SPARCworks/Visual User's Guide

Version 1.1

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A

Part No.: 802-3528-10
Revision A, November 1995

© 1995 Sun Microsystems, Inc. All rights reserved. Manufactured in the United States of America.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

Copyright ©1992, 1993, 1994, 1995 by Imperial Software Technology Limited. All rights reserved. This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or documentation may be reproduced by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® system and from the Berkeley 4.3 BSD system, licensed from the University of California. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the 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 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS: Sun, Sun Microsystems, the Sun logo, Sun Microsystems Computer Corporation, and Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and may be protected as trademarks in other countries.UNIX is a registered trademark in the United States and other countries, exclusively licensed through X / Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product, service, or company names mentioned herein are claimed as trademark names by their respective companies.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. in the United States and may be protected as trademarks in other countries. SPARCcenter, SPARCcluster, SPARCompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, and UltraSPARC are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK™ and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

OSF / Motif and Motif are trademarks of Open Software Foundation, Inc. X Window System is a product of the Massachusetts Institute of Technology. X WindowSystem is a trademark of XConsortium, Inc.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND / OR CHANGES IN THE PRODUCT(S) AND / OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Please
Recycle

Contents

Overview 1

1.1 Introduction to SPARCworks/Visual

SPARCworks/Visual is an interactive tool for building graphical user interfaces (GUIs) using the widgets of the standard OSF/Motif toolkit as building blocks. SPARCworks/Visual lets you design a hierarchy of widgets on the screen quickly and easily by clicking on icons. It displays your design in two ways simultaneously: as a tree structure which represents the widget hierarchy and as a dynamic display—an active prototype which shows what your interface looks like and how it behaves. Interactive editing features let you browse through the hierarchy, cut and paste and set attributes of individual widgets. Because the dynamic display changes as you edit your widget hierarchy, you can immediately see the effects of your actions.

When your design is complete, SPARCworks/Visual automatically generates the code files required for your interface. You can compile, link and run the code generated by SPARCworks/Visual without modification as a prototype of your interface. You connect the prototype interface to your application code by writing connecting code. SPARCworks/Visual provides sample files which you can use as templates for the connecting code.

One way you can connect your SPARCworks/Visual interface to the application is by associating callback functions with specific widgets. For example, you can designate a certain routine to be called whenever the user clicks a certain pushbutton in the interface. Callbacks let your application receive and handle user events from the interface.

SPARCworks/Visual can also be extended to use widgets from other X toolkits. The *User-Defined Widgets* chapter discusses how to extend SPARCworks/Visual to include additional widgets.

SPARCworks/Visual typically generates code for use with the standard OSF/Motif and X toolkits. However, it can also generate code that will result in an equivalent interface on Microsoft Windows. The code can be structured in such a way that the majority of your application will remain the same on either platform. This technique makes extensive use of SPARCworks/Visual's C++ code generation facilities and is discussed in the *Cross Platform Development* and *Cross Platform Tutorial* chapters.

1.2 Basic Concepts and Terms

The following introduces some of the basic concepts of SPARCworks/Visual together with some of the terms used in this manual.

1.2.1 Widgets

Widgets are the building blocks used to create a user interface. Some widgets have a specific appearance and behavior in the interface display. Examples in Motif include PushButton, Label, and TextField widgets. Another type of widget is invisible itself but serves to contain and organize other widgets and is thus known as a *container widget*. Container widgets include the Form, BulletinBoard, MenuBar and RowColumn widgets.

All the Motif widgets, and any additional widgets which your configuration of SPARCworks/Visual uses, are represented by icons in a *widget palette* on the left side of the main SPARCworks/Visual screen. When you click on one of these icons, a widget of that type is added to the design. Individual widgets in the design are known as *instances* of a *widget class*. For example, if you click on the PushButton icon three times, you add three instances of the widget class PushButton to your design.

1.2.2 Design Hierarchy

The widgets in a design are organized in a *design hierarchy* which SPARCworks/Visual displays as a tree which has its root at the top and branches spreading downward. The design hierarchy is displayed in the large drawing area of the main SPARCworks/Visual screen, as shown in Figure 1-1. This area is called the *construction area.*

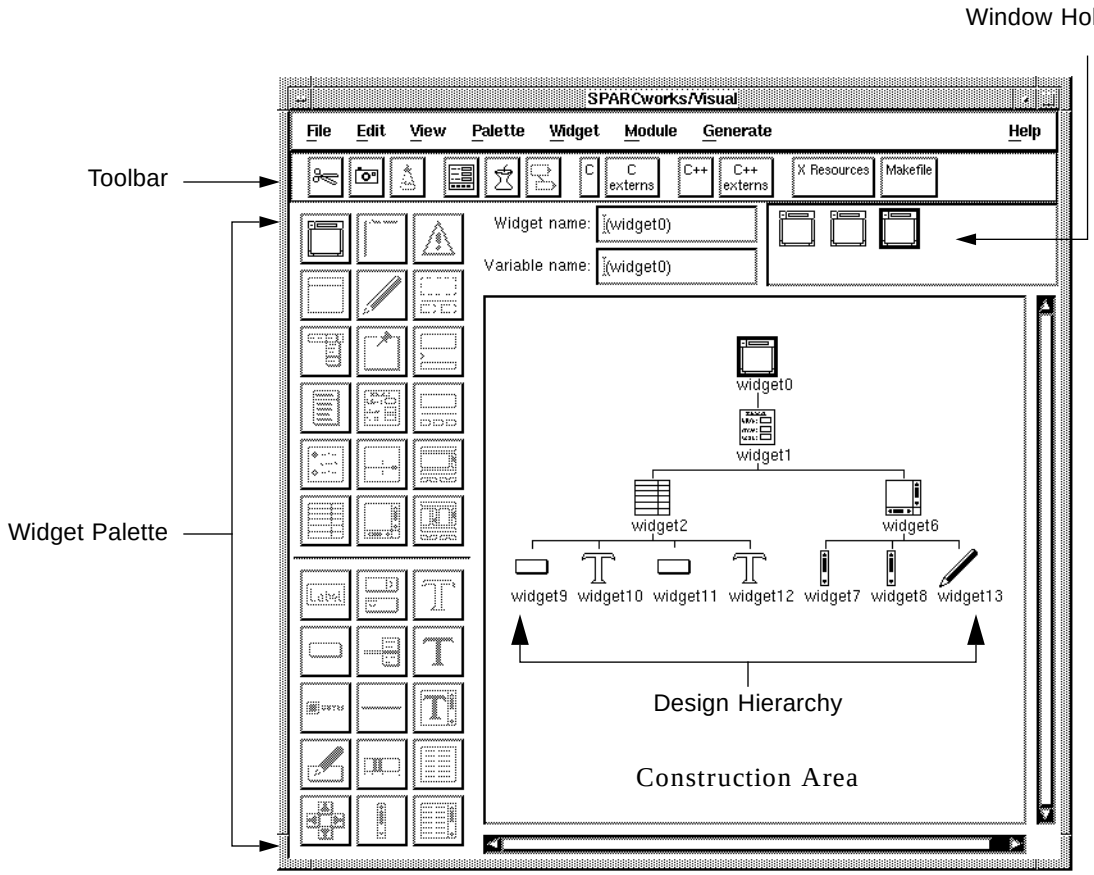


Figure 1-1 The Main SPARCworks/Visual Screen

Widgets added to the hierarchy are *children* of the *parent* widget immediately above them. This relationship is important because a parent *widget* can affect its children's appearance and behavior. For example, a RowColumn widget can impose a strict layout on its children which causes the children's size and position to change automatically.

Parent widgets appear above their children on the screen, as shown in Figure 1-2.

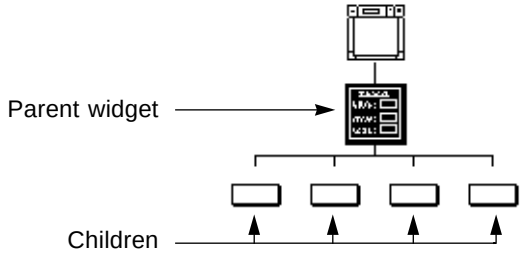


Figure 1-2 A Design Hierarchy

1.2.3 Resources

Resources are attributes of a widget which affect its appearance or behavior. Examples include dimensions, the content of a label or text field, and color. When SPARCworks/Visual creates an instance of a widget, it assigns valid default values for each resource. Interactive *resource panels* allow you to supply your own resource values.

Since resources and their valid values are defined by the widget manufacturer, and not by SPARCworks/Visual, it is not possible to document all of them fully in the SPARCworks/Visual manual. However, to aid you in learning SPARCworks/Visual, we discuss some commonly used Motif resources in this book. As you become more experienced, you should consult the Motif documentation for complete information about its widget set and the possible resource settings for each widget. If you are using widgets from other toolkits, consult the documentation from the widget developer for guidelines.

One group of resources (*attachments*) controls the spatial position of widgets within the Form container widget. SPARCworks/Visual provides an interactive editor, the *Layout Editor*, for setting attachments.

1.2.4 Gadgets

Some classes of widgets have counterparts called *gadgets*. Gadgets are like widgets but have a more restricted set of resources. Because gadgets are less expensive than widgets in terms of machine resources, they are sometimes preferred.

1.3 Protection from Invalid Actions

SPARCworks/Visual has many features designed to protect you from specifying widget hierarchies or resource combinations which are not valid in Motif. Commands that cannot be executed in the current circumstances, resources that do not apply to a particular widget and the icons of widgets that are not valid children of a proposed parent widget are *grayed out* on the screen, as shown in Figure 1-3. Grayed-out commands, resources, and icons have no effect if selected.

Figure 1-3 Active and Inactive CascadeButton Icons

SPARCworks/Visual also rejects invalid resource settings. An entry on a resource panel may be rejected for two reasons: either the value entered is outside the valid range, or you are trying to change a resource which is controlled or limited by the widget's parent. This subject is discussed in the *Using the Resource Panels* chapter. Motif's rules for resource settings are complex and invalid settings can have serious consequences. This feature of SPARCworks/Visual ensures that your resource settings are valid.

1.4 On-Line Help

On-line help is available throughout SPARCworks/Visual. For general help, pull down the Help Menu in the main window and select the "Help" option. Most dialog boxes and resource panels also have a "Help" button which you can click on for specific help about that dialog box.

Most help screens in SPARCworks/Visual provide links to related topics. A list of related topics is displayed with the help message. You can click on one of these topics and then press the "Follow Link" button to see the related help screen.

1.4.1 Palette Icons Help

The "Palette Icons" option in the Help Menu displays a copy of the widget palette with the name of each Motif widget icon. Clicking on any of the icons on this screen displays a description of the widget class of which the selected widget is a member. A list of the icons and their names is also available on the SPARCworks/Visual Quick Reference Card.

1.5 The SPARCworks/Visual Development Cycle

The process of creating SPARCworks/Visual applications usually involves the following four stages:

Designing the interface. This stage includes the following operations:

- Building the widget hierarchy
- Setting resources
- Using the Layout Editor to adjust the layout
- Designating callbacks to be associated with individual widgets

Generating code. SPARCworks/Visual automatically generates all the C or C++ code needed to display and operate your interface. SPARCworks/Visual can also generate a *stubs file* containing all the *#include* statements and function declarations necessary to connect the interface code to your application code.

Writing code. To connect your interface prototype to a real application, supply the necessary code between the empty function brackets in the stubs file.

Linking, running and testing. This phase follows the debugging cycle needed for developing any software program.

1.6 How This Manual Is Organized

This manual is organized in three main sections:

- Tutorial
- Power Use
- Reference

1.6.1 The Tutorial

Because SPARCworks/Visual is highly interactive, it is easier to learn its features by actually going through the steps for a simple layout than by reading descriptions of the various features. If you are new to SPARCworks/Visual, we recommend reading the tutorial chapters at your workstation and performing the steps given to build a simple interface. The tutorial gives detailed instructions for all stages of the development cycle: building the design hierarchy, setting resources, adjusting the layout, setting callbacks, generating code and writing a very simple callback routine. At the end of the tutorial you will have a fully operational (if rudimentary) interface and you will have used all the major features of SPARCworks/Visual.

Knowledge of the X window system and Motif is valuable at all stages of learning SPARCworks/Visual. However, if you are new to X or Motif, you can profit from the first several chapters of the SPARCworks/Visual tutorial while studying the documentation for X and Motif simultaneously. The bibliography in this manual provides a list of recommended books on X and Motif. At the code generation stage you must have some understanding of X and Motif, and one of the programming languages used by SPARCworks/Visual (C, C++, or UIL).

1.6.2 The Power Use Section

The power use section discusses advanced techniques for getting the most from SPARCworks/Visual. It is intended for users who are familiar with Motif and X and who have prior experience with SPARCworks/Visual or have finished working through the tutorial. It contains sections dealing with the advanced code generation capabilities, Makefile generation, configuring the widget palette and toolbar, integrating user-defined widgets, advanced layout and internationalization. There are also short tutorial chapters illustrating the structured code generation and cross platform development capabilities.

1.6.3 The Reference Section

The reference section is intended for SPARCworks/Visual users at all levels. It includes:

- Summaries of all the SPARCworks/Visual commands
- A summary of the Motif widgets and available resources, and some information on how they are mapped to Microsoft Windows code
- Suggestions for troubleshooting
- A description of the resources which can be set to alter SPARCworks/Visual's behavior and appearance
- Some details on the integration of SPARCworks/Visual with third party products
- A glossary

1.6.4 Conventions Used in this Manual

1. New terms occur in *italics* the first time they appear. These terms are defined in the Glossary.

2. The names of keyboard keys and mouse buttons appear in italics, enclosed by angle brackets: <*Tab*>. When two keys must be pressed simultaneously, we use this form:

   <*1st key-2nd key*>

   For example: <*Ctrl-C*>, <*Meta-H*>, <*Shift-button 1*>

3. Text to be typed at the keyboard is shown in this format:

```
type this exactly
```

without quotation marks. If quotation marks appear, they are to be entered with the text.

4. Some menu commands have keyboard accelerators—keystrokes which can be used to execute the command without using the mouse. In these instructions, we mention keyboard accelerators in parentheses after naming the menu command. The following instruction:

   Pull down the Widget Menu and select "Reset" (<Ctrl-T>).

   means to select "Reset" from the menu or press *<Ctrl-T>*—but not both.

5. File names, function names, and variable names are all given in italics. Function names are distinguished by empty parentheses after the function name: *XtAppMainLoop()*. Angle brackets indicate a variable portion of a name:

   The default widget name is in the form widget<n>, where n is a numeral.

6. "Click" always means to use mouse button 1 unless otherwise noted. Unless you have reconfigured your mouse, button 1 is usually the left button, button 2 the middle button and button 3 the right button.

   The instruction to "click twice" is different from "double-click". "Double-click" means that you must press the mouse button twice in rapid succession. "Click twice" can be done at any speed.

7. Names of Motif widget classes are capitalized: Label, PushButton. When similar words are used in an ordinary sense, they are not capitalized:

   The main window of this design does not use a MainWindow widget but a Form widget.

8. Books mentioned in the text of this manual are listed in Appendix E.

1

Building the Widget Hierarchy 2

2.1 Introduction to the Tutorial

The tutorial section of this document includes the following chapters:

- Building the Widget Hierarchy
- Using the Resource Panels
- The Layout Editor
- Other Editors
- Additional Windows and Links
- Search
- Generating Code

The tutorial is meant to be read at your workstation while you follow the step-by-step instructions to build a simple interface. In the process, you will be introduced to all of the major features of SPARCworks/Visual.

When completed, the tutorial interface looks like Figure 2-1:

2

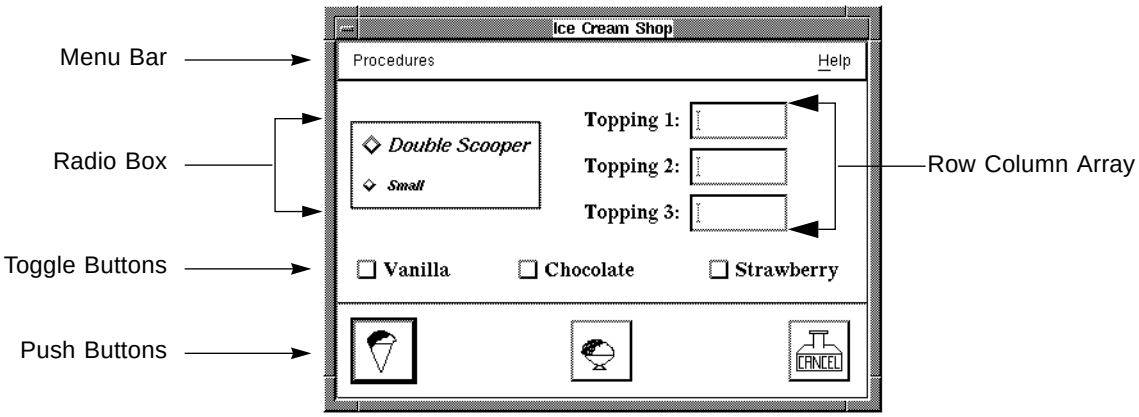


Figure 2-1 Tutorial Interface

Most of this tutorial addresses the Motif-only style of development, except where reference to a more general technique makes the inclusion of cross platform information relevant. There are further specialized tutorials on cross platform development and structured code generation in the *Power Use* section.

2.2 The Design Hierarchy

The first steps in building an interface are deciding which widgets should be used to build it and then developing an appropriate design hierarchy on the SPARCworks/Visual screen. These steps are explained in this chapter. In the process, you will learn how to:

- Start the program and begin a new design
- Build a design hierarchy for the five common combinations of widgets shown in Figure 2-1
  - Menu bar
  - Radio box
  - Row column array
  - Group of toggle buttons
  - Group of pushbuttons
- Assign names to widgets
- Save and retrieve SPARCworks/Visual files

- Edit the structure of a design hierarchy using cut, paste, copy and on-screen dragging facilities

2.3 Starting and Stopping

You should have SPARCworks/Visual properly installed on your system before you begin the tutorial. Consult the installation instructions or your system administrator if SPARCworks/Visual is not yet installed, or if the commands below do not bring up the main SPARCworks/Visual screen.

Use one of the following commands to start SPARCworks/Visual from within X. Either command brings up the main SPARCworks/Visual screen.

♦ **Type**:

`visu`

or

`small_visu` (for VGA or other small screen displays)

If you invoke SPARCworks/Visual with small_visu, the icons are smaller and slightly different from those shown in this document.

