion that allows users to move through a block of information that is too big to be displayed at one time.

scroll region: A scroll bar component in which the slider moves. The relationship between the size of the slider and the size of the scroll region corresponds to the relationship between the size of the material displayed and the size of the file.

select: To designate information, either text or graphics, that will be the object of a subsequent operation or operations.

select pointer: The pointer used for selection operations.

selection: Text or graphics that will be the object of a subsequent operation or operations.

selection hierarchy: Multiple clicks of MB1 used to select successively larger (or smaller) text regions.

sensitivity: In the XUI Toolkit, a description of the state of a widget. An "insensitive" widget is dormant and does not receive notification of input events that may occur within it. The screen appearance of many widgets changes when they go into dormancy.

- server:** The program that controls workstation devices such as screens and pointers.
- shift click:** To position the pointer where you want the action to occur, and then press and hold the Shift key while clicking MB1.
- shrink-to-icon button:** A control that shrinks a window to an icon.
- sibling:** One of two or more windows created from the same parent.
- size:** To define the size of screen objects.
- slider:** A scroll bar component used to move a window over information that is too big to be displayed at one time. The relationship between the size of the slider and the size of the scroll region corresponds to the relationship between the size of the material displayed and the size of the file.
- standard dialog box:** Any specific-purpose dialog box supplied with the DECwindows Toolkit.
- See also *caution dialog box, color dialog box, file dialog box, font attributes dialog box, message dialog box, print dialog box, work-in-progress dialog box.*
- standard menu:** Any specific-purpose menu supplied with the DECwindows Toolkit.
- status region:** A region that provides users with information about the state of the operating system or application.
- stepping arrow:** A scroll bar component used to move a window incrementally through information that is too big to be displayed at one time.
- stipple:** A pixmap with a depth of 1, used to tile a region or window with foreground and background values.
- structure control:** The management of requests by other clients to resize, configure, circulate, and map windows.
- stylus:** A pointing device used with a tablet. Moving the stylus across the tablet causes a corresponding movement of the pointer. The tip of the stylus usually functions as a button; the barrel of the stylus can have one or more additional buttons.
- submenu:** A menu, associated with a pull-down or pop-up menu, that is displayed in response to dragging the pointer over a submenu icon. The availability of a submenu is shown on a menu by a submenu icon.
- See also *pop-up menu, pull-down menu, submenu icon.*
- submenu icon:** A symbol, associated with a menu item, that signifies the availability of a submenu. Dragging the pointer over the icon causes the submenu to appear.
- subregion:** A subdivision of a region. Applications can divide regions into multiple subregions.

Glossary

- substructure redirect:** An instruction from a client for the server to send configure and map requests by children of the window to the client rather than acting on the requests directly.
- symbolic reference:** In a UIL module, the use of a widget name in place of a widget identifier as an argument for a widget in the same widget hierarchy.
- tablet:** The surface across which a puck or stylus is moved.
- tabular selection:** Selection of a unit within a table or form.
- tag:** An argument to a callback procedure allowing the application programmer to pass arbitrary information to the procedure. A typical use of the tag is to identify the reason for a callback. For example, a single callback procedure may be invoked for multiple reasons. The programmer can use the tag to store a constant that identifies the particular reason the procedure is being invoked.
- text insertion cursor:** A block-shaped cursor that shows where text will be entered in a window. This cursor indicates that text will be entered in insert mode.
- text insertion pointer:** The pointer that specifies the point where text can be entered.
- text overstrike cursor:** A cursor shaped like an I-beam or a capital I that shows where text will be entered in a window. This cursor indicates that text will be entered in overstrike mode.
- text-entry field:** A dialog box control that allows the user to enter text in a structured area.
- tile:** To set up the screen so that windows do not overlap; to paint an area by replicating a pixmap within the area.
- title bar:** A horizontal bar in a window that identifies the application and contains window management controls.
- toggle:** To switch a two-state option to its opposite state.
- toggle button:** An on-screen control consisting of a button name with an indicator next to it. Toggle buttons allow users to choose one of two alternate states.
- toggle indicator:** A square, which is part of a toggle button or toggle item, that designates which buttons or items have been chosen.
- toggle item:** An item on a menu, consisting of a name and an indicator designating each item chosen. A menu can include more than one toggle item.
- top-level widget:** The widget you get back from a call to the X Toolkit routine INITIALIZE. This widget does not have a parent. All widgets fetched by the DRM must have a parent, either a top-level widget or some widget previously created by a fetch operation.
- triple click:** To position the pointer where you want the action to occur, and then press and release a mouse button three times quickly without moving the mouse.

- UID (User Interface Definition) file:** A binary-encoded file containing the output of the UIL compiler that is read by DRM at application run time. See also *DRM hierarchy*.
- UIL (User Interface Language):** A strongly typed specification language for defining the initial state of a user interface for a DECwindows application; used in conjunction with the XUI Resource Manager (DRM).
- UIL include directive:** A UIL directive that incorporates the contents of a specified file into the UIL module.
- UIL include file:** A file that holds a set of UIL statements, which may define values, widgets, and so on. Digital provides DECW\$DWTDEF.UIL (VAX binding) and DwtAppl.uil (C binding), which define standard enumeration values for XUI Toolkit widgets.
- UIL module:** The contents of a UIL specification file. The name of the UIL module follows the MODULE keyword.
- Each UID file contains an interface module that provides DRM access to all top-level widgets defined in a UIL module. The name of this interface module is the UIL module name.
- UIL specification file:** An ASCII-encoded file containing the definition of the initial state of the user interface for an application; the file that holds a UIL module. UIL (and UID) files are separate and distinct from application programs.
- Undo:** A type of operation that reverses the effects of the last operation performed.
- User Interface Definition file:** See *UID file*.
- User Interface Language:** See *UIL*.
- user-defined widget:** A widget designed and built by an application programmer, not part of the XUI Toolkit or the X Toolkit.
- version clause:** Specifies the version of a UIL module.
- vertical pane pointer:** The pointer used to reposition a horizontal boundary between panes by moving the boundary up and down.
- See also *pane*.
- viewable:** A window is viewable if it and its ancestors are mapped. The window does not have to be visible. For example, if a window is mapped but hidden by another window, it is viewable.
- visible:** A window is visible if one can see it on the screen.
- visual:** A resource that describes how pixel values are interpreted on a screen. For example, a screen may be configured variously to display color or black and white images.
- wait pointer:** The pointer used to indicate that work is in progress.

Glossary

widget: A primary component of an interface to a DECwindows application.

For a user of an application, a widget appears on a display as a collection of graphics (such as a window, title bar, or menu) and text. Using the keyboard and mouse, the user manipulates this visual manifestation of a set of widgets to communicate with the application.

For an application developer, a widget is a data structure and a collection of routines that will create, destroy, and modify the data structure. The contents of the data structure are a set of resources including arguments that control the visual manifestation, callbacks that describe routines to be called when the user manipulates the widget, and child widgets that define and control subfunctions of the widget.

In UIL, a widget is a declaration that specifies the initial state a widget is to take when DRM fetches the widget. By using the UIL, an application programmer can specify the initial arguments, callbacks, and children of a widget.

widget class: A means of grouping widgets according to common properties and functions. Widgets can be classified as main windows, scroll bars, menu bars, and so on. Each of these is a separate widget class. An application may consist of zero or more widgets of each widget class. The class of a widget is fixed when the widget is created and determines which additional resources it supports.

widget creation: In the XUI Toolkit, the process through which an application obtains an instance of a widget. In the creation process, the widget is allocated and initialized with resources. See also *widget management* and *widget realization*.

widget hierarchy: The hierarchy formed by the widgets in an application user interface that is defined by parent-child relationships of the widgets. When DRM fetches a widget, the entire widget hierarchy rooted at that widget is also fetched. A widget hierarchy may be resolved across a DRM hierarchy; that is, the widget definitions are not required to be retrieved from a single UID file.

widget identifier: An identifier for a widget generated by a DRM fetch operation or by an XUI Toolkit creation routine.

widget management: In the XUI Toolkit, the process through which a widget becomes displayable. When an application creates an instance of a widget through the widget creation process, the widget does not appear on the display. To make the widget displayable, the application must manage the widget, that is, add the widget to the set of widgets controlled by a particular parent widget. The parent widget is responsible for the overall management and physical arrangement of its children. Every widget in an application is the child of some other widget in the application. When a parent widget is realized, it and all of the children managed are mapped on the display. See also *widget realization*.

widget realization: In the XUI Toolkit, the process that causes a widget to appear on the display. The realization process obtains a window for the widget and, if the widget is managed, maps the window to the display. See also *widget management*.

window: An area on the screen in which users can interact with an application.

window crossing: The window entry or exit of the cursor.

window gravity: The location of a window after its parent has been resized and repainted.

window manager: The application that governs sizing, positioning, and stacking of windows, determines the object that has input focus, and creates and controls icons and the icon box.

window stack: Windows on the workstation screen that overlap, like sheets of paper stacked one on top of another.

work region: A subdivision of a window or subwindow in which users perform most application tasks.

work-in-progress dialog box: A standard dialog box that indicates that an application is completing an ongoing operation.

XUI Resource Manager: See *DRM*.

Index

A

- Accelerator • D-71
 - adding to a widget or gadget • 5-28
 - definition • 5-27
 - installing an accelerator in a widget • 5-28
 - using with command window widget • 4-17
 - using with push button widget • 5-27
 - using with push button widgets in a dialog box widget • 7-12
 - using with scroll bar widget stepping arrows • 10-14
 - using with toggle button widget • 5-27
- Accelerator label • 5-29
- Action procedure
 - adding to a widget • 6-27
 - definition • 6-26
 - example • 6-27
 - standard arguments • 6-26
- Action table • D-60
 - definition • 6-27
- ACTIVATE WIDGET routine • 5-1
- Ada
 - Hello World! sample application • B-7
 - programming interfaces • B-1
- ADD ACTIONS routine • 6-28, D-61
- ADD CALLBACKS routine • 2-31
- ADD EVENT HANDLER routine • D-47
- ADD FONT LIST routine • 5-25
- ADD RAW EVENT HANDLER routine • D-47
- ADD WORK PROC routine • 2-30
- Application
 - associating form with function • 1-7
 - creating form of • 1-6
 - interaction with window manager • 14-1
 - specifying the name that appears in the title bar • 14-18
 - starting in iconic state • 14-27
 - structure of • 2-1
- APPLICATION ADD CONVERTER routine • D-56
- Application context
 - creating • 2-5
 - definition • 2-5
 - routines • 2-6, 2-26
- APPLICATION CREATE SHELL routine • 2-5, 2-7, 2-8
- APPLICATION CREATE SHELL routine (Cont.)
 - setting shell widget attributes • 14-11
- APPLICATION MAIN LOOP routine • 2-26
- Application shell widget • 2-6
 - assigning values to attributes • 14-13
 - attributes that must be set at creation time • 14-11
 - communicating with the window manager • 14-8, 14-11
 - creating • 2-4
 - definition • 1-6
 - setting attributes at creation time • 14-11
 - using multiple • 2-7
- Application widget hierarchy • 1-6
 - specifying in UIL • 3-8
 - traversing up and down • D-27
- Argument
 - defining for user-defined widget in UIL • 3-66
- Argument data structure • 2-10
- Argument list
 - specifying in UIL • 2-18, 3-25
 - using with low-level widget creation routines • 2-10
- ARGUMENTS keyword • 3-25
- ATTACHED DIALOG BOX CREATE routine • 7-13
- ATTACHED DIALOG BOX POPUP CREATE routine • 7-13
- ATTACHED DIALOG BOX routine • 7-13
- Attached dialog box widget
 - attaching a child to • 7-16
 - attaching a child to another child • 7-17
 - attachment types • 7-15
 - callbacks • 7-22
 - comparing to dialog box widget • 7-2
 - creating • 7-13
 - customizing • 7-21
 - defining attachments • 7-14
 - defining offsets between children • 7-22
 - fractional positioning • 7-18
 - convenience attachment attribute • 7-18
 - defining the default fraction denominator • 7-22
 - positioning child widgets • 7-14
 - resizing behavior • 7-2
 - rubber positioning • 7-22
 - specifying in UIL • 3-17
- Attachment attributes
 - definition • 7-15
 - summary table • 7-15
 - using • 7-19

Index

Attributes

See Widget attributes

AUGMENT TRANSLATIONS routine • D-71

B

BEGIN COPY TO CLIPBOARD routine • 13-2

Bidirectional text cursor

definition • 9-1

Blink rate

specifying for text cursor • 9-10

Border visibility

in text widgets • 9-11

BUILD EVENT MASK routine • D-48

Building widgets • D-1

sample widget • D-2

using UIL • 3-65

Button widget

See Push button widget

See Toggle button widget

C

Callback data structure • 2-27

Callback mechanism

definition • 1-7

Callback reason

See Reason

Callback routine

associating with a widget using UIL • 3-27

clipboard callback routine format • 13-10

creating • 2-27

example • 2-28

standard arguments • 2-27

using UIL identifier to pass data structure to • 3-58

Callback routine data structure • 2-12

Callback routine list

specifying in UIL • 2-19, 3-27

using with high-level widget creation routines • 2-16

using with low-level widget creation routines • 2-12

CALLBACKS keyword • 3-27

Callbacks list entry • 3-27

Callback tag

See Tag

CANCEL COPY FORMAT routine • 13-1

CANCEL COPY TO CLIPBOARD routine • 13-1

Caution box widget • 7-4

C binding

See MIT C binding

Character

See Text character

Character sets

specifying in a compound string • 5-20

specifying in compound string text widget • 9-1

Children

clipping • 1-6

definition • 1-6

managing • 2-21

managing multiple • 2-23

positioning in a dialog box widget • 7-6

positioning in an attached dialog box widget • 7-14

specifying in UIL • 3-26

Class

definition • D-1

Clear function

in Edit menu • 13-5

Client message event

used by window manager • 14-14

Clipboard

accessing • 13-1

callback routine format • 13-10

communicating with other applications • 13-3

copying data from • 13-11

copying data to • 13-5

data formats • 13-11

deleting data from the clipboard • 13-11

description • 13-1

inquire routines • 13-15

locking the clipboard • 13-3, 13-15

passing data by name • 13-9

typical copy operation • 13-4

typical paste operation • 13-4

CLIPBOARD LOCK routine • 13-3, 13-15

CLIPBOARD REGISTER FORMAT routine • 13-2, 13-11

CLIPBOARD UNLOCK routine • 13-3, 13-15

Clipping of child widgets • 1-6

CLOSE HIERARCHY routine • 3-34

Coding techniques

in UIL • 3-5

Color

color mixing widget • 11-1

color models • 11-14

defining colors • 11-1

RGB values • 11-2

server names for defining in UIL • 3-22

- COLOR function
 - in UIL • 3–22
- COLOR MIX CREATE routine • 11–7
- COLOR MIX GET NEW COLOR routine • 11–8
- Color mixing widget
 - callbacks • 11–14
 - color display subwidget • 11–4
 - defining the background color • 11–13
 - replacing • 11–14
 - color mixer subwidget • 11–5
 - replacing • 11–14
 - color resources required • 11–5
 - components • 11–2
 - creating • 11–7
 - customizing • 11–9
 - deleting labels from • 11–10
 - label subwidgets • 11–7
 - margins • 11–9
 - on grayscale device • 11–5
 - overview • 11–1
 - push button subwidgets • 11–6
 - retrieving the new color • 11–8
 - setting the new color • 11–8
 - sizing • 11–9
 - specifying labels in • 11–10
 - supporting color models • 11–14
 - support routines • 11–9
 - work area subwidget • 11–7
 - specifying • 11–13
- COLOR MIX SET NEW COLOR routine • 11–8
- Color models
 - HLS • 11–1
- Color table
 - default in UIL • 3–24
- Color values
 - defining in UIL • 3–22
- COLOR_TABLE function
 - in UIL • 3–23
- COMMAND APPEND routine • 4–16
- COMMAND ERROR MESSAGE routine • 4–16, 4–17
- Command line
 - command history display • 4–17
 - specifying command in command window widget • 4–16
 - specifying the prompt for • 4–17
- Command line prompt • 4–17
- Command procedure
 - for mapping colors • 3–22
- COMMAND SET routine • 4–16
- COMMAND WINDOW CREATE routine • 4–15
- COMMAND WINDOW routine • 4–15
- Command window widget
 - callbacks • 4–18
 - creating • 4–15
 - customizing • 4–17
 - overview • 4–2
 - sizing the command history display • 4–17
 - support routines
 - summary • 4–16
- Communicating between applications
 - using the QuickCopy function • 13–16
- Communicating with other applications
 - using the clipboard • 13–3
- Composite widget • 1–6
 - adding children to • D–34
 - as container for other widgets • D–27
 - class part data structure • D–14
 - instance data structure • D–14
 - default values • D–15
 - removing children • D–35
 - specifying the order of children • D–34
- Compound string
 - comparing • 5–23
 - containing multiple segments • 5–22
 - copying • 5–23
 - creating • 5–21
 - extracting text from a compound string • 5–23
 - finding the length • 5–23
 - freeing memory • 5–22
 - overview • 5–19
 - retrieving information about • 5–23
- Compound string context
 - definition • 5–23
 - freeing • 5–24
 - initializing • 5–24
- Compound string routines
 - summary • 5–20
- Compound strings, multiline • 3–20
- Compound string text widget
 - bidirectional text cursor • 9–1
 - callbacks • 9–13
 - controlling resizing behavior • 9–10
 - creating • 9–4
 - customizing • 9–8
 - determining editing direction • 9–11
 - determining positions in • 9–6
 - determining writing direction • 9–11
 - disabling text editing • 9–7
 - half-border visibility • 9–11
 - including scroll bars in • 9–10
 - margins • 9–9
 - overview • 9–1