2.3.1 Command-Line Options

SPARCworks/Visual has numerous command-line options. These are documented in the *Command Line Operations* chapter of this manual; you can also get a listing of them when you invoke SPARCworks/Visual with the `-x` option. For example, to resume work on a previously saved design, you can specify the filename on the command line: visu `<filename>`. SPARCworks/Visual also accepts all the standard X toolkit options.

To start SPARCworks/Visual in *Windows mode*, which enables the cross platform development capabilities, use the command line switch `-windows`.

The File Menu of SPARCworks/Visual provides commands to let you save your work, exit the program and come back to your design later. We introduce these commands here so you can use them at any time during the tutorial.

2.3.2 Save, Save As

You can save your design by selecting “Save as...” (<*Ctrl-V*>) from the File Menu. “Save as...” displays a file browser, described later in this chapter, which lets you specify a filename for your design.

If you have already specified a filename, you can simply select “Save” (<*Ctrl-A*>). This procedure is faster since you are not prompted for a filename. By convention, names for SPARCworks/Visual design files have the suffix `.xd`.

2.3.3 Open, New, Exit

To load a saved design file into SPARCworks/Visual, use the “Open” command (<*Ctrl-O*>). “Open” displays a file browser, described later in this chapter, which lets you select an existing design file. “New” (<*Ctrl-N*>) clears the construction area and starts over with an empty design. “Exit” (<*Ctrl-E*>) terminates the program. All three of these commands discard any changes you have made to the design. If you have changes in your design that have not yet been saved, SPARCworks/Visual asks if you want to save before it executes any of these commands.

2.4 Navigating in the Menus

You can select commands from the SPARCworks/Visual menus in three ways:

- Clicking with the mouse
- Using keyboard *accelerators*
- Using keyboard *mnemonics*

2.4.1 Accelerators

A keyboard accelerator is a keystroke which is designated to execute a menu command. For example, you can press <*Ctrl-A*> to execute the “Save” command.

Accelerators work whenever the input focus is in any region of the SPARCworks/Visual screen. Accelerators are printed in parentheses wherever this tutorial instructs you to execute a command. They also appear opposite the command name in the pulldown menu on the screen.

2.4.2 Mnemonics

The underscored characters in menu names and options are mnemonics, a way of navigating in the menus without using the mouse. To pull down a menu by using its mnemonic character, press that character while holding down the <*Meta*> key. For example, you can pull down the File Menu by pressing <*Meta F*>. After pulling down the menu, you can select any item by pressing its mnemonic character without <*Meta*>. The complete sequence for calling "Save" using mnemonics is: <*Meta F*>, <*s*>.

In the *Using the Resource Panels* chapter, you will learn how to set up accelerators and mnemonics on the menu bars you create in SPARCworks/Visual.

2.5 Toolbar

The SPARCworks/Visual interface includes a toolbar which can be configured by the user to contain buttons corresponding to menu items. The default toolbar layout is shown in Figure 2-2.

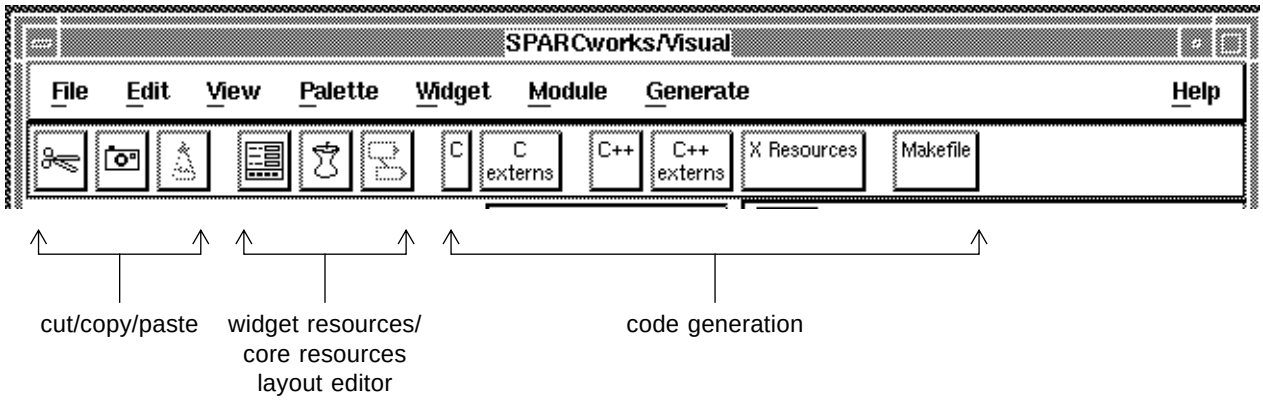


Figure 2-2 Default Toolbar Layout

Generally, when you select a toolbar button, it does exactly the same thing as the corresponding menu button. The code generation buttons are an exception. When you generate code from the toolbar, the file is generated immediately using your last specifications for that file. The Generate Dialog is only shown the first time you generate that file. When you generate from a menu bar command, SPARCworks/Visual always displays the dialog first.

2.5.1 Windows mode specific elements

When SPARCworks/Visual is in Windows mode the default toolbar configuration contains two additional elements: the *flavor* option menu and the *Windows compliant* toggle button. Briefly the flavor option menu is used to indicate which target code will be generated, e.g. Motif, MFC. This is useful when using the toolbar code generation buttons. The Windows compliant toggle indicates whether the current design is Windows compliant. That is, whether SPARCworks/Visual could generate valid Windows code for it. The toggle button is also used to invoke the compliance checking process. These features are discussed more fully in the *Cross Platform Development* chapter and tutorial.

2.6 Starting the Design

All dialogs start with a Shell widget. The Shell icon, shown in Figure 2-3, is in the upper left corner of the widget palette. Whenever you start a new design, all icons except the Shell are disabled.

Figure 2-3 Shell icon

♦ **Click on the Shell icon.**

A copy of the Shell icon appears in the construction area.

2.7 Naming Widgets

When an instance of a widget is added to the hierarchy, it is assigned two names by SPARCworks/Visual: a *widget name* and a *variable name*. The variable name is the name by which the widget is referenced in the code. The widget name is the name used by the toolkits to assign resources. By default these names are identical, but need not be, and are of the form *widget*<*n*>, where *n* is an integer assigned by SPARCworks/Visual. However, because several features of SPARCworks/Visual require explicit variable names, it is a good habit to assign explicit variable names to the most important widgets as you add them.

To name the Shell widget:

1. Double-click in the box opposite "Variable Name" at the top of the screen.

When you double-click in the box, all text in it is highlighted. Entering new text replaces the highlighted text.

2. Type: `shell_1`

The name is automatically assigned to the widget when you create and select another widget. You may also assign the name by pressing <*Return*> in the variable name box.

The variable name must be unique because it is used to refer to the widget structure for that instance of a widget when SPARCworks/Visual generates code. SPARCworks/Visual does not let you enter a variable name used elsewhere in your design. To avoid problems in compiling, never use the names of your application functions or variables, Motif or X defines or routine names, or C or C++ reserved words as widget variable names.

When you change the variable name, SPARCworks/Visual automatically assigns the same name to the widget name unless you also explicitly specify a widget name in the "Widget Name" box. The widget name does not have to be unique. Widgets with the same widget name can be configured to share resource settings. It is often convenient to group widgets by a common widget name so that end users can reset their resources with a single operation. This subject is discussed more extensively in the *Generating Code* chapter.

2.8 Adding Children to the Hierarchy

The Shell widget can have any kind of widget as its child; it can, however, only have one child. Therefore, you should choose a widget which can contain the rest of your layout. A MainWindow, BulletinBoard, Form, or DialogTemplate are commonly used. However, the DialogTemplate provides a convenient layout for the tutorial interface.

♦ **Click on the DialogTemplate icon.**

Figure 2-4 Dialog Template Widget Icon

If you need help identifying the icons, turn on both names and icons from SPARCworks/Visual's View menu or bring up the Palette Icons Dialog from the Help Menu (<*Meta H*> <*p*>).

2.8.1 The DialogTemplate

The DialogTemplate is a container widget which optionally can have three kinds of children: a MenuBar, any number of buttons and one additional child which is called the *work area*. The DialogTemplate positions the MenuBar child at the top of the window and arranges all children which are buttons of any type in an evenly spaced row at the bottom of the window. This row of buttons is called the *button box*.

The DialogTemplate places the work area between the MenuBar and the button box, separated from the button box by a Separator (a horizontal line). The Separator is created automatically as part of the DialogTemplate and will appear in your widget hierarchy automatically.

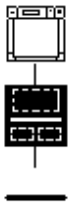


Figure 2-5 The DialogTemplate in the Hierarchy

In the tutorial interface, Figure 2-1, the menu bar cascade buttons and the pushbuttons which make up the button box are labeled. The work area contains the radio box, row column array and toggle buttons.

♦ **Assign the variable name** `diag_1` **to the DialogTemplate.**

As you add widgets to the design, we recommend that you continue to assign variable names to them. Explicit names make it easier to identify widgets and are required for certain operations. However, since SPARCworks/Visual does not strictly require names unless you refer to the widget in some way, these instructions only include this step for names which the tutorial uses later.

2.9 Dynamic Display of Layout

When you added the DialogTemplate widget, you may have noticed that your layout became visible as a small rectangle over the construction area. This is the *dynamic display window* in which SPARCworks/Visual builds a working example of your interface.

What you see in the dynamic display is a collection of widget instances. SPARCworks/Visual does not draw pictures of widgets but actually creates them using the same Motif function calls that your interface will use when it is running. Right now the dynamic display window has few identifiable features because it contains only the Shell and DialogTemplate.

As you add widgets and move them around, they appear in this window as they will appear in your finished interface. You can use the normal window manager facilities to move the dynamic display window to a part of your screen where it does not obstruct your view of the hierarchy.

When you add widgets to your hierarchy, they may not appear in the dynamic display window where you want them. Later, you will use the Layout Editor to achieve the correct appearance.

The layout shown in the dynamic display is a fully active prototype of your interface; you can click on the buttons, pull down the menus, type text into text fields, and so on.

2.10 Currently Selected Widget

In SPARCworks/Visual, one widget is always currently *selected.* A widget must be selected before you can do anything to it, such as setting its resources, cutting and pasting, or giving it children. The selected widget is highlighted in the construction area and in the dynamic display.

Widgets that cannot legally be children of the selected widget are grayed out on the widget palette so you cannot select them.

As a rule, widgets are selected when you first add them to the hierarchy. Therefore, when you add the DialogTemplate, it is automatically selected and the next widget you add will be its child. However, widgets are not automatically selected if they cannot have children. To select a different widget, click on its icon in the construction area.

2.11 Adding the Buttons

The buttons at the bottom of the layout can be added directly as children of the DialogTemplate, as shown in Figure 2-6.

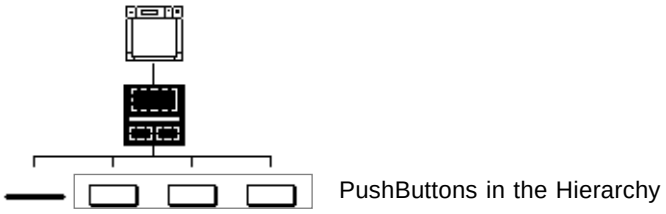


Figure 2-6 Hierarchy for the Buttons

♦ **With the DialogTemplate widget selected in the construction area, click three times on the PushButton icon.**

Each time you click, a PushButton is added as a child of the DialogTemplate. The PushButtons also appear in your dynamic display with the default label "button". Later, in the *Using the Resource Panels* chapter, you will assign proper text strings to these labels.

The dynamic display now looks like Figure 2-7.

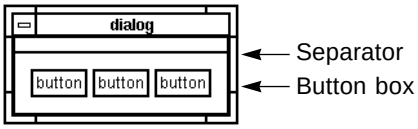


Figure 2-7 Dynamic Display of the Buttons

2.12 Building the Menu Bar

A menu bar at the top of the screen is a common feature of many computer interfaces. Motif provides a MenuBar widget, which is invisible until you add a series of other widgets to form pulldown menus. SPARCworks/Visual guides you through the process of building a menu bar by graying out all widgets except the relevant ones. The hierarchy you need to add is shown in Figure 2-8.

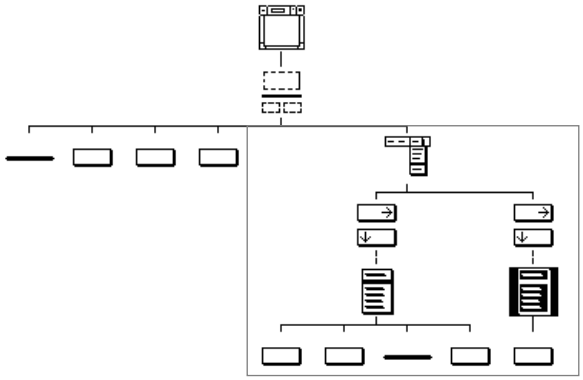


Figure 2-8 Hierarchy for the MenuBar and Its Children

2.12.1 Creating the Menu Bar

To build the menu bar:

1. **With the DialogTemplate widget selected, click on the MenuBar icon.**

The DialogTemplate automatically places the menu bar above the work area in the dynamic display.

2. **Assign the variable name** `main_menu` **to the MenuBar.**

3. **Click on the CascadeButton icon twice.**

When you add the CascadeButtons, they appear in your dynamic display with the default label “cascade” as shown in Figure 2-9.

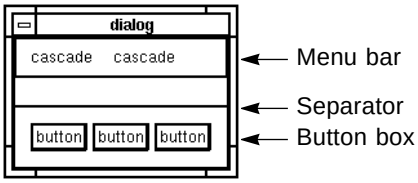


Figure 2-9 Dynamic Display of the CascadeButtons

You can click on these buttons with the mouse and see that they are active but they don’t do anything because they don’t as yet contain any menus.

4. **Select the first CascadeButton in the construction area and assign it the name:** `procedure_cascade`

2.12.2 Adding the Menus

To attach a menu:

1. **Select the left CascadeButton in the hierarchy and click on the Menu icon.**

2. **Click on the PushButton icon twice.**

3. **Click on the Separator icon; then click on the PushButton icon again.**

4. **Click on the last PushButton and assign it the variable name:** `exit_button`

The left cascade button in your dynamic display now has a working pulldown menu, which you can see by placing your cursor on the cascade button and holding down mouse button 1, as shown in Figure 2-10.

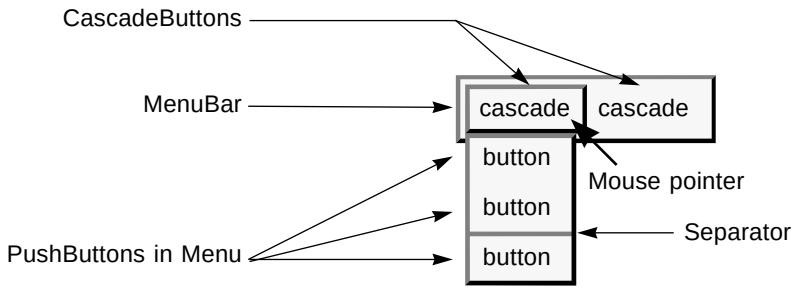


Figure 2-10 Pulldown Menu in the Dynamic Display

The PushButtons in this menu have default labels, which can be changed later. Menus can have three kinds of selectable children: PushButtons, ToggleButtons, or CascadeButtons (used to create submenus). They can also contain non-selectable Labels and Separators. This menu contains a Separator, which appears as a horizontal bar that separates the buttons into two regions.

To complete the MenuBar portion of the design hierarchy:

5. **Select the second CascadeButton in the construction area and assign it the variable name:** `help_cascade`
6. **Click on the Menu icon.**
7. **Click on the PushButton icon.**
8. **Click on the PushButton in the construction area and assign it the variable name:** `help_button`

The second CascadeButton now also has an active menu with one option which you can pull down with the mouse in the dynamic display.

2.13 Adding the Work Area

The interface now has a menu bar at the top and several buttons at the bottom. There is no work area until you add one. The DialogTemplate can have only one work area child. However, that child can be a container widget with multiple children. Since our work area will contain several widgets for choosing the ice cream flavors and toppings, we use a Form for the work area.

1. **Select the DialogTemplate in the hierarchy.**

2. **Click on the Form icon.**

The Form is invisible until you give it children. The options in our interface are arranged in three groupings:

- A RadioBox containing the “Large” and “Small” toggles
- A RowColumn array for the topping options
- Three ToggleButtons for the ice cream flavors

2.14 Building the Radio Box

The RadioBox, like the Form, is an invisible widget which exists only to control the behavior of its children. It can contain a group of ToggleButtons which it configures as radio buttons. Only one radio button can be selected by the user at any one time. The hierarchy you need to add is shown in Figure 2-11.

To build the radio box:

1. **Click on the Form in the hierarchy.**

2. **Click on the Frame widget icon.**

The black line around the “Double Scooper” and “Small” radio buttons in Figure 2-1 is not the RadioBox itself but a Frame widget which contains the RadioBox. The Frame is used to display the logical grouping of the radio box components.

3. **Click on the RadioBox icon.**

4. **Click on the ToggleButton icon twice.**

The resulting hierarchy for the radio box is shown in Figure 2-11.

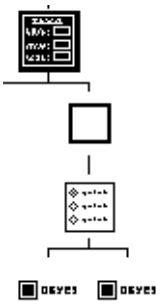


Figure 2-11 Hierarchy for the Framed Radio Box

The dynamic display should now resemble Figure 2-12:

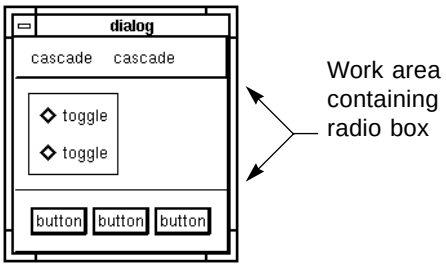


Figure 2-12 Dynamic Display So Far

2.15 Options for Viewing the Hierarchy

At this point, you may find that your widget hierarchy is too wide to fit in the construction area. You can use the window manager to make the SPARCworks/Visual interface bigger, or use the scrollbars at the bottom of the construction area to display a different part of the hierarchy. The following commands may also be helpful.

2.15.1 Fold/Unfold Widget

After you finish building a portion of your hierarchy, you may want to *fold* the topmost widget of that portion so that its children do not take up so much space in the construction area. For example, in the tutorial layout, you may want to fold the MenuBar widget because its children fill so much display space. When a widget is folded, the background color of the icon is grayed and its children are not shown in the hierarchy. Folding widgets is only a display convenience and does not remove widgets from your design.