Index

- Compound string text widget (Cont.)
 - placing text in • 9-6
 - recommended way to specify size • 9-8
 - retrieving current value • 9-7
 - selecting text • 9-12
 - specifying insertion position • 9-11
 - support routines
 - advantages • 9-6
 - summary • 9-3
 - text cursor
 - specifying the blink rate • 9-10
- COMPOUND_STRING function
 - in UIL • 3-6, 3-19
- CONFIGURE_WIDGET routine • D-37, D-41
- Constraint widget
 - class part data structure • D-15
 - defining arguments for in UIL • 3-17
 - defining subclasses • D-35
 - destroy procedure • D-32
 - initializing • D-29
 - instance data structure • D-16
- Context-sensitive help
 - creating • 12-13
- CONTROLS keyword • 3-26
- Controls list • 3-26
- CONVERT_CASE routine • D-73
- CONVERT routine • D-56
- COPY FROM CLIPBOARD routine • 13-2
- CopyFrom operation • 13-16
- Copy function
 - description • 13-4
 - how to implement • 13-5
 - in Edit menu • 13-3
- Copying data from the clipboard
 - incremental copying • 13-12
 - procedure • 13-11
- Copying data to the clipboard
 - passing data by name • 13-9
 - procedure • 13-5
- COPY TO CLIPBOARD routine • 13-1
- CopyTo operation • 13-16
- Core widget • D-1
 - class part data structure • D-11
 - instance data structure • D-12
 - default values • D-12
- CREATE APPLICATION CONTEXT routine • 2-5
- CREATE FONT LIST routine • 5-25
- CREATE MANAGED_WIDGET routine • D-33
- CREATE POPUP_SHELL routine • 2-7
- CREATE_WIDGET routine • D-33, D-48
 - obtaining widget resources • D-52
- CREATE_WIDGET routine (Cont.)
 - using with constraint widget • D-36
- Creating a DECterm
 - example • A-1
- CS BYTE_CMP routine • 5-23
- CS CAT routine • 5-22
- CS COPY routine • 5-23
- CS EMPTY routine • 5-23
- CS LEN routine • 5-23
- CS STRING routine • 5-21
- CS TEXT CLEAR SELECTION routine • 9-4
- CS TEXT CREATE routine • 9-4
- CS TEXT GET EDITABLE routine • 9-3
- CS TEXT GET MAX LENGTH routine • 9-3, 9-8
- CS TEXT GET SELECTION routine • 9-3, 9-12
- CS TEXT GET STRING routine • 9-3, 9-7
- CS TEXT REPLACE routine • 9-3, 9-6
- CS TEXT routine • 9-4
- CS TEXT SET EDITABLE routine • 9-3
- CS TEXT SET MAX LENGTH routine • 9-3, 9-8
- CS TEXT SET SELECTION routine • 9-4, 9-12
- CS TEXT SET STRING routine • 9-3, 9-6
- Customizing widgets • 1-2
- Cut and paste routines • 1-4
 - callback routine format • 13-10
 - data formats • 13-11
 - inquiring about clipboard contents • 13-15
 - list of • 13-1
 - overview • 13-1
 - passing data by name • 13-9
- Cut function
 - description • 13-4
 - in Edit menu • 13-3
- Cut routines
 - See Cut and paste routines

D

- DECBURGER.COM command procedure • 3-14
- DECburger sample application • 3-4
 - creating the dialog box widget • 7-8
 - creating the list box widget • 8-5
 - creating the main window widget • 4-7
 - creating the menu bar widget • 6-18
 - creating the radio box widget • 5-14
 - French version • 3-50
 - international version • C-1
 - introduced • 1-6
 - user interface • 1-11

Declaring objects in UIL
 as a template • 3–45
 DECterm
 creating • A–1
 DECTERM PORT routine • A–1
 DECW\$DWTDEF.H include file • 3–32
 DECW\$DWTDEF.UIL include file • 3–13
 DECW\$RGB.COM command procedure • 3–22
 DECwindows window manager
 communicating with • 14–10
 extensions • 14–4
 DEC WM Hints data structure
 definition • 14–5
 DEC_WM_DECORATION_GEOMETRY property •
 14–5
 definition • 14–7
 DEC_WM_HINTS property • 14–5
 definition • 14–5
 Deleting data from the clipboard • 13–11
 DESTROY WIDGET routine • 2–8, D–33
 traversing the application widget hierarchy • D–27
 using with constraint widget • D–36
 DIALOG BOX CREATE routine • 7–5
 DIALOG BOX POPUP CREATE routine • 7–5
 DIALOG BOX routine • 7–5
 Dialog box widget
 See also Attached dialog box widget
 callbacks • 7–12
 comparing to attached dialog box widget • 7–2
 creating • 7–5
 customizing • 7–10
 defining accelerators for push button widgets •
 7–12
 generic • 7–1
 handling the input focus • 7–12
 modal style • 7–5
 modeless style • 7–5
 overview • 7–1
 pop-up • 7–4
 positioning • 7–11
 positioning child widgets • 7–6
 resizing behavior • 7–2
 controlling • 7–11
 sizing • 7–10
 specifying translations for text widgets • 7–11
 specifying unit of measure • 7–11
 standard • 7–4
 types of • 7–4
 work area style • 7–4
 DIRECT CONVERT routine • D–56
 DISPATCH EVENT routine • 2–26

DRM (XUI Resource Manager)
 creating a user interface with • 3–31
 definition • 1–4
 include file for constants • 3–32
 initialization of • 3–2
 list of routines • 3–34
 registering names for • 3–36
 registering user-defined classes for • 3–71
 retrieving literal values from UID files • 3–38
 scope of references in UID file • 3–16
 using to create widgets in a user interface • 2–20
 DRM FREE RESOURCE CONTEXT routine • 3–34
 DRM GET RESOURCE CONTEXT routine • 3–34
 DRM HGET INDEXED LITERAL routine • 3–34
 DRM RC BUFFER function • 3–34
 DRM RC SET TYPE function • 3–34
 DRM RC SIZE function • 3–34
 DRM RC TYPE function • 3–34
 DwtAnyCallbackStruct
 definition • 2–27
 DwtAppl.h include file • 3–32
 DwtAppl.uil include file • 3–13
 DwtXLatArg.uil • 3–51
 DwtXLatText.uil • 3–51

E

Editing path
 in a compound string text widget • 9–11
 Editing text
 using the text widgets • 9–1
 Edit menu
 functions • 13–3
 list of • 13–5
 END COPY FROM CLIPBOARD routine • 13–2
 END COPY TO CLIPBOARD routine • 13–1
 Error message
 displaying in command window widget • 4–17
 Event
 handling expose events in a window widget • 4–12
 ignoring the event queue • 2–29
 processing loop • 2–25
 receiving from window manager • 14–14
 Event management • D–44
 procedure • D–47
 Events
 relationship to callback reasons • 1–8
 types of • D–66
 Event specification • D–62

Index

Event specification (Cont.)
 in a translation table • 5-27
Example program
 See DECburger sample application
 See Hello World! sample application
EXPORTED keyword • 3-16
Exported resource
 defining in UIL • 3-16
Expose event
 redisplaying a widget • D-45
Exposure • D-45

F

FETCH COLOR LITERAL routine • 3-38
FETCH ICON LITERAL routine • 3-38
Fetching off-screen widgets
 deferred using DRM routines • 3-37
FETCH INTERFACE MODULE routine • 3-34
FETCH LITERAL routine • 3-38
FETCH NAME routine • 14-9
Fetch operation
 definition • 3-3
FETCH SET VALUES routine • 3-40
FETCH WIDGET OVERRIDE routine • 3-45
FETCH WIDGET routine • 2-20, 3-3, 3-37
File selection widget • 7-4
Font
 determining font used by window manager • 14-22
 specifying in text strings • 5-25
Font list
 definition • 5-25
Font unit
 definition • 7-11
Font values
 defining in UIL • 3-21
Format
 data format used by clipboard • 13-11
FORTRAN
 Hello World! sample application • B-11
Fractional positioning
 defining the default fraction denominator • 7-22
 in an attached dialog box widget • 7-18
FREE routine • 5-22
Functions (UIL)
 COLOR • 3-22
 COLOR_TABLE • 3-23
 COMPOUND_STRING • 3-6, 3-19
 DRM RC BUFFER • 3-34

Functions (UIL) (Cont.)
 DRM RC SET TYPE • 3-34
 DRM RC SIZE • 3-34
 DRM RC TYPE • 3-34
 ICON • 3-23
 STRING TABLE • 8-3
 XBITMAPFILE • 3-23