To fold a widget, select it, pull down the Widget Menu and select "Fold / unfold" (<*Ctrl-F*>). Figure 2-13 shows a folded widget. The same command unfolds the selected widget if it is already folded.

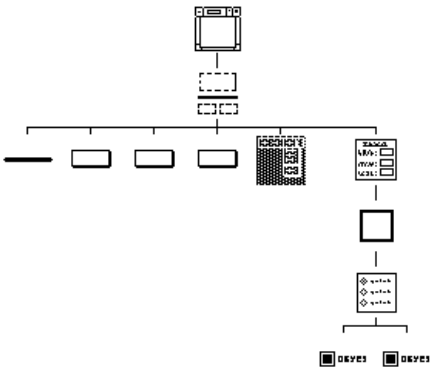


Figure 2-13 Tutorial Hierarchy So Far, with MenuBar Widget Folded

Additional commands for changing the display of the hierarchy are discussed in *The View Menu* section on page 32 of this chapter.

2.16 Building the Row Column Array

A RowColumn container widget will be used for the array of labels and text fields which specify the toppings in Figure 2-1.

1. **Select the Form in the hierarchy.**
2. **Click on the RowColumn icon.**

3. **Click on icons in the following order: Label, TextField, Label, TextField, Label, TextField.**

The Label and TextField widgets must be added in this order because the RowColumn always takes its children in order when constructing rows or columns. In this case, you are building rows and each row should have a label and a text field.

The RowColumn part of the hierarchy is shown in Figure 2-14.

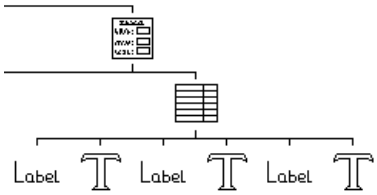


Figure 2-14 Partial Hierarchy: the RowColumn Widget and Its Children

This hierarchy results in the dynamic display shown in Figure 2-15.

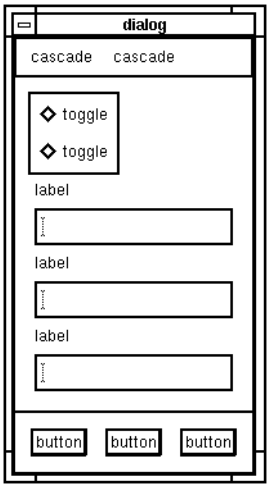


Figure 2-15 Dynamic Display So Far

Because, by default, the RowColumn widget is laid out in a single vertical column, it doesn't look much like the finished layout in Figure 2-1. Later in the tutorial, you will change its resources to achieve the desired effect.

2.17 Adding the Toggle Buttons

The last group of options required for the layout is the set of toggle buttons (representing ice cream flavors) at the bottom of the work area.

1. **Select the Form in the hierarchy.**

2. **Click on the ToggleButton icon three times.**

Your design hierarchy for the tutorial interface is now complete, as shown in Figure 2-16.

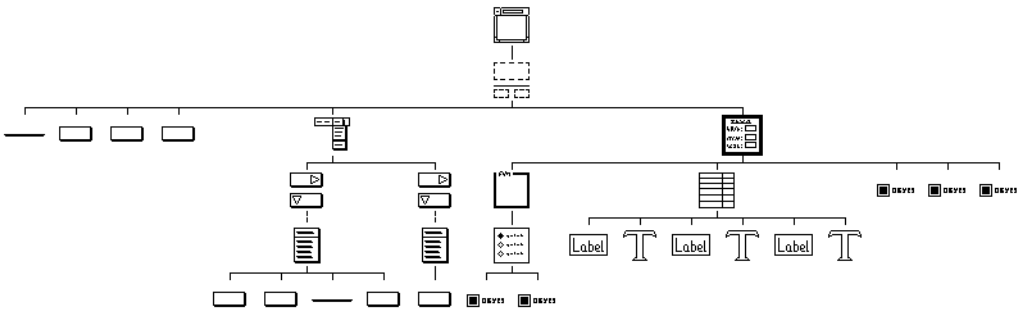


Figure 2-16 Complete Hierarchy

The interface, shown in Figure 2-17, is not yet finished but it contains all the necessary widgets in their proper parent-child relationships. In later chapters you will use resource panels and the Layout Editor to make it look more attractive.

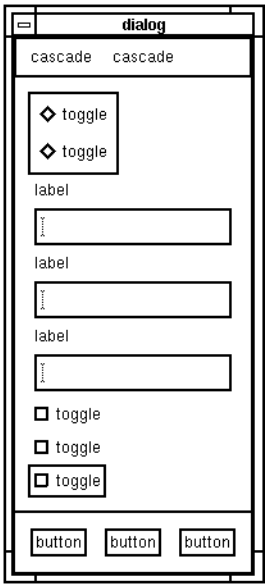


Figure 2-17 Dynamic Display So Far

Notice the difference between the appearance of the toggle buttons you just added and the ones in the RadioBox. When ToggleButtons are inside a RadioBox, they become radio buttons. Radio buttons are distinguished by a diamond-shaped indicator. ToggleButtons that are not the children of a RadioBox can be switched on and off independently and have a square indicator.

Now that you have added the last widget to the design, save your work using the “Save as...” command.

3. Select “Save as...” from the File Menu.

4. Click to the right of the text in the “Selection” text field.

5. Enter a filename for your design.

By convention, SPARCworks/Visual design files have the suffix `.xd`.

6. Click on “OK”.

If you have already saved your design, you can use the “Save” command instead.

7. This concludes the step-by-step tutorial in this chapter.

At this point, you can proceed directly to the next chapter to continue the tutorial or you can continue reading and experiment with the various editing features discussed in the following pages.

2.18 Editing the Hierarchy

SPARCworks/Visual provides dragging, cutting and pasting facilities to let you edit the hierarchy. By using these facilities, you can alter your design dramatically without losing any of the resource values you have specified.

All editing functions act equally on the children of the selected widget. This lets you retain the relative positions of widgets inside a container widget such as a Form or RowColumn by moving the container widget and everything beneath it as a unit.

2.18.1 Dragging Widgets Around the Hierarchy

To drag a widget and its children to a new location, hold down mouse button 1 over the widget and drag it to its new location. When the widget is correctly positioned beneath a potential parent, a vertical line appears connecting it to the new parent. When you see the line, release the mouse button. If there is no line when you release the mouse button, the widget being dragged reverts to its former position.

2.18.2 Rules When Dragging Widgets

You can drag widgets to a different position beneath the same parent, or to a new parent. However, SPARCworks/Visual does not let you drag a widget to a position which is not valid in Motif.

Widgets that are part of a composite widget, such as the ScrollBars which form part of the MainWindow, can only be dragged by dragging their parent.

Because a widget’s children are dragged with it, you cannot drag a widget to a position beneath its own child. To get this effect, use the copying facility described below.

If you change your mind after starting to drag a widget, you can cancel by dragging to an empty spot in the construction area.

The Shell widget cannot be dragged because it is not a valid child of any class of widget.

2.18.3 Copying Widgets

To copy a widget and its children to a new location while leaving the original widget in place, drag the widget using mouse button 2.

2.18.4 Edit Commands: Cut, Paste, Copy and Clear

The Edit Menu has “Cut”, “Paste”, “Copy”, and “Clear” commands which can also be used to alter the hierarchy. To copy a widget and its children onto the SPARCworks/Visual clipboard, select the widget and use the “Copy” command (*<keypad>Copy*).

“Cut” (*<keypad>Cut*) deletes the selected widget and its children and copies them onto the clipboard. “Clear” also deletes the selected widget and children but does not affect the clipboard. Cleared items cannot be pasted back into the hierarchy.

“Paste” (*<keypad>Paste*) inserts the contents of the clipboard directly beneath the currently selected widget. “Paste” is disabled if the clipboard is empty, or if the widget in the clipboard is not a valid child of the currently selected widget. The pasted widget is always made the last child of the selected widget. To place it in a different position, drag the selected widget with the mouse.

2.18.5 Copy to File, Paste from File

As well as copying to the SPARCworks/Visual clipboard, you can copy a widget and its children to a clipboard file and paste in a widget from an existing clipboard file. This feature lets you build a library of design fragments, such as a standard menu bar. By convention, SPARCworks/Visual clipboard filenames have the suffix `.cxd`.

2.18.6 Alternate Method of Selecting Widgets

To select any widget that is not highlighted, you can use the mouse or you can step up, down, left, or right in the hierarchy by using the arrow keys. The arrow keys only work this way when the construction area has the input focus. If the arrow keys seem to be disabled, use the <*Tab*> key to cycle the focus around the various areas of the SPARCworks/Visual screen until they become active.

2.19 The View Menu

The View Menu offers four options that affect the display of the SPARCworks/Visual hierarchy. Each option is a toggle and can be turned on or off. By default, all options are off when you start SPARCworks/Visual. These options only change the appearance of the SPARCworks/Visual display and do not affect your design.

2.19.1 Show Widget Names

This option (<*Ctrl-W*>) displays the name of each widget beneath its icon in the construction area as shown in Figure 2-18. The name shown is the unique variable name assigned to the widget, not the widget name.

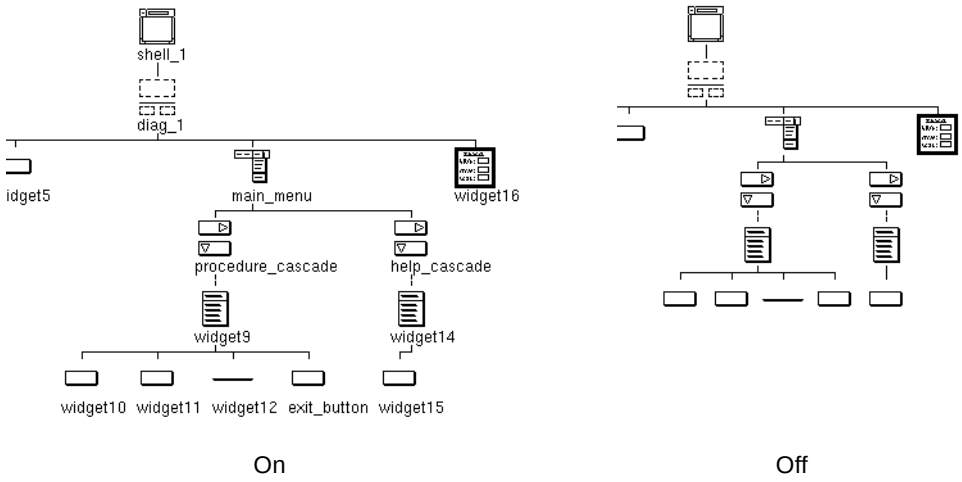


Figure 2-18 Show Widget Names

2.19.2 Show Dialog Names

Each Shell widget in the design is represented by an icon in the rectangular area at the top right corner of the SPARCworks/Visual screen. This rectangular area is called the *window holding area.* "Show Dialog Names" (<*Ctrl-D*>) displays the variable name of each Shell widget beneath its icon in the window holding area, as shown in Figure 2-19. The icon shrinks to accommodate the name. This feature is useful in layouts with multiple windows.

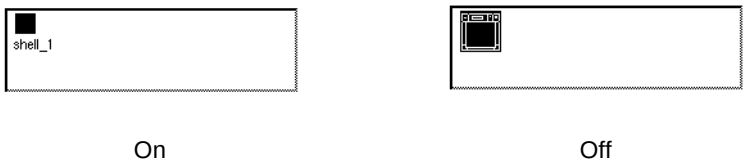


Figure 2-19 Show Dialog Names (Window Holding Area Shown)

2.19.3 Left Justify Tree

This option (<*Ctrl-L*>) changes the appearance of the hierarchy in the construction area from a centered tree with branches spreading in both directions to a left-justified tree with branches spreading to the right, as shown in Figure 2-20. This feature can be useful for the rapid location of parent widgets in large designs

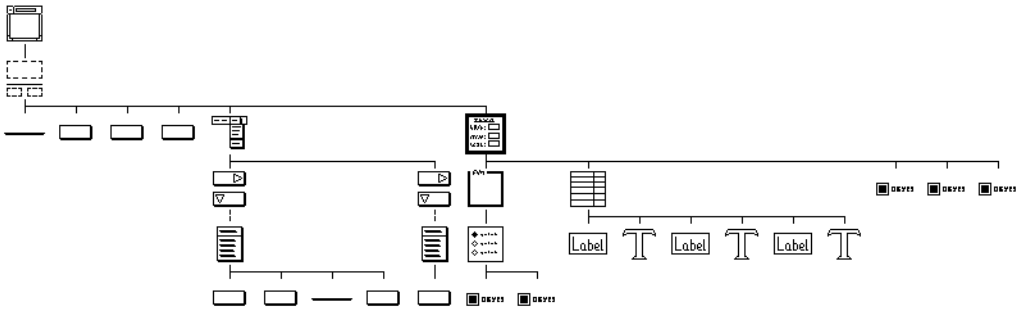


Figure 2-20 Left Justified Hierarchy

2.19.4 Shrink Widgets

This option is useful when the hierarchy is large and you want to see more of the structure in the same size window. The widgets shrink to a uniform small square so that more fit in the construction area, as shown in Figure 2-21. However, the distinction between widget classes and between folded and unfolded widgets, is lost. As with the other View options, your actual design is not affected.

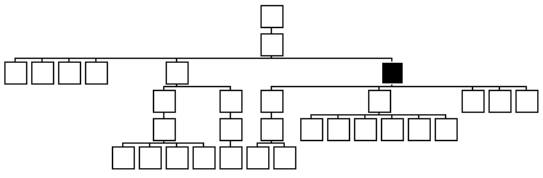


Figure 2-21 Shrink Widgets

The “Fold/unfold” command in the Widget Menu, described earlier in this chapter, is another way to save display space.

The “Structure Colors” option is related only to structured code features, which are discussed in the *Coding Techniques* and *Command Summary* chapters.

2.20 Printing Your Hierarchy

The Print Dialog lets you print out a hard copy of your hierarchy at any time while you are developing it. The Print Dialog is shown in Figure 2-22.

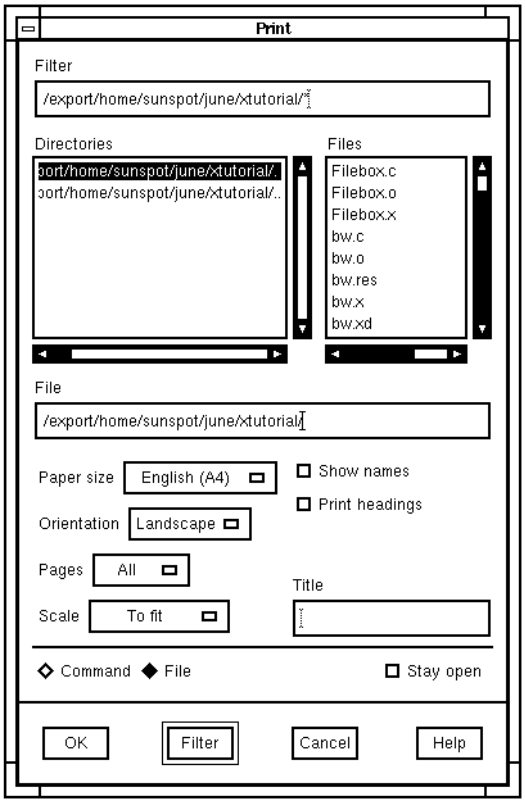


Figure 2-22 Print Dialog

To print to a file, click on the “File” toggle and enter the filename in the text box under “File”. To send to a printer, click on the “Command” toggle and enter the command, such as `lpr`, in the same text box, which is now labelled “Command”. The output is Postscript, so a Postscript printer or viewer is required.

The option menus in the Print Dialog let you specify the page size, orientation, pages and scale. In the “Scale” option menu, the reduced scale option prints the diagram two-thirds of its actual size. Note that if the “Scale to fit” option is not selected, the diagram prints on as many pages as required. The “Pages” option menu lets you print either all the hierarchies in your design if your design contains more than one window or just the hierarchy currently displayed in the construction area.

Selecting the “Show names” toggle lets you print the variable names of the widgets. Selecting the “Print headings” toggle puts a border around the hierarchy and prints a title, which you can specify in the “Title” text field. The title is restricted to one line of text.

2.21 Using the File Browser

The file browser, shown in Figure 2-23, lets you specify the name of a SPARCworks/Visual file to open or save. The file browser is displayed when you select any command that requires you to specify a filename, such as the “Open” and “Save as...” commands from the File Menu. You can either enter a pathname in the “Selection” field or use the mouse to select an existing filename from the “Files” list.

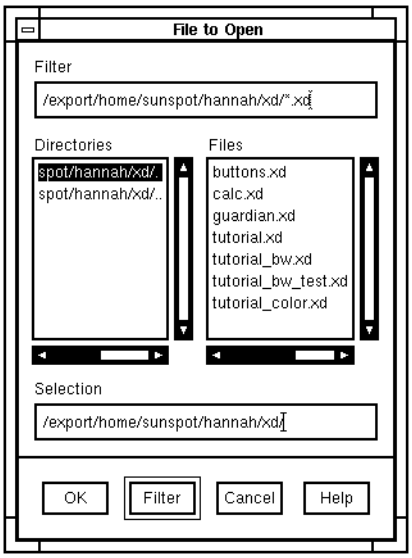


Figure 2-23 The File Browser

The “Filter” text field displays the current directory and a filename pattern to be matched in the “Files” list. You can change the current directory and filename pattern by editing the text in the “Filter” text field and clicking on the “Filter” button at the bottom of the screen.

The subdirectories of the current directory appear in the “Directories” box. To navigate through the directory structure, either click on a selection in this list and then click on “Filter”, or double-click on the selection.

The filename pattern controls the “Files” listing. Any filenames in the current directory that match the pattern appear in the “Files” box. You can change the pattern by editing the text and clicking on the “Filter” button at the bottom of the screen. If the pattern is an asterisk (*), all files in the current directory are listed. If the pattern is `*.xd`, only files that have the `.xd` suffix are listed. To select a file, either click on the filename then click on the “OK” button at the bottom of the screen, or double-click on the filename. When you select a file, SPARCworks/Visual proceeds with the operation you requested, such as “Open”, “Read”, or “Save as....”.

When you save a file or generate code, you can either select an existing filename or specify a new filename in the “Selection” field and click on “OK”.