G

Gadgets
 as menu items • 6-3
 definition • 1-2
 label • 5-1
 pull-down menu entry • 6-3
 push button • 5-1
 separator • 5-1
 toggle button • 5-1
Geometry management • 1-14
 creating a procedure • D-40
 negotiation between parent and child • D-39
 of widgets • D-37
 return codes from a procedure • D-38
 techniques • D-33
GET APPLICATION RESOURCES routine • D-52
GET CLASS HINT routine • 14-9
GET ICON NAME routine • 14-9
GET ICON SIZES routine • 14-9
GET NEXT SEGMENT routine • 5-24
GET NORMAL HINTS routine • 14-9
GET RESOURCE LIST routine • D-51
GET SELECTION OWNER routine • 13-24
GET SELECTION VALUE routine • 13-19, 13-22
GET SUBVALUES routine • D-57
GET TRANSIENT FOR HINT routine • 14-9
GET VALUES routine
 using with constraint widget • D-36
GET WINDOW PROPERTY routine • 14-11
GET WM HINTS routine • 14-9
Global values
 defining in UIL • 3-16
Graphics
 using the window widget • 4-12

H

Half-border
 definition • 9-1

- Hello World! sample application
 - complete source listing • 2–32
 - introduced • 1–4
 - in VAX Ada • B–7
 - in VAX FORTRAN • B–11
 - in VAX Pascal • B–14
 - Help file
 - sample of • 12–6
 - Help library
 - default file type • 12–4
 - include command • 12–5
 - key names • 12–4
 - keyword command • 12–5
 - sample help file • 12–5
 - title command • 12–5
 - using with help widget • 12–3
 - Help session
 - definition • 12–3
 - Help widget
 - C code example • 12–9
 - creating context-sensitive help using • 12–13
 - creation routines • 12–9
 - definition • 12–3
 - description • 12–1
 - DWT\$C_TEXT_LIBRARY value • 12–3
 - DwtTextLibrary value • 12–3
 - first topic attribute • 12–7
 - glossary topic attribute • 12–7
 - invoking • 12–2
 - library information • 12–3
 - library type attribute • 12–3
 - modifying appearance • 12–7
 - overview topic attribute • 12–7
 - sample • 12–1
 - specifying in UIL • 3–46
 - specifying topic • 12–7
 - terminology • 12–3
 - UIL code example • 12–11
 - using • 12–8
 - Help window
 - definition • 12–3
 - High-level widget creation routines • 2–14
 - compared to low-level routines • 1–3
 - using to associate callback routines with a widget • 2–16
 - using to define initial appearance of widget • 2–15
 - using to define parent/child relationship • 2–15
 - HLS color model • 11–1
 - Hue, Lightness, Saturation Color Model
 - See HLS color model
-
- ICCCM (*Inter-Client Communication Conventions Manual*)
 - cut and paste routine compliance • 13–2
 - data formats • 13–11
 - Icon
 - description • 14–23
 - positioning in Icon Box • 14–27
 - screen appearance under DECwindows window manager • 14–15
 - specifying in UIL • 3–23
 - specifying the pixmap used as • 14–24
 - specifying the text in • 14–24
 - using as a label in UIL • 3–29
 - using a window as • 14–26
 - Icon Box
 - definition • 14–23
 - positioning icons in • 14–27
 - screen appearance under DECwindows window manager • 14–15
 - ICON function
 - in UIL • 3–23
 - using to create pixmap labels • 5–5
 - Iconic state
 - starting your application in • 14–27
 - Iconify button
 - See Shrink-to-icon button
 - Identifier
 - defining in UIL • 3–57
 - registering for DRM • 3–36
 - using in object declaration template • 3–45
 - using to pass data structure to callback routine • 3–58
 - Identifier declaration
 - UIL coding techniques for • 3–6
 - IMPORTED keyword • 3–17
 - Imported resource
 - defining in UIL • 3–16
 - Include directive
 - in UIL • 3–13
 - Include file • 2–2
 - of constants for UIL • 3–13
 - Indicator
 - in toggle button widget
 - customizing • 5–17
 - Inherit constants
 - definition • D–26
 - INIT GET SEGMENT routine • 5–24

Index

Initialization
 of DRM • 3–2
 of user interface • 3–2
 of widget instance • D–28
 of XUI Toolkit • 2–4, 3–2
INITIALIZE DRM routine • 3–34, 3–35
INITIALIZE routine • 2–4
Input focus
 dialog box widget grabbing • 7–12
 in text widgets • 9–1
 using with accelerators • 5–29
Input/output widgets • 1–8
INQUIRE NEXT PASTE COUNT routine • 13–2, 13–15
INQUIRE NEXT PASTE FORMAT routine • 13–2, 13–15
INQUIRE NEXT PASTE LENGTH routine • 13–2, 13–15
Inserting text
 in the text widgets • 9–6
Insertion point
 specifying position in text widgets • 9–11
 visibility in text widgets • 9–10
INSTALL ACCELERATORS routine • 5–28
INSTALL ALL ACCELERATORS routine • 5–28
Integer values
 defining in UIL • 3–18
Inter-client communication • 13–3
Inter-Client Communication Conventions Manual
 See ICCCM
Interface
 See User interface
Internationalization
 of text strings in labels • 5–19
 using UIL and DRM • 1–4, 3–49
 using UIL include files for • 3–65
INTERN ATOM routine • 13–19
Intrinsics
 commonly used routines • 2–31
 definition • 1–3
IS MANAGED routine • 2–22
IS REALIZED routine • 2–24
Item list
 canceling selections • 8–12
 creating • 8–3
 selecting items • 8–6

K

Key code

Key code (Cont.)
 conversion to key symbol • D–72
Key symbol
 in event specification • D–67

L

Label
 specifying as an icon in UIL • 3–29
LABEL CREATE routine • 5–2
Label gadget
 as menu item • 6–3
 attributes • 5–5
 creating • 5–2
 overview • 5–1
LABEL GADGET CREATE routine • 5–2
LABEL routine • 5–2
Label widget
 alignment • 5–4
 as menu item • 6–3
 attributes supported by gadget version • 5–5
 creating • 5–2
 customizing • 5–3
 margins • 5–4
 overview • 5–1
 sizing • 5–3
 text content • 5–5
 using attachment attributes with • 7–19
LATIN1 STRING routine • 5–21
LIST BOX ADD ITEM routine • 8–10
LIST BOX CREATE routine • 8–2
LIST BOX DELETE ITEM routine • 8–10
LIST BOX DELETE POSITION routine • 8–11
LIST BOX DESELECT ALL ITEMS routine • 8–12
LIST BOX DESELECT ITEM routine • 8–12
LIST BOX ITEM EXISTS routine • 8–11
LIST BOX routine • 8–2
LIST BOX SELECT ITEM routine • 8–11
LIST BOX SET HORIZ POS routine • 8–13
LIST BOX SET ITEM routine • 8–14
LIST BOX SET POS routine • 8–14
List box widget
 adding items to an item list • 8–9
 callbacks • 8–15
 canceling selections • 8–12
 comparing with menu widget • 8–1
 creating • 8–2
 creating item list • 8–3
 customizing • 8–12
 deleting items from an item list • 8–9, 8–10

List box widget (Cont.)

- margins • 8–14
 - overview • 8–1
 - selecting list items • 8–7
 - using support routines • 8–11
 - sizing • 8–12
 - specifying in UIL • 3–40
 - specifying visible items • 8–14
 - support routines
 - summary • 8–8
- LIST PENDING ITEMS routine • 13–1, 13–11
- Literals
 - retrieving from UID files • 3–38
- Literal values
 - obtaining from UID files • 3–38
- Local values
 - defining in UIL • 3–16
- Locking the clipboard • 13–3, 13–15
- Logical names
 - DECW\$INCLUDE • 3–14
 - for color names • 3–22
 - SY\$LIBRARY • 3–13
 - UIL\$INCLUDE • 3–14
- Low-level widget creation routines
 - compared to high-level routines • 1–3
 - standard arguments • 2–10
 - using to associate callbacks with a widget • 2–12
 - using to define initial appearance of a widget • 2–10
 - using to define parent/child relationship • 2–10

M

- MAIN LOOP routine • 2–25
- Main window
 - customizing • 14–17
- MAIN WINDOW CREATE routine • 4–4
- MAIN WINDOW routine • 4–4
- MAIN WINDOW SET AREAS routine • 4–6
- Main window widget
 - adding child widgets • 4–5
 - callbacks • 4–8
 - creating • 4–4
 - layout of children • 4–1
 - overview • 4–1
 - positioning • 4–7
 - sizing • 4–7
- MAKE GEOMETRY REQUEST routine • D–37, D–38
 - stack modes • D–39
 - traversing the application widget hierarchy • D–27
- MAKE RESIZE REQUEST routine • D–37
- MANAGE CHILDREN routine • 2–21, D–33
- MANAGE CHILD routine • 2–21, D–33
 - managing a single widget • 2–22
 - using with help widget • 12–8
- Managing widgets • 2–21
- Margins
 - in color mixing widget • 11–9
 - in compound string text widget • 9–9
 - in label widget • 5–4
 - in list box widget • 8–14
 - in menu widget • 6–10
 - in simple text widget • 9–9
- MENU BAR CREATE routine • 6–16
- MENU BAR routine • 6–16
- Menu bar widget • 6–1
 - creating • 6–15
 - customizing • 6–19
 - using with main window widget • 4–2
- MENU CREATE routine • 6–6
- Menu item
 - active • 6–3
 - alignment • 6–11
 - arrangement in a menu • 6–10
 - definition • 6–2
 - inactive • 6–3
 - radio button exclusivity • 6–11
 - restricting widget type • 6–11
- Menu packing
 - definition • 6–10
- MENU POPUP CREATE routine • 6–25
- MENU POSITION routine • 6–26
- MENU PULLDOWN CREATE routine • 6–13
- MENU routine • 6–6, 6–13, 6–25
- Menu widget
 - application widget hierarchy • 6–8
 - arrangement of menu items • 6–10
 - callbacks • 6–12
 - creating • 6–5
 - customizing • 6–9
 - maintaining menu history • 6–23
 - nesting menu widgets • 6–4
 - overview • 6–1
 - pop-up • 6–24
 - restricting child widgets • 6–11
 - specifying margins • 6–10
 - spring-loaded • 6–1
- Message box widget • 7–4
- MIT C binding
 - include file for DRM constants • 3–32
 - include file for UIL constants • 3–13