Note that if files have been added to the current directory since the filter has been applied, they will not appear in the “Files” listing until the filter is re-applied. This is the case even if the dialog has been dismissed and applied again in the meantime.

Using the Resource Panels 3

3.1 Introduction

In Motif, the appearance and behavior of a widget is controlled by its resources. Resources include colors, fonts, images and text, titles, positions and sizes of windows or widgets, callbacks, and all other customizable parameters that can affect the behavior of the interface. Resources can have indirect as well as direct effects. For example, changing the label on a PushButton also changes the size of the button to allow space for the new label.

When you add a widget to the design hierarchy, SPARCworks/Visual uses default values for all of its resources. In most cases, however, you must set some resources explicitly to make the widget useful. To make it easy to set these resources, SPARCworks/Visual groups resources on dialogs called *resource panels.*

In this chapter, you will use resource panels to:

- Display the resources of widgets in the hierarchy
- Edit the text for Labels and the various types of button widgets
- Designate keyboard accelerators and mnemonics
- Designate a Help widget for the menu bar
- Set the arrangement of rows and columns in a RowColumn widget
- Edit resources for the Shell widget, including the main title at the top of the dialog frame
- Designate a callback

3

In addition to resource panels, SPARCworks/Visual offers special editors for laying out widgets in a Form, setting fonts and pixmaps, editing *XmString* (Motif compound string) structures and selecting colors. These special editors are discussed in separate chapters.

3.2 The Label Resource Panel

To make the tutorial interface meaningful, you must set the text of all labels and buttons to something other than the default. Begin by changing the three labels in the RowColumn widget to the text shown in Figure 3-1.

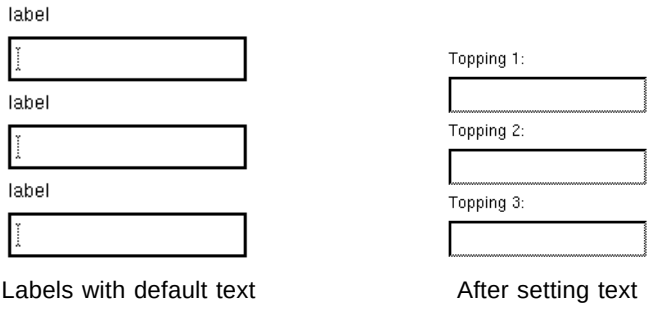


Figure 3-1 Labels Before and After Setting Text

First, bring up the resource panel for the first label:

1. **Select the first Label under the RowColumn widget.**
2. **Click on the Label's icon a second time.**

When you click on the Label the second time, SPARCworks/Visual displays its resource panel. Figure 3-2 shows part of the panel.

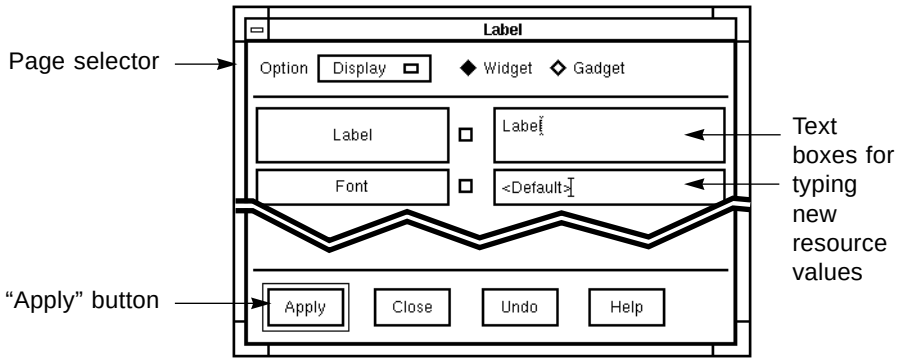


Figure 3-2 Label Resource Panel (Partial)

Resource panels usually have several pages. The option menu in the top left corner of the panel is a page selector. If the "Display" page is not already selected:

3. Select the "Display" page.

Now, edit the text of the label.

4. Double-click in the text box opposite "Label".

Editing text in these boxes works in much the same way as assigning widget names. When you double-click, the first word in the box is highlighted. Triple-clicking highlights all the words in the box. Entering new text replaces the highlighted text.

5. Type: `Topping 1:`

Note – Do not press <*Return*>. Labels can contain multiple lines and pressing <*Return*> inserts a newline character into your label. If you unintentionally press <*Return*>, you can backspace to remove the newline.

---

6. Click on the "Apply" button at the bottom of the resource panel.

The "Apply" command sets the new resource value. When you click on "Apply", the dynamic display shows the new label text.

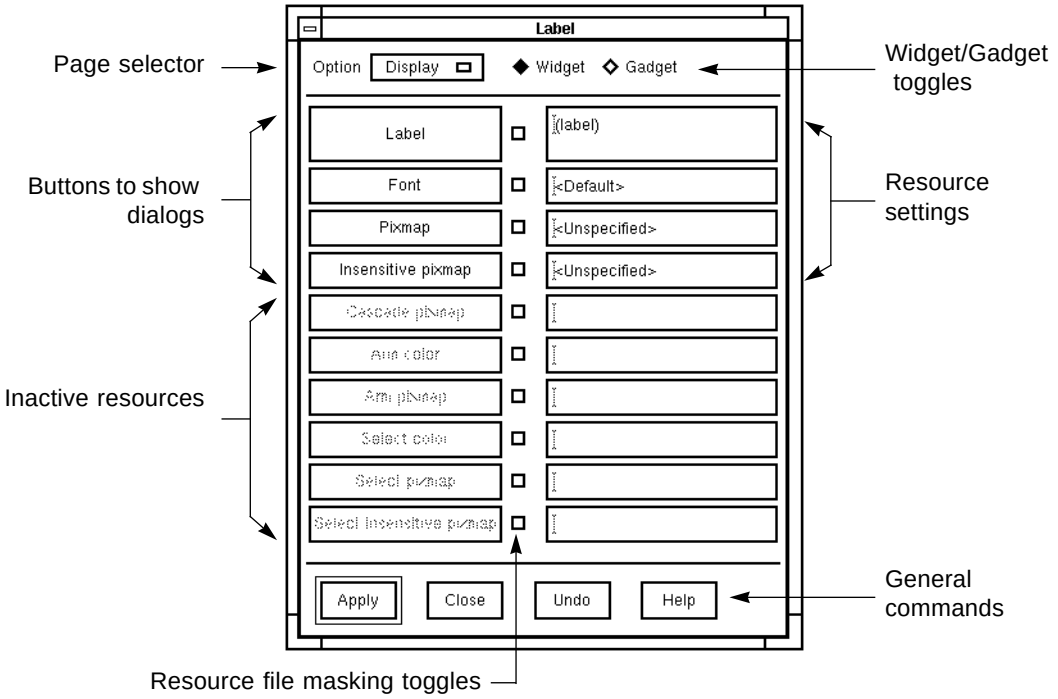


Figure 3-3 The Label Resource Panel

3.2.1 Regions of the Resource Panel

Although different classes of widgets have different resource panels, all resource panels have the same basic structure. The central section of the panel displays a group of resource settings for the widget. Resource names are shown in the boxes on the left. Any resources that do not apply to the selected widget are grayed out. Current resource values are shown in the boxes on the right. These boxes are either single-line text fields, multi-line text boxes or menu options. You can edit resource settings by typing into these boxes. Default values are shown in parentheses.

3.2.2 Masking Toggles

The unlabeled toggle between each resource name and its setting is the *masking toggle* which can be used at the code generation stage to mask resources in or out of the X resource file. This topic is discussed in the *Generating Code* chapter. You do not have to set these toggles in order to set resource values.

3.2.3 Page Selector and Toggle Switches

The main section of the resource panel usually has several pages. The option menu at the top of the panel is a *page selector* which lets you move from one page to another. The Label panel also has a toggle switch which can be used to designate this widget as a gadget. The widget-gadget toggle appears in the resource panels for other widget classes if the widget class has a gadget counterpart. Some widget classes also have other toggle switches appropriate to the class.

3.2.4 General Commands: Apply

The four buttons at the bottom of the panel offer general commands. “Apply” causes your new resource settings to take effect. If you do not click “Apply”, your edited settings are lost when you select another widget or close the resource panel.

3.2.5 Undo, Close, Help

“Undo” makes all edited settings revert to the last applied settings. “Close” makes the resource panel disappear. “Help” displays the appropriate help screen.

Note that in Figure 3-3, the “Apply” button is highlighted. Pressing <*Return*> when a resource panel has the input focus generally executes the highlighted command. However, <*Return*> does not work in this way when you type into a multi-line text box, because the text box accepts <*Return*> as a newline character.

Now set the text of the other two Label widgets. You do not have to close the Label resource panel. However, you may need to move it if it covers the construction area.

1. **Select the second Label widget.**

Note that the “Label” resource on the panel changes to reflect the current setting of the newly selected widget.

2. **Double-click in the “Label” box in the resource panel to highlight the default label.**

Additional ways to edit in text boxes include: dragging with the mouse to highlight text, using <*Delete*> to delete highlighted text, or just typing at the current cursor location.

3. **Type**: `Topping 2:`

4. **Click on “Apply”.**

5. **Repeat Steps 1 through 4 for the third Label, using the text**:

`Topping 3:`

6. **Click on “Close”.**

“Close” makes the resource panel disappear.

3.3 Resource Panels in Windows Mode

When SPARCworks/Visual is running in Windows mode it is “conscious” of making sure that the design you are creating is Windows compliant. Motif resources do not map directly to Windows resources. Some resources can be translated into something similar on Windows (which may not necessarily be a resource), some cannot.

Resources which are not mapped to Windows have their fields colored pink in the resource panel. You can still enter values into these fields and they will be generated for Motif (both plain Motif and MFC Motif), but nothing will be generated for Windows.

You can change the color of these non-Windows resource fields by changing an entry in the SPARCworks/Visual application resource file.

3.4 Button Widget Resources

Use the same procedure to set the labels of ToggleButtons, CascadeButtons and PushButtons. Set labels for the ToggleButtons in the work area as shown in Figure 3-4.

Figure 3-4 ToggleButtons Before and After Setting Labels

1. Click twice on the first ToggleButton to bring up its resource panel.

You can also bring up the selected widget's resource panel by selecting the "Resources" command on the Widget Menu, or by pressing <*Return*> any time the input focus is in the construction area.

2. Click twice in the "Label" box and type: `Vanilla`

3. Click on "Apply".

4. Select the second ToggleButton.

5. Click twice in the "Label" box and type: `Chocolate`

6. Click on "Apply".

7. Select the third ToggleButton.

8. Click twice in the "Label" box and type: `Strawberry`

9. Click on "Apply".

3.5 Shared Resource Panels

Notice that the resource panel for a ToggleButton is the same as for a Label and has the word “Label” in the title. The ToggleButton uses the Label’s resource panel because, in Motif, the ToggleButton widget class is *derived* from the Label class. Another way of saying this is that the ToggleButton is a specialized kind of Label, also called a *subclass* of Label. The Label is the ToggleButton’s *superclass*.

3.5.1 Widget Class Inheritance

The ToggleButton class *inherits* most of its characteristics, including most of its resources, from the Label class and uses the same resource panel. By means of inheritance, all Motif widget classes are organized in a hierarchy known as the *class hierarchy*. The class hierarchy is an abstract hierarchy of available widget classes and should not be confused with the design hierarchy you are building on the screen.

Several types of buttons - ToggleButtons, PushButtons, CascadeButtons and DrawnButtons - are derived from the Label class. You can set the text of all widgets of these classes using the same Label resource panel without having to close and re-open it.

Set labels for the radio buttons (the ToggleButtons in the RadioBox) as shown in Figure 3-5.

Default labels After setting

Figure 3-5 Radio Buttons Before and After Setting Labels

1. **Click twice on the first radio button to bring up its resource panel.**
2. **Click twice in the “Label” box and type:** `Large`
3. **Click on “Apply”.**

4. **Select the second radio button.**

5. **Click twice in the “Label” box and type:** `Small`

6. **Click on “Apply”.**

If you assign a label to the radio button which is longer than the default label, the Frame widget changes size to accommodate the new width. This represents a chain reaction: the ToggleButtons resize to accommodate longer text strings, then their RadioBox parent and its Frame parent resize to fit in turn. Motif does all this automatically. This behavior is not obvious in the tutorial layout but it is illustrated in Figure 3-6.

Figure 3-6 Radio Buttons With Longer Labels

Use the same Label resource panel to set the label resources of the CascadeButtons as shown in Figure 3-7.

Figure 3-7 Menu Bar Before and After Setting CascadeButton Labels

7. **Select the first CascadeButton and assign it the label:** `Procedures`

8. **Select the second CascadeButton and assign it the label:** `Help`

Finally, set the labels of the three PushButtons in the button box, as shown in Figure 3-8. PushButtons also resize automatically to accommodate their labels.

Figure 3-8 Button Box Before and After Setting Resources

9. **Select the first PushButton and assign it the label:** `Cone`
10. **Select the second PushButton and assign it the label:** `Dish`
11. **Select the third PushButton and assign it the label:** `Cancel`

3.5.2 Tip

Remember to click on “Apply” after setting resources, or your new settings will have no effect. If you do not apply your new settings before selecting another widget, SPARCworks/Visual displays the warning shown in Figure 3-9.

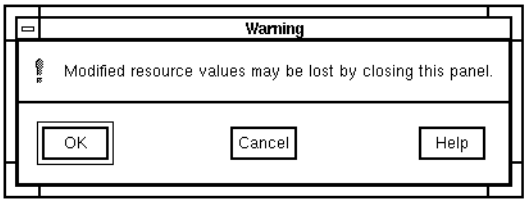


Figure 3-9 Warning that New Resources Were Not Applied

Click “Cancel” on this display to return to the resource panel.

3.6 Resources for Menu Items

Next, set resources for the buttons in the pulldown menus of the menu bar. So far you have set only the CascadeButton labels and the pulldown menus look like Figure 3-11.

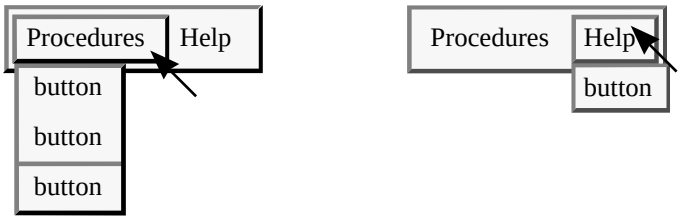


Figure 3-10 Pulldown Menus After Setting CascadeButton Labels

1. Select the first PushButton child of the first Menu widget.

2. Change the "Label" resource to: `Wash Dishes...`

By convention, the ellipsis (...) is used to indicate that a menu item brings up an additional dialog box before the command is executed.

3. Select the second PushButton and assign it the label:

Count Money

4. Select the third PushButton and assign it the label: `Exit`

5. Select the PushButton child of the second Menu widget.

Assign it the label: `About This Layout`

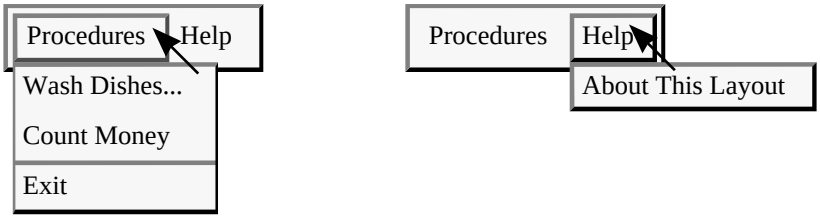


Figure 3-11 Pulldown Menus After Setting PushButton Labels

In the following sections, you will:

- Set the new labels on the PushButtons
- Designate <*Ctrl-E*> as an accelerator for the "Exit" button

- Put a visible “Control+E” label on the “Exit” button to help the user remember the accelerator
- Designate “H” as a mnemonic for “Help” and “A” as a mnemonic for “About This Layout”

3.7 The Keyboard Page

Accelerators and mnemonics are keyboard resources which are found on a separate page of the Label resource panel. Use the following steps to set an “H” mnemonic on the “Help” CascadeButton:

1. **Select the “Help” CascadeButton.**
2. **Select “Keyboard” from the resource panel’s page selector.**

This brings up the “Keyboard” page, shown in Figure 3-12. The resources that are not grayed out are the ones which apply to the CascadeButton class.

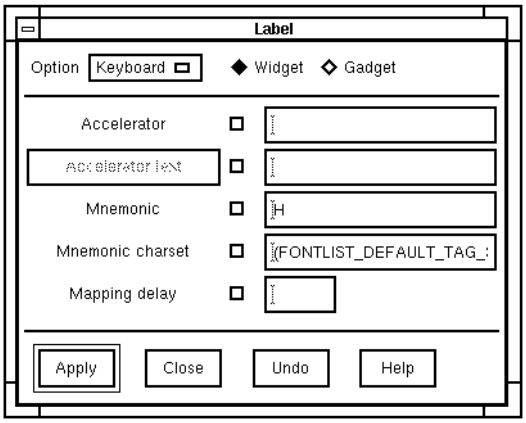


Figure 3-12 Keyboard Resources for the CascadeButton

3.7.1 Mnemonics

SPARCworks/Visual lets you set keyboard mnemonics which work like those in the SPARCworks/Visual interface. A mnemonic can be any character, even one which is not in the label. It is easiest for the end user if you use a character which appears in the label, preferably the first character. Mnemonics must be unique within a menu bar or menu.

1. **Double-click in the "Mnemonic" box.**
2. **Type**: `H`
3. **Click on "Apply".**
4. **Select the PushButton child of the "Help" menu.**
5. **Double-click in the "Mnemonic" box.**
6. **Type**: `A`
7. **Click on "Apply".**

Note that the "H" and "A" characters now appear underscored, which is Motif's way of indicating a mnemonic. This lets the user invoke the "About This Layout" command in two ways: with the mouse, or by pressing *<Meta-H>, <A>*.

3.7.2 Accelerators

Now add the keyboard accelerator for the Exit button. As in SPARCworks/Visual, an accelerator immediately executes a menu command, whether or not the menu is displayed.

1. **Select the "Exit" button (the third PushButton under the first pulldown Menu).**
2. **Double-click in the "Accelerator" box.**
3. **Type the text string**: `Ctrl<Key>E`
4. **Click on "Apply".**

These steps make the accelerator active. When the interface is running, <*Ctrl-E*> will have the same effect as the “Exit” button. The exact syntax of the accelerator is important. If a syntax error occurs, SPARCworks/Visual displays the error message shown in Figure 3-13 when you try to apply.