Index

Modal dialog box • 7–5
Modeless dialog box • 7–5
Module declaration
 in UIL • 3–12
Module header clauses
 in UIL • 3–12
MoveFrom operation • 13–16
MoveTo operation • 13–16
MOVE WIDGET routine • D–37, D–41
Multiline compound strings • 3–20

N

Names
 defining in UIL • 3–6
Nesting menu widgets • 6–4
NEXT EVENT routine • 2–26, 2–30

O

Object
 local definition of in UIL • 3–10
 modifying at run time • 3–40
Object arguments
 specifying in UIL • 3–25
Object declaration • 3–24
 UIL coding techniques for • 3–7
 using as a template • 3–45
OFFSET routine • D–51
OPEN DISPLAY routine • 2–5
OPEN HIERARCHY routine • 3–35, 3–36
OPTION MENU CREATE routine • 6–20
OPTION MENU routine • 6–20
Option menu widget • 6–1
 creating • 6–19
 customizing • 6–23
 overview • 6–19
 specifying default option • 6–23
 specifying label • 6–24
OVERRIDE TRANSLATIONS routine • 6–28, D–71
OWN SELECTION routine • 13–17

P

Page increment
 using scroll bar widget • 10–14

Parent/child relationship
 defining using UIL • 2–18
 defining with a high-level routine • 2–15
 defining with a low-level routine • 2–10
Parent/child widget relationship • 1–6
Parent widget
 definition • 1–6
PARSE ACCELERATOR TABLE routine • D–72
PARSE TRANSLATION TABLE routine • 6–27, D–71
Pascal
 Hello World! sample application • B–14
Paste function
 description • 13–4
 how to implement • 13–11
 in Edit menu • 13–3
Paste routines
 See Cut and paste routines
Pending delete
 in text widgets • 9–12
Pixmap
 as label • 5–5
 using in push button widget • 5–10
 using with toggle button widget • 5–17
Pixmap value
 specifying in UIL • 3–23
Pointer motion • D–45
Pop-up dialog box widget • 7–5
Pop-up menu widget • 6–1
 callbacks • 6–31
 creating • 6–24
 customizing • 6–31
 overview • 6–1
Positioning
 specifying the initial position of your application •
 14–19
Primitive widget • 1–6
PRIVATE keyword • 3–17
Private resource
 defining in UIL • 3–16
Procedure declaration
 in UIL • 3–15
 UIL coding techniques for • 3–6
PROCESS EVENT routine • 2–30
Property
 communicating with window manager using • 14–2
 DEC_WM_DECORATION GEOMETRY • 14–7
 DEC_WM_DECORATION_GEOMETRY • 14–5
 DEC_WM_HINTS • 14–5
 definition • 14–5
 defined by window manager vendors • 14–4
 setting values • 14–10

Property (Cont.)

- list of DECwindows window manager properties • 14-4
- predefined for communicating with window manager • 14-2
 - list of • 14-2
 - setting values • 14-8
- setting values of properties that are data structures • 14-9
- types of • 14-2
- WM_CLASS • 14-3
- WM_COMMAND • 14-3
- WM_HINTS • 14-3
- WM_ICON_NAME • 14-3
- WM_ICON_SIZE • 14-3
- WM_NAME • 14-3
- WM_NORMAL_HINTS • 14-3
- WM_TRANSIENT_FOR • 14-3
- WM_ZOOM_HINTS • 14-3

Prototypes

- developing using UIL • 3-59
- testing using UIL • 3-60

PULL DOWN MENU ENTRY CREATE routine • 6-13

Pull-down menu entry gadget

- as menu item • 6-3
- creating • 6-13
- definition • 6-3
- pulling callback • 6-15

PULL DOWN MENU ENTRY GADGET CREATE routine • 6-13

PULL DOWN MENU ENTRY routine • 6-13

Pull-down menu entry widget

- as menu item • 6-3
- creating • 6-13
- definition • 6-3
- pulling callback • 6-15

Pull-down menu widget • 6-1

- callbacks • 6-14
- creating • 6-12
- customizing • 6-14
- overview • 6-1
- using with an option menu widget • 6-20

Pull-right menu widget

- definition • 6-5

PUSH BUTTON CREATE routine • 5-7

Push button gadget

- as menu item • 6-3
- attributes • 5-10
- callbacks • 5-11
- creating • 5-7
- defining accelerators for • 5-27
- overview • 5-1

PUSH BUTTON GADGET CREATE routine • 5-7

PUSH BUTTON routine • 5-7

Push button widget

- as menu item • 6-3
- attributes supported by gadget version • 5-10
- callbacks • 5-11
- creating • 5-7
- customizing • 5-10
- defining accelerators for • 5-27
- highlighting behavior • 5-10
- overview • 5-1
- shadowing • 5-10
- simulating activation • 5-1
- specifying the insensitive pixmap • 5-10

Push-to-back button

- removing from title bar • 14-22

Q

QUERY GEOMETRY routine • D-42

QuickCopy function.

- callback routines • 13-18
- CopyFrom operation • 13-16
- CopyTo operation • 13-16
- differences among operations • 13-17
- getting the selection data • 13-19
- how to implement • 13-16
- KILL_SELECTION message • 13-16
- message types • 13-16
- MoveFrom operation • 13-16
- MoveTo operation • 13-16
- possible operations • 13-16
- sample • 13-17, 13-22
- selection threshold use • 13-17
- sending STUFF_SELECTION message • 13-18
- STUFF_SELECTION message • 13-16

R

Radio box widget • 6-1, 6-11

- creating the DECburger radio box widget • 5-14

Radio button exclusivity

- definition • 6-11

Realization

- of user interface • 2-24, 3-4

REALIZE WIDGET routine

- creating a window • D-30

Index

REALIZE WIDGET routine (Cont.)

traversing the application widget hierarchy • D-27

Reason

defining for user-defined widget in UIL • 3-66

definition • 1-8

in callback data structure • 2-27

REASON function • 3-28

RECOPY TO CLIPBOARD routine • 13-1

Red, Green, Blue color model

See RGB color model

Redo function

in Edit menu • 13-5

REGISTER CASE CONVERTOR routine • D-73

REGISTER CLASS routine • 3-3, 3-66, 3-71

REGISTER DRM NAMES routine • 3-3, 3-28, 3-37

Registration

of callback routines for DRM • 3-48

of names for DRM • 3-3, 3-28, 3-36

of user-defined classes for DRM • 3-71

REMOVE CALLBACKS routine • 2-31

REMOVE EVENT HANDLER routine • D-47

REMOVE WORK PROC routine • 2-30

Resize button

removing from title bar • 14-22

RESIZE WIDGET routine • D-37, D-41, D-43

RESIZE WINDOW routine • D-41

Resources

See Widget resources

RGB color model

definition • 11-2

RGB values

definition • 11-2

Routines

ACTIVATE WIDGET • 5-1

ADD ACTIONS • 6-28, D-61

ADD CALLBACKS • 2-31

ADD EVENT HANDLER • D-47

ADD FONT LIST • 5-25

ADD RAW EVENT HANDLER • D-47

ADD WORK PROC • 2-30

APPLICATION ADD CONVERTER • D-56

APPLICATION CREATE SHELL • 2-5, 2-7, 2-8

setting shell widget attributes • 14-11

APPLICATION MAIN LOOP • 2-26

ATTACHED DIALOG BOX • 7-13

ATTACHED DIALOG BOX CREATE • 7-13

ATTACHED DIALOG BOX POPUP CREATE •
7-13

AUGMENT TRANSLATIONS • D-71

BEGIN COPY TO CLIPBOARD • 13-2

BUILD EVENT MASK • D-48

Routines (Cont.)