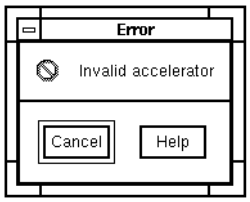


Figure 3-13 Accelerator Syntax Error Message

Accelerator syntax is the same as that used for translation tables. This topic is discussed in the *Translations* chapter.

3.7.3 Accelerator Text

A related resource is *accelerator text*. This resource displays extra text on the Menu pushbutton to remind the user what the accelerators are. Since accelerator text is just a display convenience, you do not have to use any particular syntax. “Control+E”, “Ctrl-E” and “^E” are some common forms.

1. **Double-click in the “Accelerator Text” box.**
2. **Type the text string:** `Control+E`
3. **Click on “Apply”.**

When you pull down the left menu, you now see the new labels on all the buttons and the designated accelerator text on the Exit button, as shown in Figure 3-14.

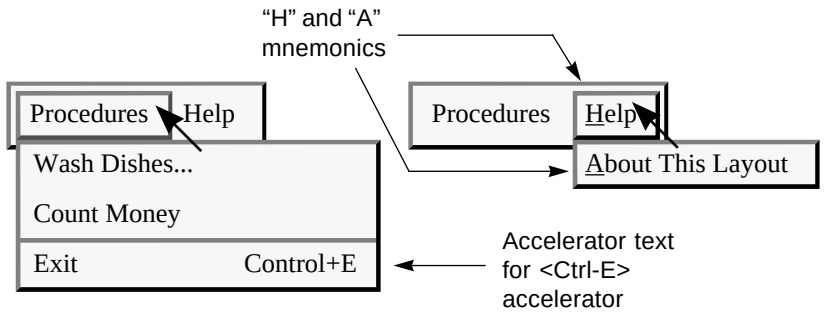


Figure 3-14 Pulldown Menus After Setting Resources

4. Click on "Close".

3.8 Designated Help Widget

The *Motif Style Guide* suggests designating one CascadeButton child of the MenuBar as the Help widget. The Help widget always appears at the right end of the menu bar.

1. Click twice on the MenuBar to bring up its resource panel.

The resource panel has "RowColumn" in the title. The MenuBar is a specially configured RowColumn and shares its resources. The resources that are not grayed out apply to the MenuBar.

2. Select the "Display" page if not already selected.

To designate a Help widget:

3. Click in the "Help widget" field and type: `help_cascade`

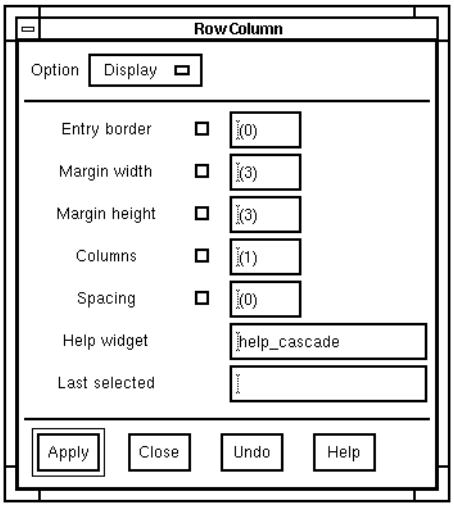


Figure 3-15 Display Resources for the MenuBar

4. Click “Apply”.

The Help widget must be one of the CascadeButtons in the menu bar. You must type the CascadeButton’s variable name exactly. If you make a mistake, SPARCworks/Visual does not accept the entry.

The dynamic display does not show this change automatically. To see the effect of designating the Help widget, resize the dynamic display.

Leave the RowColumn resource panel open since you will use it in the next section on RowColumn resources.

3.9 RowColumn Resources

By default, the RowColumn widget has one vertical column. Its resources can be set to change it from this default state to an arrangement of three horizontal rows, as shown in Figure 3-16.

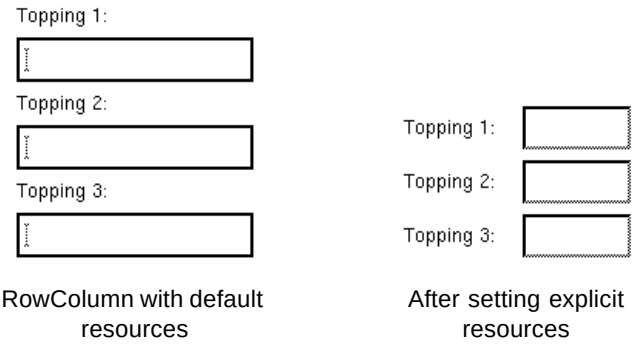


Figure 3-16 RowColumn Array, Before and After Setting Explicit Resources

To achieve the result shown in Figure 3-16, you must also set the size of the TextFields.

1. Select the first TextField widget in the hierarchy.

Note that when you select a TextField widget, the RowColumn resource panel, which is still present on the screen, becomes inactive. Its “Apply” button is grayed out and if you try to type into its text fields, your machine beeps. The panel becomes active again whenever a RowColumn widget or subclass widget is selected.

2. Click on the TextField again to bring up its resource panel and select the “Display” page.

The TextField is a variant of the Text widget and shares its resource panel, shown in Figure 3-17. The distinction is that Text widgets can contain multiple lines of text, while TextFields are limited to a single line. Because these classes have no gadget counterparts, this resource panel has no widget-gadget toggle. There is, however, a toggle to change the widget from TextField to Text. This toggle lets you change from one variant of the Text widget to the other without disturbing your hierarchy or resource settings.

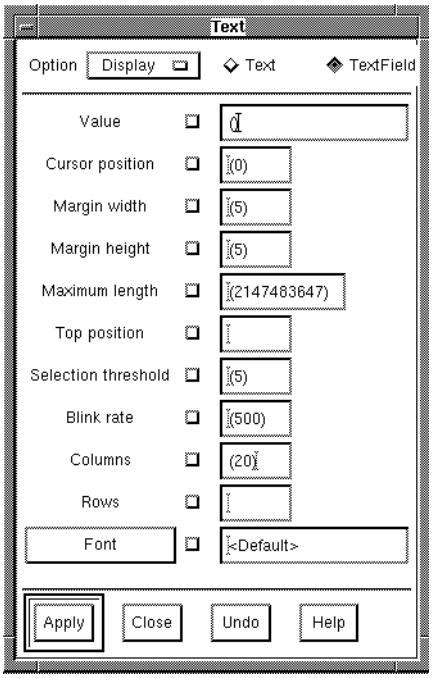


Figure 3-17 "Display" Page of TextField (or Text) Resource Panel

We want to make the text field boxes narrower. The size of the text field boxes is determined by two factors: the TextField's "Columns" resource and the rules imposed by its RowColumn parent. Begin by setting the "Columns" resource to a smaller number:

3. **Double-click in the "Columns" box.**
4. **Type:** 8
5. **Click on "Apply".**

SPARCworks/Visual accepts the new value even though the box does not change size in the dynamic display. If the entry had been rejected, the value in the resource panel would revert to the default. However, the TextField's appearance is also controlled by its parent, the RowColumn. The RowColumn, by default, forces all its children to be the same width and chooses the width of the largest child. Therefore, all the TextFields in the dynamic display remain the default size until all of them are made smaller. At that time they will all resize at once.

6. Select the second TextField.

7. Repeat Steps 3 through 5.

8. Select the third TextField and repeat Steps 3 through 5.

After you resize the third TextField, the RowColumn no longer has any default-size children. So, by its rules, all its child widgets can become smaller, as shown in Figure 3-18.

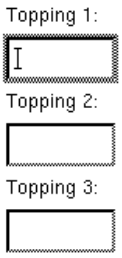


Figure 3-18 RowColumn After Resizing TextFields

The "Columns" resource of a TextField or Text widget only affects the size of the box and not the number of characters the user can enter. If you want to limit input to a specific number of characters, set the "Maximum Length" which defaults to a very large number.

3.9.1 The RowColumn Resource Panel

To get the layout of three horizontal rows, you need to set the resources of the RowColumn.

1. Select the RowColumn widget in the construction area.

If you already have the RowColumn resource panel up on your screen, it becomes active again and reflects the current settings of the RowColumn. If it is not on the screen, bring it up by clicking on the RowColumn again.

2. Select the “Settings” page.

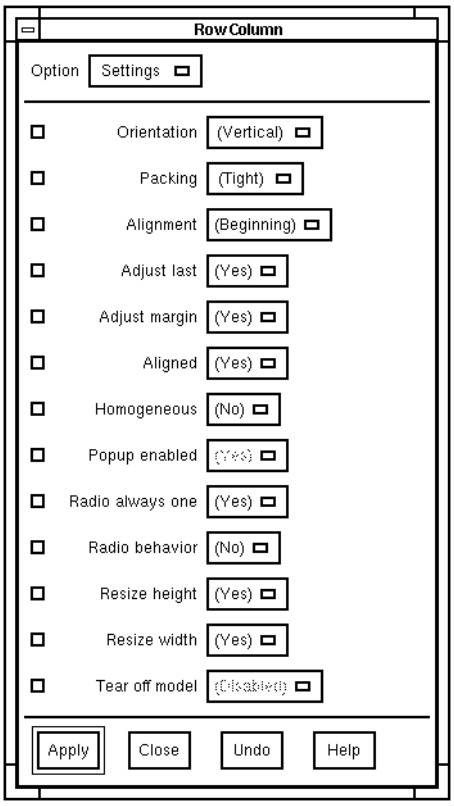


Figure 3-19 “Settings” Page of RowColumn Resource Panel

The “Settings” page, shown in Figure 3-19, lists resources that have a limited number of possible settings. Therefore, it has option menus instead of text fields in its right column.

3. In the “Orientation” option menu, select “Horizontal”.

4. In the “Packing” option menu, select “Column”.

"Horizontal" orientation lays the RowColumn out in rows rather than columns. "Column" packing is necessary if the RowColumn is to have more than one row or column. To see the effect of these two resources:

5. Click on "Apply".

At this point, the RowColumn appears in a horizontal row, with all the cells the same size. When "Orientation" is "Horizontal", the sense of rows and columns is reversed. Therefore, to make three rows you must set the "Number of Columns" resource.

6. Select the "Display" page.

7. Double-click in the "Columns" box and type: 3

8. Click on "Apply".

The result is shown in Figure 3-20.

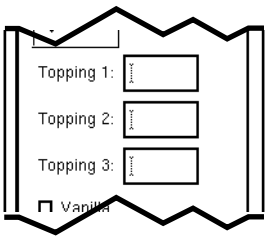


Figure 3-20 RowColumn Portion of the Dialog with Horizontal Orientation

3.10 Setting Resources for the Shell

The Shell widget exists primarily as an interface between your application and the X window system. Its behavior is mainly controlled by the window manager and by the widgets it contains. However, it does have a few interesting resources of its own.

1. Click twice on the Shell widget to bring up its resource panel.

At the top of this panel is a toggle switch which sets the type of Shell widget. Every application must have at least one (and usually only one) *Application Shell* which serves as the main window and as the interface of the application to the X window system.

Subsidiary windows can be either *Dialog Shells* or *Top level Shells*. This distinction is discussed in the *Additional Windows and Links* chapter where you will create a secondary Shell for this interface. The present Shell, which is the main window, must be an Application Shell. Therefore:

2. Click on the “Application shell” toggle.

You can also set the main dialog title on this resource panel. To do this:

3. Select the “Display” page.

4. Double-click in the “Title” box and type: `Ice Cream Shop`

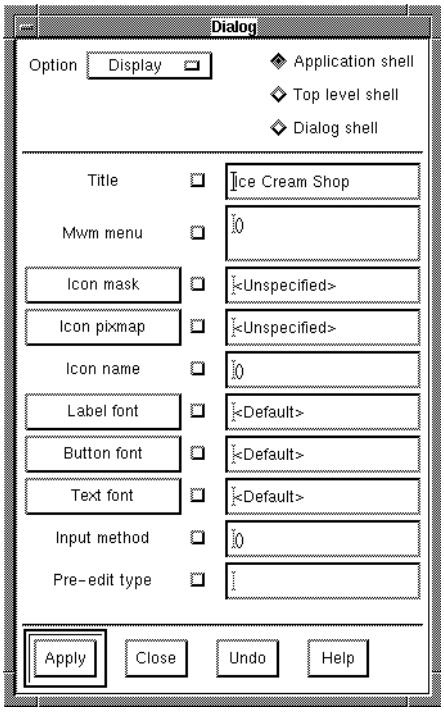


Figure 3-21 “Display” Page of the Dialog (Shell) Resource Panel

5. Click on “Apply”.

6. Click on “Close”.

The layout now looks like Figure 3-22.

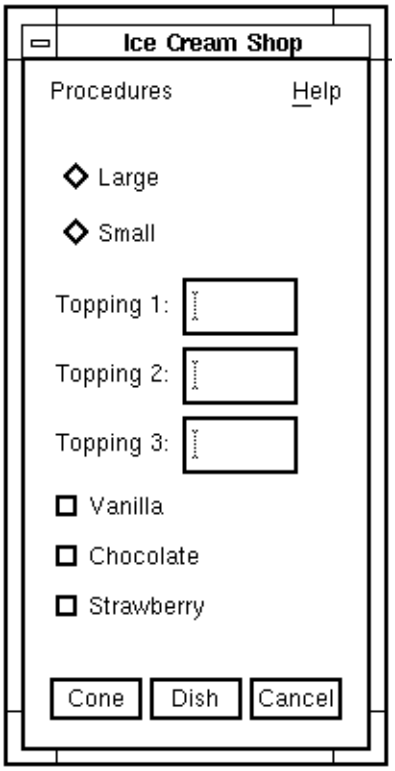


Figure 3-22 Tutorial Interface So Far

3.11 Designating a Callback

A *callback list* is a list of *callback functions* designated to be triggered by *user actions* in your application. User actions include mouse button presses, keyboard selections and movements of the pointer. By setting up a *callback*, you can instruct your interface to call the functions on the callback list whenever a certain user action occurs within a widget.

In the following steps you will designate the simplest example of a callback. Clicking on the “Exit” button (*exit_button*) triggers a callback list with just one function, *quit(),* which terminates the program.

The “Callbacks” page of the resource panel lets you associate lists of functions with user actions. You can associate *quit()* with *exit_button* now, even though *quit()* has not yet been written.

quit() and other callback routines are written in C or C++ and linked in with the code generated by SPARCworks/Visual. You will write your *quit()* routine in the *Generating Code* chapter of this tutorial. The topic of writing callbacks is discussed in greater depth in the *Coding Techniques* chapter of this book.

1. **Click twice on the “Exit” button (exit_button) to bring up its resource panel.**

2. **Select the “Callbacks” page.**

The “Callbacks” page, shown in Figure 3-23, displays a list of possible callbacks for the selected widget.

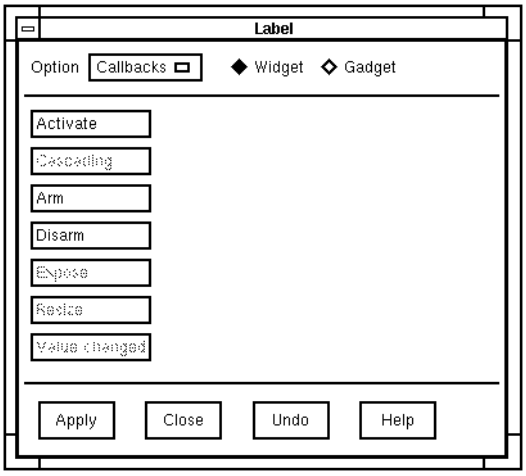


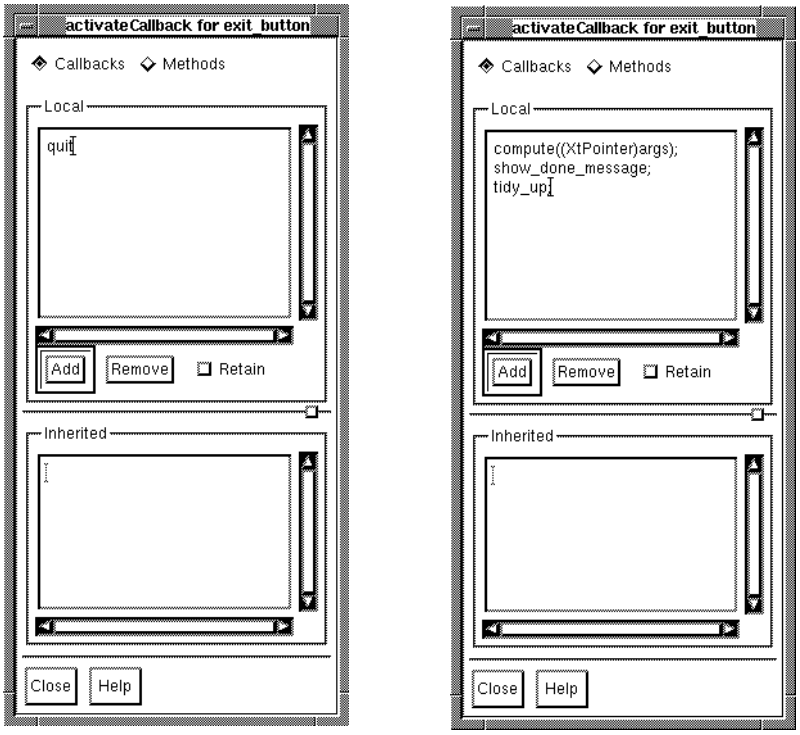
Figure 3-23 Callback Resources for PushButton Widget in Menu

You are going to associate *quit()* with “Activate”. Activating means that the user presses and then releases a mouse button while the pointer is located inside the widget. The user can also activate with the <*Return*> key, or other keys as described in the *Motif User Guide.*

Each type of callback displayed on the “Callbacks” page is a button which produces a text box when you click on it. To add an Activate callback:

3. Click on the “Activate” button.

This displays a dialog containing two lists. The first shows those callbacks local to the widget and the second those inherited by it. (Inherited callbacks are only applicable to widgets that have been designated as C++ classes or instances of definitions which contain widgets on which callbacks have been declared.) You may only change the local list. Figure 3-24 shows two typical examples. Example A of the figure shows the callback list for the “Exit” button in the tutorial interface; Example B shows a slightly more complex list.



Example A Example B

Figure 3-24 Callback Text box and Two Examples of Syntax

3.11.1 Callback Syntax

In general, the syntax for each function call in the callback list is the same as C syntax. However, if your callback list has only one function call, it does not have to end with a semicolon (as in example A). Function brackets () are not required unless you are passing a parameter to the callback.

1. **Click in the local definitions list.**
2. **Type**: `quit`
3. **Click on "Add".**
4. **Close the dialog.**
5. **Save your design.**

This concludes the tutorial portion of this chapter. During the rest of this chapter you will learn additional important techniques for handling callbacks and other resources.

3.11.2 Order of Execution

While the callback list looks like C code, it has no logical flow. This means that you can neither use C's logical operators such as *if...else* and *while*, nor can you rely on your callbacks being executed in any particular order. All routines in your list will be executed whenever the specified event occurs but not necessarily in the order you type them. If the execution sequence is important, you can write a single callback function which contains subroutine calls in the order you want.

3.11.3 Passing Data To Callbacks

You can pass a single parameter to each function in a callback sequence. This parameter is called the *client data.* If you do not specify client data, SPARCworks/Visual supplies a *NULL* parameter. The client data must be of type *XtPointer.* If you want to pass a variable of a different type, you can pass its address cast to *XtPointer* as shown in Example B of Figure 3-24. Complex data can be passed by using the address of a structure.

All callback routines must return *void*. They can affect the application by modifying the client data structure or global variables.

3.11.4 Remove

The “Remove” button in the “Callback” text box removes the whole of the current callback list. Be careful when using this button as the operation cannot be undone.

3.11.5 Retain

The “Retain” toggle causes SPARCworks/Visual to retain the callback list you have typed for one widget when you select another widget in the hierarchy. This is a convenient way to assign the same callback list to several events of the same widget, or to each of several widgets. The steps for copying a callback list from one widget to another are:

1. **Add the callback list to the first widget.**
2. **Click on “Retain” in the callback dialog.**
3. **Select the second widget.**
4. **Click on the desired callback, such as “Activate” or “Arm”, on the “Callback” page of the second widget’s resource panel.**

This changes the variable name in the title of the callback dialog to that of the second widget and enables the “Add” button.

5. **Click on “Add” in the callback dialog.**

Callbacks from the first widget are not retained if the second widget already has a callback list.

3.12 Navigating in the Resource Panels

Because Motif has so many resources, SPARCworks/Visual uses multiple-page resource panels to display them on the screen. You may find it useful to refer to the *Widget Reference* chapter while you are becoming familiar with the structure of the resource panels. That chapter has a list of resources by widget class and page.

Some resources that are not available on the resource panel of a specific widget class can be found on the Core resource panel, which is discussed in the next section.

3.12.1 Settings

In general, any multiple-choice resource - one with a limited number of settings - is found on the “Settings” page, where you can set it with an option menu. All other resources that require you to type in a new value or call an additional dialog to set the new value are divided among the other pages. Except for the “Settings” page, resources are organized loosely by topic.

3.12.2 Display, Margins

Resources that affect the widget’s appearance are generally found on the “Display” page. These resources include text, colors, fonts and dimensions. The Core resource panel also includes some resources that affect the dimensions and the location of the widget.

Labels and Label derivatives have enough display resources to require a second page. The Label resource panel divides these resources into a “Display” page, containing color, font, text and pixmap resources, and a “Margins” page, containing size and margin width resources.

3.12.3 Keyboard, Callbacks

The “Keyboard” page lets you set keyboard mnemonics and accelerators for widgets that can have them. “Callbacks” lets you designate callback routines.

3.13 The Core Resource Panel

At the root of the Motif class hierarchy are widget classes called the *Core*, *Primitive* and *Manager* widgets. All widget classes are derived from the Core class and most are also derived from either the Primitive or the Manager class.

SPARCworks/Visual gives you access to resources for these broad superclasses via a single resource panel, the *Core resource panel*. To bring up the Core resource panel for a specific widget:

Select the widget.

Pull down the Widget Menu and select “Core resources...”,
or press <*Ctrl-C*>.

A page from the Core resource panel is shown in Figure 3-25.

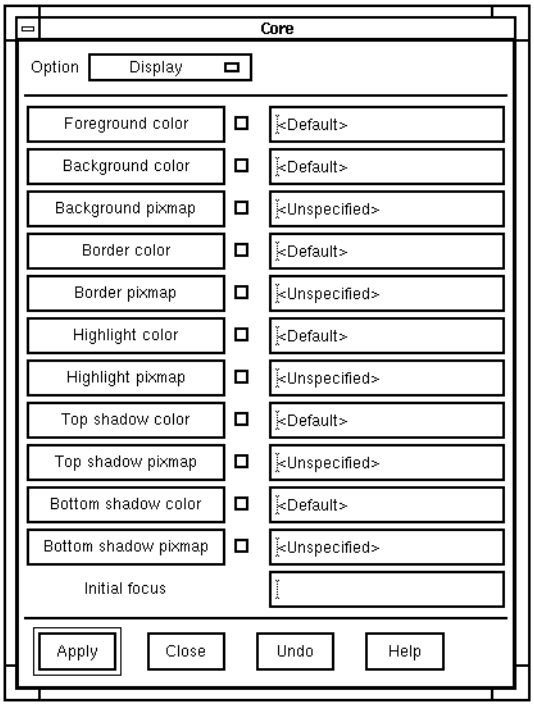


Figure 3-25 “Display” Page of the Core Resource Panel

3.13.1 Pages of the Core Resource Panel

The “Display” page of the Core resource panel has basic color and pixmap resources. For example, you can set the colors for the widget’s foreground, background, highlighting and shadows. You will do this in the *Other Editors* chapter.

The “Dimensions” page offers resources that affect the widget’s size and location on the screen. Class-specific dimension resources may override the settings on the Core panel. You may want to experiment with the effects of setting “Shadow thickness” on a TextField or “Highlight thickness” on a PushButton.

The “Code Generation” page gives you increased control over the generation of code. For example, you can designate a specific widget as static, local, or global and, if you are using C++, you can also designate it as private, protected, or public. These settings are discussed in the *Generating Code* chapter.

The “Settings” page offers miscellaneous multiple-choice settings which apply to most widget classes.

The “Callbacks” and “Drop site” pages offer settings related to advanced coding techniques and are discussed in the *Coding Techniques* chapter.

3.14 The Constraints Panel

Motif has two classes of widget, the PanedWindow and the Form, which are called *constraint widgets*. Widgets of these classes have a special set of resources, called *constraint resources*, that control the size and position of their children. A constraint widget maintains a separate set of constraints for each of its children. SPARCworks/Visual lets you set them as if they were resources of the children.

To view constraint resources for any child of a constraint widget:

1. **Select the child widget.**
2. **Pull down the Widget Menu and select “Constraints...”.**

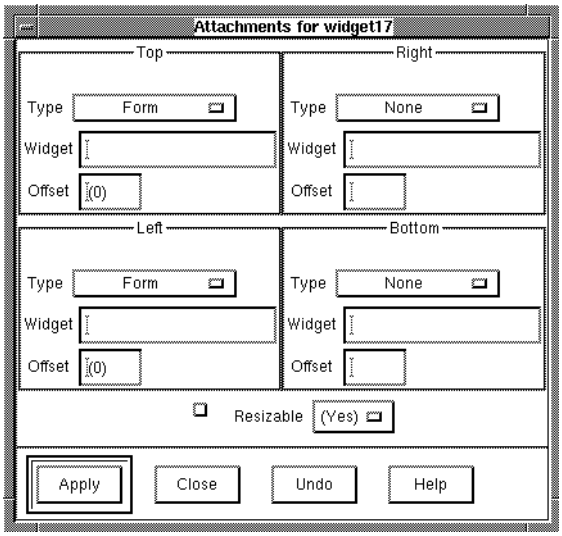


Figure 3-26 Constraints Panel

Figure 3-26 shows the default constraints resource panel for the Frame child of the Form in the tutorial layout. This panel shows that the Frame's top is attached to the top of the Form and its left side is attached to the left side of the Form, with an offset of 0 pixels. This is why it is located at the upper left corner of the Form. The bottom and right side of the Frame are unconstrained.

The constraints resources which the Form imposes on its children interact in complex ways and the preferred way of setting them is by using the interactive Layout Editor, which is discussed in the next chapter. The constraints resource panel is mainly useful for viewing the constraints which have been set, as an adjunct to the Layout Editor. Advanced users may want to set values on the panel itself. This can be done as with any other resource panel, by typing in the new value and clicking on "Apply".

The constraints resource panel for children of the PanedWindow displays a different set of values. This panel is discussed in the section on the PanedWindow in the *Widget Reference* chapter.

3.15 Default Resource Settings

You may have noticed that SPARCworks/Visual displays default resource settings in parentheses. Default settings are different from explicit settings, even if the values are the same when you build the interface. The difference is that default settings are not added to the generated code or X resource files. If your interface uses default settings, and is then run on a machine other than the one used to design it, it will use the Motif defaults for the machine on which it is running.

Many resources, such as the label on a PushButton or the number of columns in a RowColumn, are unlikely to cause portability problems. Others, such as dimensions and colors, are machine-specific. To make your interface portable, you must either use default values for such resources or put them into an X resource file at the time of code generation so they can be edited for each machine. This is discussed in the *Generating Code* chapter.

To revert to the default setting for a resource for which you have entered an explicit setting:

1. **Delete all text in the resource's text field on the resource panel.**
2. **Click on "Apply".**

To select the default value for a resource on the "Settings" page:

3. **Select the option displayed in parentheses from that option menu.**
4. **Click on "Apply".**

3.16 The Reset Command

When you set any resource, SPARCworks/Visual tries to apply that value to the selected widget in your dynamic display. Note that what you see in the dynamic display is a collection of widget instances. SPARCworks/Visual does not draw pictures of widgets but actually creates them, using the same Motif function calls which your interface will use when it is running. When SPARCworks/Visual sets a resource, it makes the appropriate Motif function call to set that resource's value for that widget.

Usually, the result of setting a value is the same as creating the widget with that value in the first place. However, this is not always the case. SPARCworks/Visual has a command on the Widget Menu, “Reset” (<*Ctrl-T*>), which destroys the selected widget and its children and recreates them with the most recently applied resource settings. If your layout does not look or behave as expected, try using the “Reset” command. The following steps demonstrate a case where “Reset” is required:

1. **Select the “Help” CascadeButton widget in the hierarchy.**
2. **Bring up its resource panel and select the “Keyboard” page.**
3. **Remove your previously set mnemonic by deleting all text from the “Mnemonic” text field.**
4. **Click on “Apply”.**

Notice at this point that the “Mnemonic” text field reverts to the <*Default*> setting. However, the “Help” button in the dynamic display still has an underscore under the “H”, indicating a mnemonic which is no longer present. To update the display:

5. **Select “Reset” from the Widget Menu.**

Resetting only affects the selected widget and its children. Resetting a widget that is low in the hierarchy may leave inaccuracies elsewhere in the dynamic display. If you set many resources, it is wise to reset the Shell to guarantee that what you see is what you get.

The “Reset” command is particularly useful when using the Form widget and its attachment resources. This topic is discussed in the *Layout Editor* chapter.

6. **Reinstate the mnemonic.**

3.17 Resource Settings Rejected

Motif and other widget toolkits have rules which control legal settings of resources and, since SPARCworks/Visual works with real instances of widgets and not simulations, any new resource setting that does not satisfy these rules is rejected. The rules include valid values for the particular widget, requirements of a parent widget such as a RowColumn and requirements of the machine you are using to build your design.

For example, if you are designing an interface for use on a large-screen workstation, you might want to set a dimension resource to a large number of pixels. If you are designing on a smaller-screen machine, you may find you cannot set the value you want even though the interface will run on the large-screen machine later. (In this situation, you could still set the width you want by using an X resource file.)

When Motif rejects a new resource setting, it does not revert to the previous setting but calculates a new value based on defaults and other resource settings in the hierarchy. This new value is reflected on the resource panel and in your dynamic display.

3.18 Where to Look for More Information

The *Widget Reference* chapter of this manual contains a summary of the most commonly used resources for each of the Motif widget classes. While this summary is necessarily brief, it will help you get started.

There are many books available that provide a more complete discussion of Motif widget resources. The *Motif Programming Manual* includes a summary which is both thorough and readable. Several other useful books are listed in the bibliography.

If you are using SPARCworks/Visual with additional widgets, you should also consult the documentation provided by your widget developer.

The Layout Editor 4

4.1 Introduction

The next step in designing your interface is to rearrange the widgets geometrically. The DialogTemplate automatically places the menu bar at the top of the window and arranges the buttons at the bottom. However, the arrangement of widgets inside the Form is up to you. Figure 4-1 shows the present appearance of the tutorial interface and how it will appear when you finish making the modifications in this chapter.

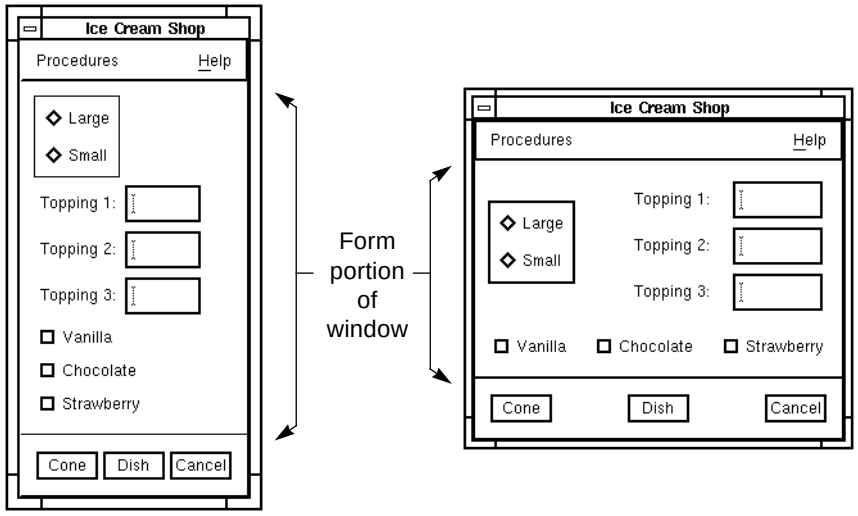


Figure 4-1 Default and Modified Layout for Tutorial Interface

The arrangement of widgets inside the Form is called the *layout*. To make these changes in the layout, you will perform the following steps with SPARCworks/Visual's interactive screen Layout Editor:

1. **Move widgets around to the approximate layout you want.**
2. **Attach the top and left side of the Frame to the sides of the Form at a fixed offset.**
3. **Attach the RowColumn to the Frame at a fixed offset.**
4. **Align the tops of the three ToggleButtons located at the bottom of the layout.**
5. **Set position attachments for proportional spacing of the three ToggleButtons.**

4.2 Concepts

Several classes of Motif widgets can impose geometric rules on their children. You have already seen that the children of a RadioBox, RowColumn and MenuBar are laid out in specific ways. In a MenuBar, the CascadeButtons are laid out in a single row. In a RowColumn, all children are laid out in a grid.

Three types of widget, Form, BulletinBoard and DrawingArea, allow more flexible layout of their children. These three widget classes are called *layout widgets*. SPARCworks/Visual provides an interactive screen Layout Editor for laying out the children of these widgets.

4.2.1 Attachments

Attachments are constraints that force a widget to be in a certain location relative to the layout widget or to another child of the layout widget. All three types of layout widget let you attach their children's upper left and lower right corners at any *x*,*y* location relative to the layout widget. If you constrain the upper left corner of a widget, you fix its location. If you also constrain its lower right corner, you also fix its size. These are the only attachments offered by the BulletinBoard and DrawingArea widgets.

4.2.2 Form Attachments

The Form widget lets you attach a side of a child widget to a side of the Form at a specified distance (measured in pixels). This has the effect of positioning the widget at a specific *x*,*y* location, like the attachments provided by the BulletinBoard and DrawingArea. In the case of the Form, this type of attachment is called a *Form attachment* to distinguish it from other types of attachments available for the Form.

4.2.3 Position Attachments

The Form also has *position attachments*, which are specified as a percentage of the total width or height of the Form. When you use position attachments, widgets get farther apart when the window gets larger so that the interface can take advantage of extra space when it is available.

4.2.4 Widget Attachments

Widget attachments let you attach two of the Form's children to each other at a specified absolute distance (measured in pixels). Attachments can be made edge to edge or top to bottom. Facilities are also provided for you to align and distribute a group of widgets.

These additional features of the Form allow you to design a layout which retains its appearance when the main window or an individual widget is resized. For example, by attaching a widget to both sides of the Form, you can make it stretch when the main window becomes larger. By attaching two widgets edge-to-edge, you can ensure that the spacing between them will be preserved even if one of them moves or changes size.

4.3 Displaying the Layout Editor

You can display the Layout Editor for any layout widget in your hierarchy by clicking twice on the widget. To edit the layout in your Form:

♦ **Click twice on the Form.**

This displays the dialog shown in Figure 4-2.

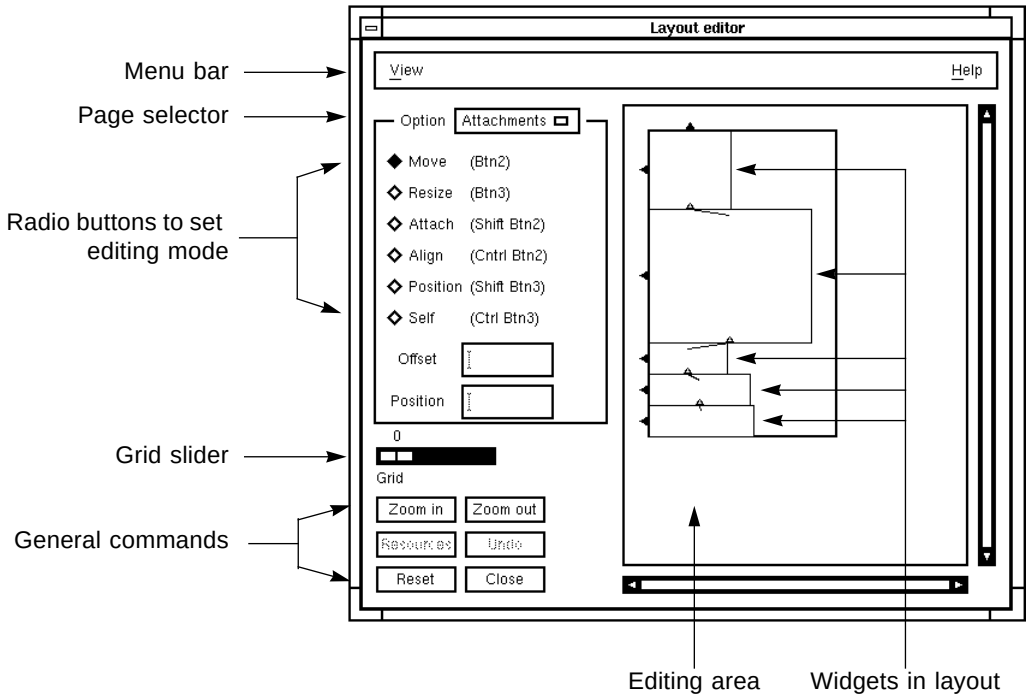


Figure 4-2 Layout Editor Dialog

4.3.1 Regions of the Layout Editor Dialog

The *editing area* displays a sketch of the layout. Widgets in the editing area are displayed schematically as boxes within a larger box which represents the Form. In Figure 4-2, the boxes represent, from top to bottom, the Frame, the RowColumn and the three ToggleButtons. The Layout Editor shows the Form's children but nothing lower on the hierarchy. For example, you cannot see the RadioBox and its ToggleButtons inside the Frame nor the Labels and TextFields inside the RowColumn. You also cannot see the MenuBar or the PushButtons, which are outside the Form.