CANCEL COPY FORMAT • 13-1

CANCEL COPY TO CLIPBOARD • 13-1

CLIPBOARD LOCK • 13-3, 13-15

CLIPBOARD REGISTER FORMAT • 13-2, 13-11

CLIPBOARD UNLOCK • 13-3, 13-15

CLOSE HIERARCHY • 3-34

COLOR MIX CREATE • 11-7

COLOR MIX GET NEW COLOR • 11-8

COLOR MIX SET NEW COLOR • 11-8

COMMAND APPEND • 4-16

COMMAND ERROR MESSAGE • 4-16, 4-17

COMMAND SET • 4-16

COMMAND WINDOW • 4-15

COMMAND WINDOW CREATE • 4-15

CONFIGURE WIDGET • D-37, D-41

CONVERT • D-56

CONVERT CASE • D-73

COPY FROM CLIPBOARD • 13-2

COPY TO CLIPBOARD • 13-1

CREATE APPLICATION CONTEXT • 2-5

CREATE FONT LIST • 5-25

CREATE MANAGED WIDGET • D-33

CREATE POPUP SHELL • 2-7

CREATE WIDGET • D-33, D-48

CS BYTE CMP • 5-23

CS CAT • 5-22

CS COPY • 5-23

CS EMPTY • 5-23

CS LEN • 5-23

CS STRING • 5-21

CS TEXT • 9-4

CS TEXT CLEAR SELECTION • 9-4

CS TEXT CREATE • 9-4

CS TEXT GET EDITABLE • 9-3

CS TEXT GET MAX LENGTH • 9-3, 9-8

CS TEXT GET SELECTION • 9-3, 9-12

CS TEXT GET STRING • 9-3, 9-7

CS TEXT REPLACE • 9-3, 9-6

CS TEXT SET EDITABLE • 9-3

CS TEXT SET MAX LENGTH • 9-3, 9-8

CS TEXT SET SELECTION • 9-4, 9-12

CS TEXT SET STRING • 9-3, 9-6

cut and paste • 13-1

DECTERM PORT • A-1

DESTROY WIDGET • 2-8, D-33

DIALOG BOX • 7-5

DIALOG BOX CREATE • 7-5

DIALOG BOX POPUP CREATE • 7-5

DIRECT CONVERT • D-56

DISPATCH EVENT • 2-26

DRM FREE RESOURCE CONTEXT • 3-34

Routines (Cont.)

DRM GET RESOURCE CONTEXT • 3-34
 DRM HGET INDEXED LITERAL • 3-34
 END COPY FROM CLIPBOARD • 13-2
 END COPY TO CLIPBOARD • 13-1
 FETCH COLOR LITERAL • 3-38
 FETCH ICON LITERAL • 3-38
 FETCH INTERFACE MODULE • 3-34
 FETCH LITERAL • 3-38
 FETCH NAME • 14-9
 FETCH SET VALUES • 3-40
 FETCH WIDGET • 2-20, 3-3, 3-37
 FETCH WIDGET OVERRIDE • 3-45
 for deferred fetching with DRM • 3-37
 FREE • 5-22
 GET APPLICATION RESOURCES • D-52
 GET CLASS HINT • 14-9
 GET ICON NAME • 14-9
 GET ICON SIZES • 14-9
 GET NEXT SEGMENT • 5-24
 GET NORMAL HINTS • 14-9
 GET RESOURCE LIST • D-51
 GET SUBVALUES • D-57
 GET TRANSIENT FOR HINT • 14-9
 GET WINDOW PROPERTY • 14-11
 GET WM HINTS • 14-9
 INIT GET SEGMENT • 5-24
 INITIALIZE • 2-4
 INITIALIZE DRM • 3-34, 3-35
 INQUIRE NEXT PASTE COUNT • 13-2, 13-15
 INQUIRE NEXT PASTE FORMAT • 13-2, 13-15
 INQUIRE NEXT PASTE LENGTH • 13-2, 13-15
 INSTALL ACCELERATORS • 5-28
 INSTALL ALL ACCELERATORS • 5-28
 IS MANAGED • 2-22
 IS REALIZED • 2-24
 LABEL • 5-2
 LABEL CREATE • 5-2
 LABEL GADGET CREATE • 5-2
 LATIN1 STRING • 5-21
 LIST BOX • 8-2
 LIST BOX ADD ITEM • 8-10
 LIST BOX CREATE • 8-2
 LIST BOX DELETE ITEM • 8-10
 LIST BOX DELETE POSITION • 8-11
 LIST BOX DESELECT ALL ITEMS • 8-12
 LIST BOX DESELECT ITEM • 8-12
 LIST BOX ITEM EXISTS • 8-11
 LIST BOX SELECT ITEM • 8-11
 LIST BOX SET HORIZ POS • 8-13
 LIST BOX SET ITEM • 8-14
 LIST BOX SET POS • 8-14

Routines (Cont.)

LIST PENDING ITEMS • 13-1, 13-11
 MAIN LOOP • 2-25
 MAIN WINDOW • 4-4
 MAIN WINDOW CREATE • 4-4
 MAIN WINDOW SET AREAS • 4-6
 MAKE GEOMETRY REQUEST • D-37, D-38
 MAKE RESIZE REQUEST • D-37
 MANAGE CHILD • 2-21, D-33
 MANAGE CHILDREN • 2-21, D-33
 MENU • 6-6, 6-12, 6-25
 MENU BAR • 6-16
 MENU BAR CREATE • 6-16
 MENU CREATE • 6-6
 MENU POPUP CREATE • 6-25
 MENU POSITION • 6-26
 MENU PULLDOWN CREATE • 6-12
 MOVE WIDGET • D-37, D-41
 NEXT EVENT • 2-26, 2-30
 OFFSET • D-51
 OPEN DISPLAY • 2-5
 OPEN HIERARCHY • 3-34, 3-36
 OPTION MENU • 6-20
 OPTION MENU CREATE • 6-20
 OVERRIDE TRANSLATIONS • 6-27, D-71
 PARSE ACCELERATOR TABLE • D-72
 PARSE TRANSLATION TABLE • 6-27, D-71
 PROCESS EVENT • 2-30
 PULL DOWN MENU ENTRY • 6-13
 PULL DOWN MENU ENTRY CREATE • 6-13
 PULL DOWN MENU ENTRY GADGET CREATE •
 6-13
 PUSH BUTTON • 5-7
 PUSH BUTTON CREATE • 5-7
 PUSH BUTTON GADGET CREATE • 5-7
 QUERY GEOMETRY • D-42
 RECOPY TO CLIPBOARD • 13-1
 REGISTER CASE CONVERTOR • D-73
 REGISTER CLASS • 3-3, 3-66, 3-71
 REGISTER DRM NAMES • 3-3, 3-28, 3-37
 REMOVE CALLBACKS • 2-31
 REMOVE EVENT HANDLER • D-47
 REMOVE WORK PROC • 2-30
 RESIZE WIDGET • D-37, D-41, D-43
 RESIZE WINDOW • D-41
 SCALE • 10-2
 SCALE CREATE • 10-2
 SCROLL BAR • 10-11
 SCROLL BAR CREATE • 10-11
 SCROLL WINDOW • 4-9
 SCROLL WINDOW CREATE • 4-9
 SCROLL WINDOW SET AREAS • 4-10

Index

Routines (Cont.)

SEPARATOR • 5-6
SEPARATOR CREATE • 5-6
SEPARATOR GADGET CREATE • 5-6
SET ARG • 2-11
SET CLASS HINT • 14-9
SET ICON NAME • 14-9
SET ICON SIZES • 14-9
SET KEY TRANSLATOR • D-73
SET MAPPED WHEN MANAGED • D-33
SET NORMAL HINTS • 14-9
SET TRANSIENT FOR HINT • 14-9
SET VALUES • 2-29, 2-31, 14-13
SET WM HINTS • 14-9
START COPY FROM CLIPBOARD • 13-2
START COPY TO CLIPBOARD • 13-1
S TEXT • 9-4
S TEXT CLEAR SELECTION • 9-4, 9-13
S TEXT CREATE • 9-4
S TEXT GET EDITABLE • 9-3, 9-7
S TEXT GET MAX LENGTH • 9-3, 9-8
S TEXT GET SELECTION • 9-3, 9-12
S TEXT GET STRING • 9-3, 9-7
S TEXT REPLACE • 9-3, 9-6
S TEXT SET EDITABLE • 9-3, 9-7
S TEXT SET MAX LENGTH • 9-3, 9-8
S TEXT SET SELECTION • 9-4, 9-12
S TEXT SET STRING • 9-3, 9-6
STORE NAME • 14-9
STRING • 5-21
STRING CONVERSION WARNING • D-56
STRING FREE CONTEXT • 5-24
STRING INIT CONTEXT • 5-24
TOGGLE BUTTON • 5-12
TOGGLE BUTTON CREATE • 5-12
TOGGLE BUTTON GADGET CREATE • 5-12
TOGGLE BUTTON GET STATE • 5-16
TOGGLE BUTTON SET STATE • 5-16
TOOLKIT INITIALIZE • 2-5
TRANSLATE KEYCODE • D-73
UNDO COPY TO CLIPBOARD • 13-1, 13-11
UNMANAGE CHILD • 2-21, D-33
UNMANAGE CHILDREN • 2-21, D-33

Rubber positioning

in attached dialog box widget • 7-22

S

Sample program

See DECBurger sample application

Sample program (Cont.)

See Hello World! sample application

SCALE CREATE routine • 10-2

SCALE routine • 10-2

Scale widget

adding labeled tick marks • 10-7

callbacks • 10-8

creating • 10-2

customizing • 10-4

displaying the value of • 10-6

overview • 10-1

sizing • 10-4

specifying the orientation • 10-5

specifying the range of values • 10-3

specifying the title • 10-5

SCROLL BAR CREATE routine • 10-11

SCROLL BAR routine • 10-11

Scroll bar widget

callbacks • 10-15

creating • 10-11

customizing • 10-14

defining stepping functions • 10-13

overview • 10-10

sizing the slider in • 10-13

specifying the range of values • 10-12

using with main window widget • 4-2

using with text widgets • 9-10

SCROLL WINDOW CREATE routine • 4-9

SCROLL WINDOW routine • 4-9

SCROLL WINDOW SET AREAS routine • 4-10

Scroll window widget

adding child widgets • 4-10

creating • 4-8

overview • 4-3

Select All function

in Edit menu • 13-5

Selecting text

canceling the selection in text widgets • 9-13

in text widgets • 9-12

retrieving the current selection in text widgets •
9-12

Selection

with cut and paste routines • 13-5

Selection threshold

default value • 13-17

using in QuickCopy • 13-17

Selection widget • 7-4

SEND EVENT routine • 13-19

SEPARATOR CREATE routine • 5-6

Separator gadget

as menu item • 6-3

- Separator gadget (Cont.)
 - creating • 5–6
 - overview • 5–1
- SEPARATOR GADGET CREATE routine • 5–6
- SEPARATOR routine • 5–6
- Separator widget
 - as menu item • 6–3
 - creating • 5–6
 - customizing • 5–7
 - overview • 5–1
- Server
 - opening connection to • 2–4
- SET ARG routine • 2–11
- SET CLASS HINT routine • 14–9
- SET ICON NAME routine • 14–9
- SET ICON SIZES routine • 14–9
- SET KEY TRANSLATOR routine • D–73
- SET MAPPED WHEN MANAGED routine • D–33
- SET NORMAL HINTS routine • 14–9
- SET TRANSIENT FOR HINT routine • 14–9
- SET VALUES routine • 2–29, 2–31, 12–8
 - using to assign values to shell widget attributes • 14–13
 - using with constraint widget • D–36
- SET WM HINTS routine • 14–9
- Shell widget
 - See Application shell widget
- Shrink-to-icon button
 - determining size limitations
 - removing from title bar • 14–22
 - specifying pixmap used in • 14–19
- Simple text widget
 - callbacks • 9–13
 - controlling resizing behavior • 9–10
 - creating • 9–4
 - customizing • 9–8
 - determining positions in • 9–6
 - disabling text editing • 9–7
 - half-border visibility • 9–11
 - including scroll bars in • 9–10
 - margins • 9–9
 - overview • 9–1
 - placing text in • 9–6
 - recommended way to specify size • 9–8
 - retrieving current value • 9–7
 - selecting text • 9–12
 - specifying insertion position • 9–11
 - support routines
 - advantages • 9–6
 - summary • 9–3
 - text cursor • 9–1
- Simple text widget
 - text cursor (Cont.)
 - specifying the blink rate • 9–10
 - using attachment attributes with • 7–19
 - using in a dialog box widget • 7–11
- Sizing
 - specifying the initial size of your application • 14–19
- Slider
 - specifying color of • 10–6
 - using with scale widget • 10–1
 - using with scroll bar widget • 10–10
- Standard dialog box widget
 - See also Dialog box widget • 7–4
- START COPY FROM CLIPBOARD routine • 13–2
- START COPY TO CLIPBOARD routine • 13–1
- S TEXT CLEAR SELECTION routine • 9–4, 9–13
- S TEXT CREATE routine • 9–4
- S TEXT GET EDITABLE routine • 9–3, 9–7
- S TEXT GET MAX LENGTH routine • 9–3, 9–8
- S TEXT GET SELECTION routine • 9–3, 9–12
- S TEXT GET STRING routine • 9–3, 9–7
- S TEXT REPLACE routine • 9–3, 9–6
- S TEXT routine • 9–4
- S TEXT SET EDITABLE routine • 9–3, 9–7
- S TEXT SET MAX LENGTH routine • 9–3, 9–8
- S TEXT SET SELECTION routine • 9–4, 9–12
- S TEXT SET STRING routine • 9–3, 9–6
- Sticky windows • 14–27
- STORE NAME routine • 14–9
- STRING CONVERSION WARNING routine • D–56
- STRING FREE CONTEXT routine • 5–24
- STRING INIT CONTEXT routine • 5–24
- String literals
 - See String values
- STRING routine • 5–21
- STRING TABLE function (UIL)
 - using with list box widget • 8–3
- String table values
 - defining in UIL • 3–20
- String values
 - defining in UIL • 3–18
- Subclass
 - definition • D–2
- Submenu
 - creating • 6–4
- Superclass
 - definition • D–2
 - inheritance constants • D–26
 - inheritance of operations • D–25
 - invocation of operations • D–27

Index

Superclass chaining
 definition • D-23
Symbol definition file • 2-2
Symbolic references
 to widget identifiers • 3-58

T

Tag
 in callback routine data structure • 2-12
 in UIL • 3-15
 standard argument to callback routine • 2-27
Terminal
 creating a DECterm • A-1
Text
 determining position in • 9-6
 displaying for editing • 9-6
 displaying read-only • 5-1
 extracting from a compound string • 5-23
 handling text selection • 9-12
 inserting text in the text widgets • 9-6
 retrieving text from the text widgets • 9-7
Text character
 input using text widgets • 9-1
Text cursor
 in text widgets • 9-1
 specifying appearance • 9-10
Text editing
 disabling • 9-7
 using text widgets • 9-1
Text path
 in a compound string text widget • 9-11
Text string
 conversion to compound string • 5-19
Tick marks
 creating a scale with • 10-7
 using with scale widget • 10-1
Time stamp
 obtaining from the X Event data structure • 13-8
 with text selection • 9-12
Title bar
 components • 14-22
 customizing • 14-19
 determining width and height • 14-23
 screen appearance under DECwindows window manager • 14-15
 specifying the application name • 2-4, 14-18
TOGGLE BUTTON CREATE routine • 5-12
Toggle button gadget
 as menu item • 6-3

Toggle button gadget (Cont.)
 callbacks • 5-18
 creating • 5-12
 customizing • 5-18
 defining accelerators for • 5-27
 overview • 5-1
 specifying initial state • 5-16
TOGGLE BUTTON GADGET CREATE routine • 5-12
TOGGLE BUTTON GET STATE routine • 5-16
TOGGLE BUTTON routine • 5-12
TOGGLE BUTTON SET STATE routine • 5-16
Toggle button widget
 as menu item • 6-3
 callbacks • 5-18
 creating • 5-12
 customizing • 5-17
 defining accelerators for • 5-27
 overview • 5-1
 specifying initial state • 5-16
 specifying pixmap • 5-17
 specifying shape of indicator • 5-17
 support routines • 5-16
TOOLKIT INITIALIZE routine • 2-5
TRANSLATE KEYCODE routine • D-73
Translation management • D-60
Translation table • D-62
 definition • 6-27
 management • D-71
 modifier names • D-63
 syntax • D-63

U

UID file
 accessing at run time • 3-35
 color names in • 3-22
 compared with object module • 3-28
 definition • 3-1, 3-10
 name of UIL module in • 3-12
 resolving symbolic references in • 3-36
 retrieving literal values from • 3-38
 string literals in • 3-6
 widget definition in • 3-24
UID hierarchy
 declaring in application program • 3-36
 definition • 3-3
 uses for • 3-49
UIL (User Interface Language)
 ARGUMENT function • 3-66
 ARGUMENTS keyword • 3-25

UIL (User Interface Language) (Cont.)

- arguments list • 3-25
- attached dialog box widget • 3-17
- CALLBACKS keyword • 3-27
- callbacks list • 3-27
- coding techniques • 3-5
- CONTROLS keyword • 3-26
- controls list • 3-26
- creating module • 3-10
- defining the widgets in an interface • 2-17
- definition • 1-4
- EXPORTED keyword • 3-16
- ICON function • 3-23
- identifier declaration in • 3-6
- identifiers • 3-57
- IMPORTED keyword • 3-16
- include directive • 3-13
- include file for constants • 3-13
- keywords • 3-4
- list of value types • 3-16
- modifying objects at run time • 3-40
- module declaration • 3-12
- module header clauses • 3-12
- names • 3-6
- object declaration in • 3-7, 3-24
- PRIVATE keyword • 3-16
- procedure declaration • 3-15
- procedure declaration in • 3-6
- REASON function • 3-28, 3-66
- referencing user-defined widget in • 3-68
- registering identifiers for DRM • 3-36
- scope of references in module • 3-16
- setting up for deferred fetching • 3-37
- specifying an icon as a label in • 3-29
- specifying an interface for international markets • 3-49
- specifying callbacks in • 3-27
- specifying children in • 3-26
- specifying color values • 3-22
- specifying font values • 3-21
- specifying integer values • 3-18
- specifying object arguments in • 3-25
- specifying pixmap values • 3-23
- specifying string table values • 3-20
- specifying string values • 3-18
- specifying widget attributes in • 3-25
- specifying widget hierarchy in • 3-26
- structure of module • 3-11
- symbolic references to widget identifiers • 3-58
- using help widget • 3-46
- using identifiers in template • 3-45

UIL (User Interface Language) (Cont.)