4.3.2 Tip

The Layout Editor has its own menu bar and several command buttons. Some of the Layout Editor commands have keyboard accelerators. If you use them, be sure that the Layout Editor screen has the input focus as some of the same characters are also used as accelerators within the main SPARCworks/Visual window where they have different functions.

A set of radio buttons on the left side of the editor screen lets you select one of several *editing modes*. Selecting an editing mode assigns that function to mouse button 1. You can also use any mode, regardless of which is currently selected, by using the mouse button sequence indicated next to that radio button. For example, you can always use mouse button 2 to move a widget, or <*Shift-button 2*> to set an attachment.

4.4 Layout Editor View Menu

The View Menu of the Layout Editor provides two useful display commands.

4.4.1 Edge Highlights

This option highlights the widget edge closest to the pointer when the pointer is inside the widget. Any attachment you make is applied to the highlighted edge of the widget.

♦ **Pull down the "View" menu and select "Edge Highlights" (<*Ctrl-E*>).**

Move the pointer around the sketch of the Form and note that, whenever the pointer is inside the box representing a widget, the edge nearest the pointer is highlighted.

4.4.2 Annotation

This option displays an identifying string inside each box in the editing area. When you pull down the View Menu and select "Annotation", a pullright menu appears with a choice of "Widget names" or "Class names". "Widget names" (<*Ctrl-W*>) displays the variable name of each widget. "Class names" (<*Ctrl-N*>) displays the class of each widget, such as "Frame" or "RowColumn". Selecting one option disables the other. Selecting the same option when it is set removes all annotation from the display.

The annotation options are illustrated in Figure 4-3.

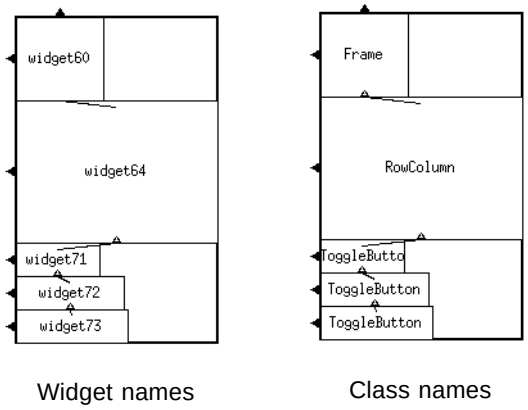


Figure 4-3 Annotation Options

- **Pull down the View Menu and select “Class names” from the “Annotation” submenu.**

4.5 General Commands

At the lower left of the Layout Editor screen are six pushbuttons for general commands and a slider for setting a grid.

4.5.1 Grid

To display a grid on your layout, use the grid slider to select a spacing from 2 to 50 pixels. The number of pixels is considered to be at 1:1 scale so that the grid will scale with the layout if you use the Zoom commands.

The “Move” and “Resize” modes snap to the grid if one is visible. When you move a widget, its top left corner snaps to the nearest grid intersection. Likewise, when you resize a widget, its lower right corner snaps to the grid.

You can disable the grid by setting the slider to zero.

- **Set the grid slider to 10 pixels.**

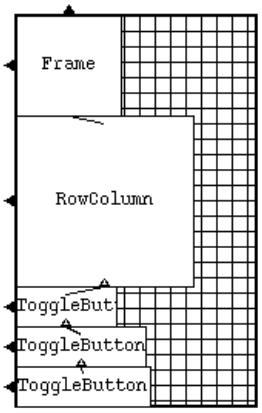


Figure 4-4 Tutorial Layout with 10 Pixel Grid

4.5.2 Zoom In, Zoom Out

Use “Zoom In” to increase the display scale and “Zoom Out” to decrease the scale.

4.5.3 Reset

“Reset” (<*Ctrl*>-*T*) does the same thing in the Layout Editor as it does in the main SPARCworks/Visual menu. It destroys the current instance of the selected widget, along with its children, and re-creates it in your dynamic display. When you are using the Layout Editor, the selected widget is always the Form or other layout widget and so “Reset” resets the layout widget and its children.

Sometimes new constraints are not reflected accurately in the dynamic display window until after you reset the layout widget. Therefore you should reset often when using the Layout Editor, especially if a change does not produce the result you expect.

If container widgets within a Form do not properly display after a reset, select a widget higher up the hierarchy and do another reset.

4.5.4 Resources...

This command brings up the resource panel for the layout widget. This command is important because the usual double-click on the layout widget's icon in the hierarchy brings up the Layout Editor and not the resource panel. You can also display the layout widget's resource panel with the "Resources" command on the Widget Menu.

4.5.5 Resize Policy

Most Form resources should be left at their default values. You could, however, consider resetting the "Resize Policy" resource to "Grow".

1. **Click on "Resources".**
2. **Select the "Settings" page of the resource panel.**
3. **Set "Resize policy" to "Grow".**
4. **Click on "Apply" and then "Close".**

If you do not set this resource, the Form always shrinks immediately to the minimum size when you move widgets around, which can be annoying. With a policy of "Grow", although you may sometimes have extra blank space in the layout, it goes away when you reset the Form.

4.5.6 Undo

The "Undo" button reverses the last operation done in the Layout Editor. Clicking on "Undo" repeatedly lets you step back through multiple operations. As with other Layout Editor commands, you may need to "Reset" before you can see the effect of "Undo".

4.5.7 Close

The "Close" button removes the Layout Editor window from the screen.

4.6 Understanding the Default Layout

The attachments are displayed in the Layout Editor using symbols as shown in Figure 4.5

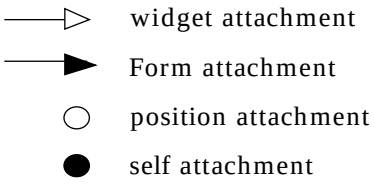


Figure 4-5 Symbols used in the Layout Editor

Your layout already has two kinds of attachments: Form attachments and widget attachments. Form attachments are shown as filled arrowheads on one edge of the widget. The Frame, which is the top widget in the default layout, has two such attachments - to the top and the left side of the Form. The other four widgets are each attached to the left side of the Form. All the widgets are stacked top to bottom in the order in which they were added to the hierarchy.

Each of the four lower widgets is attached to the widget above it. A widget attachment is shown as an arrow with a hollow arrowhead.

4.7 Rough Layout: The Move Mode

The “Move” command lets you drag a widget to a new location. This sets two Form attachments which fix the widget’s upper left corner at that point. The best way to start arranging any layout is to use this mode to place all the widgets approximately where you want them; you can then use the other modes to specify widget positioning and resize behavior more precisely.

To invoke the “Move” mode:

1. **Click on the “Move” toggle.**

Once in “Move” mode, you can drag widgets around in the layout with mouse button 1. You can also move widgets using mouse button 2, regardless of the current mode.

2. **Place the pointer inside the box corresponding to the RowColumn.**
3. **Hold mouse button 1 down and drag the RowColumn to the right until it overlaps the right edge of the Form.**

To do this, you have to drag the RowColumn so that its right margin extends beyond the edge of the Form. This is acceptable. When you release the mouse button, the Form automatically resizes to allow room for all its children.

4. **Drag the RowColumn up until its top is aligned with the top of the Frame.**

5. **Reset the Form.**

If the display is not correct, reset the Shell and then click twice on the Form to update the layout editor.

Figure 4-6 shows what your layout looks like now and the resulting dynamic display. The DialogTemplate automatically adjusts the size of the MenuBar and the spacing of the PushButtons to accommodate the new width of the Form.

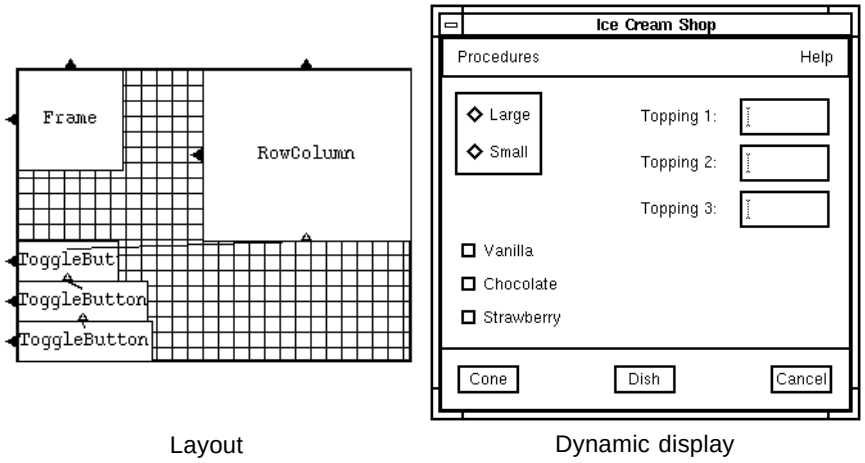


Figure 4-6 Rough Layout as Shown in the Layout Editor

4.7.1 How "Move" Mode Works

The tops and left edges of the RowColumn and Frame have filled arrowheads, indicating that they are attached to the top and left side of the Form. "Move" works by setting two attachments to the Form to fix the upper left corner of the widget at its new location. You can make these attachments either at a distance of zero (so that the widget touches the side of the Form) or at a

non-zero distance to produce a space between the widget and the Form. This distance is called the *offset*. In this case, the left edge of the RowColumn is attached to the Form at a non-zero offset. Offsets are discussed in a later section of this chapter. In the “Move” mode, SPARCworks/Visual calculates the offset based on the location of the pointer.

4.7.2 One Attachment Replaces Another

There can be only one attachment on each side of a widget. “Move” breaks any attachments that are already on those edges and then sets two new attachments on the top and left edges of the moved widget. In the default layout, the top of the RowColumn was attached to the bottom of the Frame. This attachment no longer exists because “Move” has set a new attachment on the top of the RowColumn. “Move” breaks the attachments made from a widget but preserves any attachments made to it from other widgets.

4.7.3 One Attachment Affects Another

Note that when you moved the RowColumn widget up, the three ToggleButtons moved with it. This happens because of the attachments between the widgets. Because the top of the first ToggleButton is attached to the bottom of the RowColumn, when you move the RowColumn up, the top ToggleButton also has to move. The other two ToggleButtons also move up in a chain reaction since the second is attached to the first and the third to the second.

The attachment from the ToggleButton to the RowColumn was not removed when you moved the RowColumn. This is because this attachment belongs to the ToggleButton and not to the RowColumn.This distinction is discussed later in this chapter.

4.8 Offsets

The tops of the RowColumn and the Frame are now lined up. However, there is an important difference in the way they are attached to the Form. The top of the Frame has a default attachment which was left over from the default layout. The RowColumn has a zero attachment which was set when you used the “Move” mode in Step 1 on page 82.

4.8.1 Default vs. Explicit Offsets

Default offsets are controlled by the “Horizontal spacing” and “Vertical spacing” resources of the Form. Explicit offsets override Default offsets through actions such as Move or Align in the Layout Editor. The Frame is 0 pixels from the top of the Form because the spacing resources are both 0 by default.

To see the effect of resetting the spacing:

1. **Click on “Resources”.**

On the Form resource panel:

2. **Select the “Display” page.**
3. **Double-click in the “Horizontal spacing” box and type:** `20`
4. **Double-click in the “Vertical spacing” box and type:** `20`
5. **Click on “Apply” and then “Close”.**

In the Layout Editor:

6. **Click on “Reset”.**

Figure 4-7 shows the results. All attachments with default offsets, including those on the Frame and the three ToggleButtons, now use the 20 pixel spacing. The RowColumn doesn’t move because its attachments were set with “Move” and have explicit offsets that do not refer to the spacing resources. The result is that the Frame and RowColumn are no longer aligned.

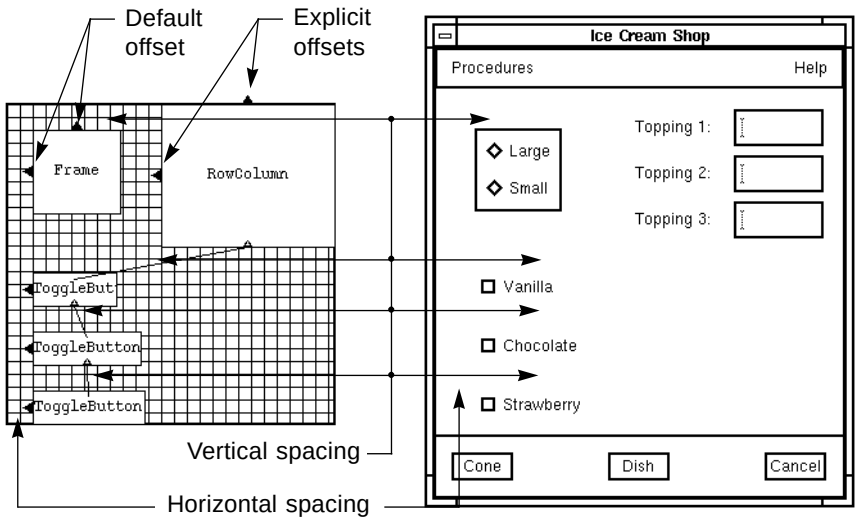


Figure 4-7 Effect of Resetting Vertical and Horizontal Spacing

The additional spacing between the ToggleButtons forces the entire Form to become larger. The DialogTemplate also resizes to accommodate the Form.

An advantage of default offsets is that they let the user control the amount of spread in the layout at run time. The main disadvantage of default offsets is that they require all spacings in your layout to be the same, which may not be what you want. Also, you should be careful not to confuse default and explicit offsets, since default offsets may change while explicit ones remain the same. For example, your Frame and RowColumn lost their alignment when you changed the spacing because one has an explicit offset and the other a default offset from the top of the Form. The Layout Editor screen does not distinguish between explicit and default offsets.

4.9 Attachments to the Form

You can attach a widget to the Form by dragging from just inside the widget's edge to a point just outside the side of the Form. This can be done with button 1 in "Attach" mode or with <*Shift-button 2*> in any mode. Attachments are set using the offset value in the "Offset" field. If the "Offset" field is empty, a default offset is used.

1. **Click on the “Attach” toggle.**

Note that when you are in the “Attach” mode or have <*Shift-button 2*> down, the pointer becomes a set of crosshairs.

Now replace some of the default attachments with attachments that use explicit offsets of 0 pixels.

2. **Click in the “Offset” box and type:** `0`

Attach the left edge of the Frame to the left side of the Form:

3. **Place the crosshairs just inside the left edge of the Frame so that the left edge highlights.**

4. **Hold down mouse button 1 and drag to a position just outside the left side of the Form.**

5. **Release the mouse button.**

The new attachment, like the old one, appears as a filled triangle on the side of the Frame. You can see its effect because the explicit 0 offset moves the Frame over to the side of the Form. If this does not happen, try setting the attachment again.

Do the same thing for the “Vanilla” ToggleButton:

6. **Place the crosshairs just inside the left edge of the top ToggleButton so that the edge highlights.**

7. **Drag with mouse button 1 to a position just outside the left side of the Form.**

8. **Release the mouse button.**

The ToggleButton moves over to the side of the Form.

The top of the Frame looks good at its present location. The 20 pixel offset from the top of the Form centers the Frame with respect to the RowColumn. However, this should be an explicit offset, not a default, so that it will remain constant if the Form spacing resources change. To change to an explicit offset, you must replace the attachment.

9. **Double-click in the “Offset” field and type:** `20`

10. **Place the crosshairs just inside the top of the Frame so that the edge highlights.**
11. **Drag with mouse button 1 to a position just outside the top of the Form.**
12. **Release the mouse button.**

    Figure 4-8 shows the result. If your dynamic display does not look the same

13. **Click on “Reset”.**

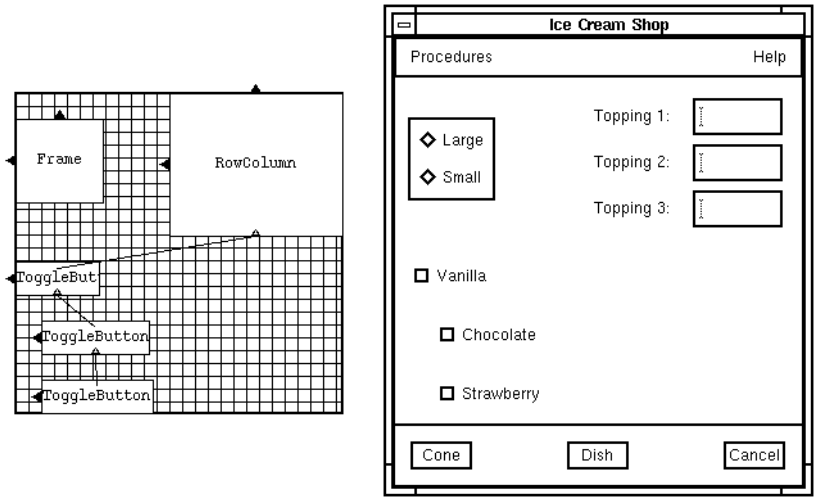


Figure 4-8 Form Layout with 20 Pixel Form Spacing

4.10 Attachments Between Widgets

Setting a widget attachment between two widgets in the Form is similar to attaching a widget to one side of the Form. To attach a pair of widgets, you simply drag the crosshairs from just inside one of the widgets to just inside the other.