- using multiple UIL modules • 3-63
- using object declaration as a template in • 3-45
- using to associate callback routines with a widget • 2-19
- using to define the initial appearance of widgets in an interface • 2-18
- using to define the parent/child relationship • 2-18
- using to develop and test prototypes • 3-59
- value declaration in • 3-6, 3-16
- widget declaration in • 3-7, 3-24
- UIL command • 3-10
- UIL compiler
 - data type checking rules • 3-15
 - invoking • 3-10
- UIL functions
 - COLOR • 3-22
 - COLOR_TABLE • 3-23
 - COMPOUND_STRING • 3-6, 3-19
 - ICON • 3-23
 - STRING TABLE • 8-3
 - XBITMAPFILE • 3-23
- UIL include file
 - for constants • 3-13
- UIL keywords • 3-4
- UIL names • 3-6
- UIL specification file
 - See UIL
- UIL value
 - global • 3-16
 - list of types • 3-16
 - local • 3-16
 - retrieving from UID files • 3-38
- UNDO COPY TO CLIPBOARD routine • 13-1, 13-11
- Undo function
 - in Edit menu • 13-5
- Unit increment
 - using scroll bar widget • 10-13
- UNMANAGE CHILDREN routine • 2-21, D-33
- UNMANAGE CHILD routine • 2-21, D-33
- User-defined widget
 - declaring creation routine for in UIL • 3-16
 - defining in UIL • 3-65
 - referencing in UIL • 3-68
- User interface • 1-4
 - creating form of • 1-6
 - creating using high-level routines • 2-14
 - creating using low-level routines • 2-9
 - creating using UIL and DRM • 3-3
 - designing for international markets • 3-49
 - initializing • 3-2

Index

User interface (Cont.)

- integrating application function with • 1–7
- manipulating at run time • 2–30
- realization of • 3–4
- set up • 2–1
- specifying using UIL • 3–4

User Interface Language

See UIL

User interface object

- declaring in UIL • 3–7, 3–24
- definition • 1–2

V

Value declaration

- in UIL • 3–16
- scope of reference to • 3–16
- UIL coding techniques for • 3–6

VAX binding

- examples • B–1
- include file for DRM constants • 3–32
- include file for UIL constants • 3–13

Vendor.h

- vendor-specific window manager attribute definition file • 14–13

VMS help library enhancements • 12–4

VMS Librarian Utility extensions • 12–5

W

Widget attributes • 1–2, 1–14

- assigning values to • 1–15
- specifying in UIL • 3–25
- specifying using high-level widget creation routines • 2–15
- specifying using low-level widget creation routines • 2–10

Widget creation routines • 1–3

Widget declaration • 3–24

- UIL coding techniques for • 3–7

WIDGET GEOMETRY structure • D–38

- request modes • D–38

Widget hierarchy

- See also Application widget hierarchy
- specifying in UIL • 3–26

Widget resource data structure • D–48

Widget resources • D–48

- assigning values to • D–57

Widget resources (Cont.)

- classes • D–48
- converters • D–52
- lists • D–48
- resource converters
 - predefined • D–53
- resource convertors
 - creating • D–54
- types of • D–49

Widgets

See also Gadgets

See also Object

- appearance attributes • 1–15
- application shell • 1–6, 2–6
- application widget hierarchy • 1–6
- as menu items • 6–2
- attached dialog box • 7–2
- attributes • 1–14
 - assigning values to • 1–15
- building • D–1
 - sample • D–2
 - using UIL • 3–65
- callback attributes • 1–15
- caution box • 7–4
- choice • 1–9
- class • D–10
- class initialization • D–24
- class part structure
 - initialization • D–20
- class relationship • D–1
- clipping • 1–6
- color mixing • 11–1
- command window • 4–15
- common attributes • 1–14
- composite • 1–6
- compound string text • 9–1
- container • 1–8
- core • D–1
- creating • 2–8
 - summary • 2–8
- creating a window
 - realize procedure • D–30
- creating subclasses of • D–16
- creation routines • 1–3
- customizing • 1–2
- declaring creation routine for user-defined • 3–16
- definition • 1–2
- destroy procedure • D–31
- dialog box • 7–2
- file selection • 7–4
- inheriting superclass operations • D–25

Widgets (Cont.)

- input/output • 1–8
 - instance initialization • D–28
 - invocation of superclass operations • D–27
 - label • 5–1
 - list box • 8–1
 - list of • 1–9
 - local definition of in UIL • 3–10
 - main window • 4–1
 - managing • 2–21
 - manipulating at run time • 2–31
 - menu bar • 6–1
 - message box • 7–4
 - naming conventions • D–17
 - option menu • 6–1
 - parent/child relationship • 1–6
 - pop-up menu • 6–1
 - position attributes • 1–14
 - primitive • 1–6
 - pull-down menu • 6–1
 - push button • 5–1
 - query geometry procedure • D–42
 - radio box • 6–1
 - realizing • 2–24
 - scale • 10–1
 - scroll bar • 10–10
 - scroll window • 4–8
 - selection • 7–4
 - separator • 5–1
 - simple text • 9–1
 - size attributes • 1–14
 - size changes
 - initiation • D–37
 - subclass
 - definition • D–2
 - superclass
 - definition • D–2
 - superclass chaining • D–23
 - support routines • 2–32
 - toggle button • 5–1
 - user-defined in UIL • 3–65
 - visibility • D–47
 - window • 4–11
 - work area menu • 6–1
 - work in progress box • 7–4
- Window background color • 3–22
- WINDOW CREATE routine • 4–11
- Window foreground color • 3–22
- Window manager
- See also DECwindows window manager bypassing • 14–27

Window manager (Cont.)

- communicating limitations • 14–16
 - communicating with • 14–1
 - communicating with using vendor-specific properties • 14–10
 - DECwindows window manager extensions • 14–4
 - determining font used by • 14–22
 - list of common programming tasks • 14–15
 - overview • 14–1
 - providing hints to • 14–1
 - relation to application • 2–6
 - screen appearance • 14–15
 - routines for setting properties • 14–8
 - using properties to communicate with • 14–2
 - using shell widget attributes to communicate with • 14–8
 - using widget attributes to communicate with • 14–11
 - vendor extensions • 14–4
- Window manager hints
- definition • 14–1
- WINDOW routine • 4–11
- Window selection
- See also Selecting text in text widgets • 9–12
- Window widget
- callbacks • 4–14
 - creating • 4–11
- WM Decoration Geometry data structure
- definition • 14–7
- WM_CLASS property • 14–3
- WM_COMMAND property • 14–3
- WM_HINTS property • 14–3
- definition • 14–3
- WM_ICON_NAME property • 14–3
- WM_ICON_SIZE property • 14–3
- WM_NAME property • 14–3
- WM_NORMAL_HINTS property • 14–3
- WM_TRANSIENT_FOR property • 14–3
- WM_ZOOM_HINTS property • 14–3
- Work area
- of main window widget • 4–3
- Work area menu widget • 6–1
- callbacks • 6–12
 - creating • 6–5
 - customizing • 6–9
 - margins • 6–10
 - overview • 6–1
 - sizing • 6–9
- Work-in-progress box widget • 7–4
- Work procedure
- creating • 2–30

Index

Work procedure (Cont.)

definition • 2–30

registering • 2–30

Writing direction

specifying in a compound string • 5–20

text cursor as indicator • 9–12

X

XA_PRIMARY atom

using in QuickCopy function • 13–16

XA_SECONDARY atom

using in QuickCopy function • 13–16

XUI Resource Manager

See DRM

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	_____	Local Digital subsidiary or approved distributor
Internal ¹	_____	USASSB Order Processing - WMO/E15 <i>or</i> U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

¹For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS DECwindows Guide to
Application Programming
AA-MG21B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I would like to see more/less _____

What I like best about this manual is _____

What I like least about this manual is _____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

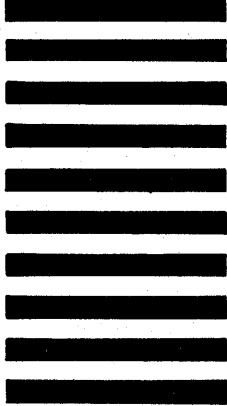
_____ Phone _____

--- Do Not Tear - Fold Here and Tape ---

digitalTM



No Postage
Necessary
if Mailed
in the
United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987



--- Do Not Tear - Fold Here ---

Cut Along Dotted Line