4.10.1 Attaching Widgets Edge to Edge

To attach two widgets edge to edge, either touching or with an offset, you either attach the right edge of one to the left edge of the other, or the top of one to the bottom of the other.

You should still be in "Attach" mode from the last section. Attach the left side of the RowColumn to the right side of the Frame:

1. **Double-click in the "Offset" box and type**: `50`
2. **Position the pointer just inside the RowColumn's left edge so that the left edge highlights.**
3. **Hold down mouse button 1 and drag to a point just inside the Frame, until the right edge of the Frame highlights.**
4. **Release button 1.**

The new attachment appears as an arrow with a hollow arrowhead pointing from the center of the left edge of the RowColumn to the center of the right edge of the of the Frame. The RowColumn repositions itself with a gap of 50 pixels from the right edge of the Frame.

Although this does not change your layout much now, there is a significant advantage to this kind of attachment if the strings inside the Frame are likely to change. The RowColumn is now positioned relative to the right side of the Frame and not relative to the side of the Form. Even if the Frame grows, the RowColumn preserves the 50 pixel distance.

5. **Double-click on the first radio button in the radio box in the construction area.**
6. **Go to the "Display" page of the resource panel and change the button's label to** `Double Scooper`**.**

The results are shown in Figure 4-9.

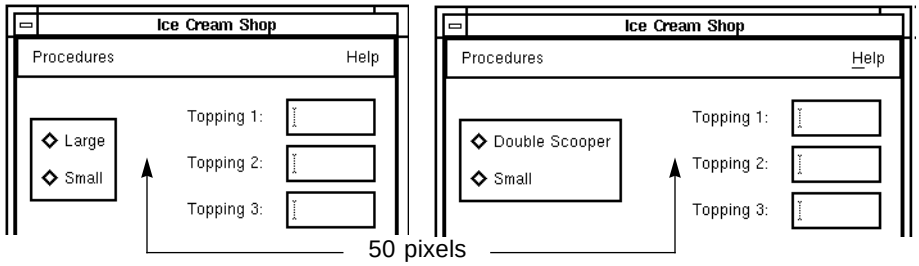


Figure 4-9 Frame Resize Behavior

4.10.2 Direction of Attachment

Attachments are not symmetrical. When you create an attachment by dragging the pointer from Widget A to Widget B, the attachment is said to *originate* from Widget A. To indicate this, SPARCworks/Visual draws an arrow from inside Widget A to Widget B. The attachment you have just made originates from the RowColumn.

Attachments apply only to the widget from which they originated. For example, the attachment you have just made constrains the RowColumn to a certain position relative to the Frame. If the Frame is moved or resized, the RowColumn moves in turn. If the RowColumn is moved or resized, the Frame is unaffected.

4.10.3 Attachments Affect Only One Coordinate

Note that the top ToggleButton has an attachment to the bottom of the RowColumn. This attachment is left over from the default layout and has the default offset. It is therefore controlled by the Form's spacing resources, which are still set at 20 pixels.

You were able to move the RowColumn away from the first ToggleButton because of the following rules:

1. Attachments on the top or bottom of a widget only affect its *y* coordinate.
2. Attachments on the left or right edge of a widget only affect its *x* coordinate.

The top of the first ToggleButton is still 20 pixels (the current vertical spacing) from the bottom of the RowColumn. Because there is no attachment between the ToggleButton's left or right side and the RowColumn, the RowColumn's position in the horizontal dimension has no effect on the ToggleButton.

Change the spacing of this attachment to 10 pixels:

1. **Double-click in the "Offset" field and type:** `10`
2. **Position the crosshairs just inside the top edge of the top ToggleButton so that the edge highlights.**
3. **Hold mouse button 1 down and drag to a position just inside the bottom edge of the RowColumn so that the edge highlights.**
4. **Release mouse button 1.**

The new attachment replaces the old one and the top ToggleButton adjusts its position.

4.10.4 Circular Attachments

The Motif rules for the Form widget prohibit you from attaching Widget A to Widget B and then also attaching Widget B to Widget A. This is called a *circular attachment*. Any larger attachment loop, such as attaching Widget A to B, B to C and C to A, is also considered circular and results in an error. If your layout contains such a loop, SPARCworks/Visual displays the error message shown in Figure 4-10 and you must break one or more attachments to eliminate the loop.

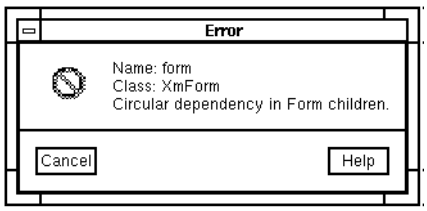


Figure 4-10 Circularity Error Message

Circularity is only a problem with widget attachments. Attachments to the Form and position attachments (see *Proportional Spacing: The Position Mode* section on page 102) cannot produce a circular attachment because these attachments can only originate from the child widget and not from the parent Form.

4.10.5 Method for Avoiding Circularity

A good method for avoiding circularity is to make all attachments between widgets point in only two directions, usually up and left. When you lay out your interface, start at the upper left corner and work down and to the right. Whenever you attach two widgets, make the attachment originate from the widget that is below or to the right. In this way, all the attachment arrows point the same way and you avoid accidental circular attachments.

4.10.6 Removing Attachments

Use any of the following methods to remove attachments in the Layout Editor:

1. Set a new attachment of any type on that edge of the widget. This removes and replaces the old attachment.
2. Use “Move” to reposition the widget. This removes all attachments that originated from it.
3. Using the “Attach” mode (<*Shift-Button 2*>), click just inside the widget on the edge where the attachment originates. This removes the attachment without setting a new one. The “Edge Highlights” mode can help you position the crosshairs properly.
4. Click on “Undo” to remove the last attachment added.

Motif requires that each widget have at least two edges attached. If you remove all attachments, the Form supplies simple Move-type attachments, based on the widget’s last location, when you reset.

4.10.7 Contradictory Attachments

You can specify attachments that contradict one another without being circular. For example, you might attach the left edge of a widget to the right side of the form using a positive offset. When you reset a Form that has contradictory attachments, Motif tries to calculate a layout that will satisfy all of them. If a satisfactory layout has not been found after a large number of iterations of a loop, the loop is broken and SPARCworks/Visual displays the error message shown in Figure 4-11. In these circumstances, some widgets may appear very small, or the Form itself may be resized very wide or long, until you remove the attachment that is causing the problem.

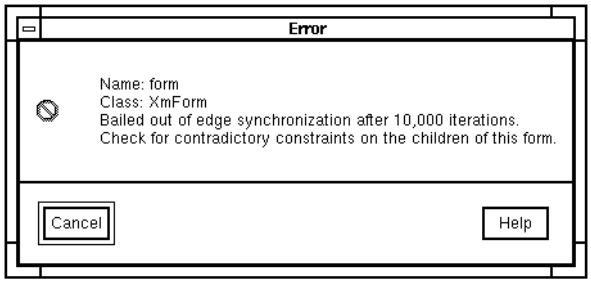


Figure 4-11 "Bailed out" Error Message

As with circular attachments, contradictory attachments must be removed before you can proceed.

4.10.8 Limit on Number of Attachments

A widget can have only one attachment originating from each of its four edges. That attachment can be of any type: a Form attachment, a widget attachment or a position attachment (see *Proportional Spacing: The Position Mode* section on page 102). Whenever you specify a new attachment originating from one edge of a widget, it replaces any attachment that was already there.

There is no limit to the number of attachments *to* a widget, provided they all originate from other widgets.

4.11 Aligning Widgets: The Align Mode

You can align the tops of two widgets by attaching them top to top with an offset of zero. An easy way to do this is to select the “Align” mode of the Layout Editor. The “Align” mode is simply an attachment with an explicit zero offset. You can also align a pair of widgets on any other edge: bottom to bottom, left to left or right to right. Be careful to avoid circularity when you use this feature.

To align the tops of the first two ToggleButtons:

1. **Use “Move” (button 2) to move the bottom two ToggleButtons into a rough horizontal row.**

Do not move the top ToggleButton.

So that you can see the effects of “Align”, and later on “Distribute”, deliberately leave the tops of the widgets and the spacing of their edges, slightly uneven, as shown in Figure 4-12.

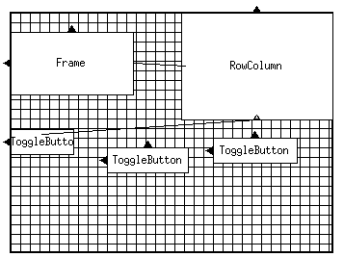


Figure 4-12 ToggleButtons Before Alignment

2. **Click on the “Align” toggle.**
3. **Position the crosshairs just inside the top of the middle ToggleButton.**
4. **Hold mouse button 1 down and drag to a position just inside the top of the first (left) ToggleButton.**
5. **Release the mouse button.**

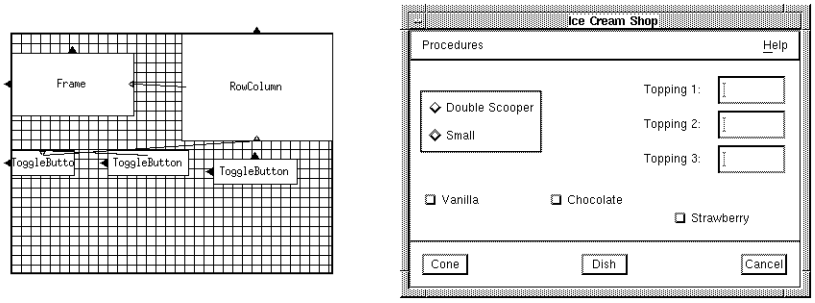


Figure 4-13 After Aligning First Two ToggleButtons

4.11.1 How Alignment Works

To understand how this works, remember that an attachment affects a widget's position in only one dimension. For example, if you attach the top of Widget 1 to the bottom of Widget 2:

```
top_1 = bottom_2 + offset
```

where *top_1* represents the *y* coordinate of the top of Widget 1 and *bottom_2* the *y* coordinate of the bottom of Widget 2. This is true whether or not the two widgets overlap in the horizontal dimension.

Example A
Top of B2 attached to bottom of B1, offset 10. Effect: Top edge of B2 is 10 pixels below bottom edge of B1.

Example B
Same widget attachment as in Example A; widgets do not overlap in x dimension.

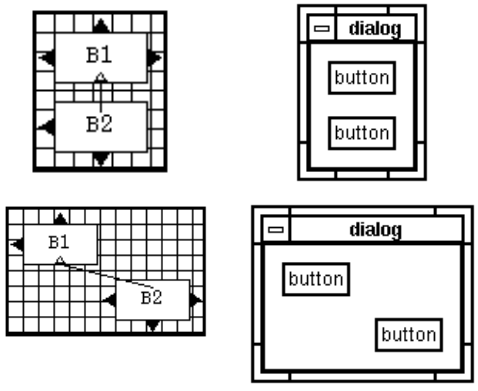


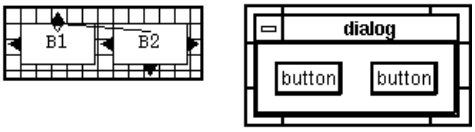
Figure 4-14 Attachment of Two Widgets Top-to-Bottom

Similarly, if you attach two widgets top to top:

```
top_1 = top_2 + offset
```

If the offset is 0, the tops of the two widgets have the same *y* coordinate; that is, they are aligned.

Example A
Top of B2 attached to top of B1, offset 0. Effect: Tops of widgets are aligned.



Example B
Top of B2 is attached to top of B1, offset 10. Effect: Top edge of B2 is positioned 10 pixels below top of B1.

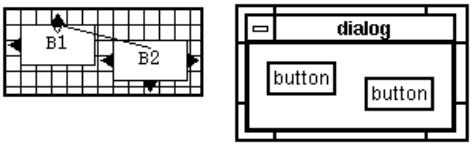


Figure 4-15 Attachment of Two Widgets Top-to-Top

Example A of Figure 4-15 shows the type of attachment used by the “Align” mode. Example B shows a similar attachment with a non-zero offset. The effect is similar to an alignment but with a step effect.

4.12 Group Align

You can align widgets in pairs using either the “Align” mode or by using the “Attach” mode with a 0 offset to attach their left, right, top or bottom edges. The “Group Align” feature is a quick way to align a group of widgets. The attachments set by this feature are the same kind used to align widgets in the “Attach” or “Align” mode.

To see the effects of this feature:

1. **Use “Move” mode or mouse button 2 to move the second ToggleButton back out of alignment.**
2. **Use “Move” mode or mouse button 2 to move the right ToggleButton so that its right edge is aligned with the right edge of the Frame.**

Now align the tops of the three ToggleButtons as a group:

3. Select the “Group align” page of the Layout Editor.

The left side of the Layout Editor displays the menu shown in Figure 4-16.

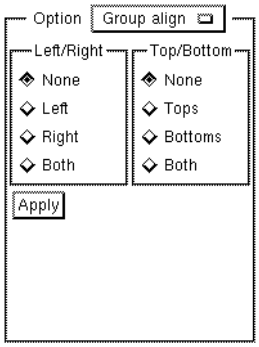


Figure 4-16 “Group align” Options

4. On the Layout Editor screen, click on the right ToggleButton.

Widgets highlight as you click on them to indicate that they are selected. After you click on the first widget, you must use <*Shift-Button 1*> to add widgets to the group. On color monitors, the last widget selected is highlighted in a different color from the others.

5. Click with <*Shift-Button 1*> on the middle ToggleButton.

6. Click with <*Shift-Button 1*> on the left ToggleButton.

To align the tops of the ToggleButtons:

7. Click on “Tops” in the “Top/Bottom” column.

8. Click on “None” in the “Left/Right” column.

9. Click on “Apply”.

This sets the attachments shown in Figure 4-17.

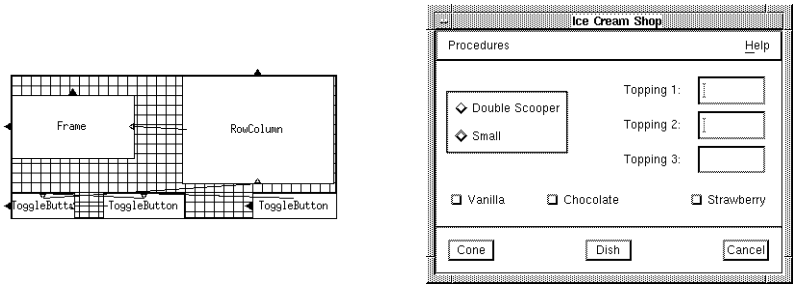


Figure 4-17 Layout After Aligning the Three ToggleButtons

4.12.1 Direction of Attachments Set by Group Alignment

When you align a group of widgets, the last widget selected is unaffected and the others are aligned to it. The order in which widgets are selected does not matter except for the last. SPARCworks/Visual sets attachments in the order of the widget's spatial positions. Each widget in the group except the last is attached to its neighbor and all attachments point toward the last widget selected. In your layout, the right widget is attached to the center widget, which is attached to the left widget.

Figure 4-18 illustrates this general rule. If a group of widgets is selected in the order shown in the first figure, the resulting attachments would connect them in the order shown in the second figure.

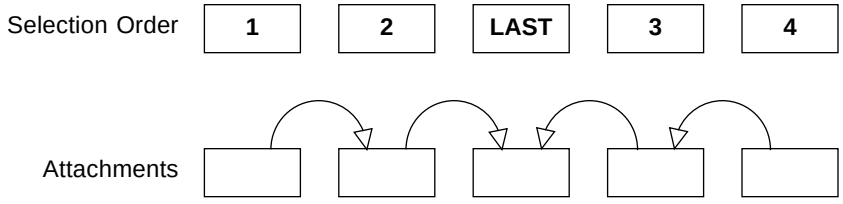


Figure 4-18 Direction of Attachments Set by Group Align

This rule works similarly for columns of widgets aligned along their left or right edges.

As when setting other attachments, be careful to avoid circularity in aligning groups of widgets. The best method is to position the top or left-most widget where you want it and then align other widgets to it, selecting them from right to left or from bottom to top.

4.13 Distribute

The “Distribute” feature lets you distribute a group of widgets evenly across a given space. You can use this feature to distribute the three ToggleButtons evenly across the bottom of the Form.

1. Select the “Distribute” page of the Layout Editor.

This page displays the options shown in Figure 4-19.

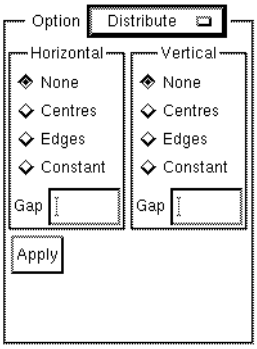


Figure 4-19 “Distribute” Options

In either the “Centers” or “Edges” mode, SPARCworks/Visual spaces all the selected widgets evenly across the space between the outside edges of the two that are farthest apart. In the “Centers” mode, widgets are spaced so that their centers are equidistant. In “Edges” mode, the widgets are positioned with equal space between their edges.

Distribute the three ToggleButtons with equal space between their edges:

2. Click on the right ToggleButton.

If widgets are still selected from the previous section, they are deselected when you click on a new widget to start a new group.

3. **Click with <*Shift-Button 1*> on the middle ToggleButton.**

4. **Click with <*Shift-Button 1*> on the left ToggleButton.**

The three buttons are now all highlighted. If you are using a color screen, the left one is highlighted in a different color from the other two. However, the order of selection does not matter to the “Distribute” feature.

Distribute the buttons horizontally with equal space between their edges:

5. **Click on “Edges” in the “Horizontal” column.**

6. **Click on “None” in the “Vertical” column.**

7. **Click on “Apply”.**

The result is shown in Figure 4-20.

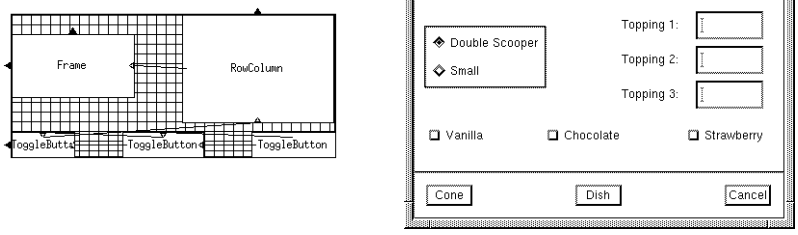


Figure 4-20 Distribution of Widget Edges

The other option on this page, “Constant”, sets attachments from widget to widget using a constant offset specified in the “Gap” field. If the “Gap” field is empty, the default Form spacing is used. Using the “Constant” mode is exactly the same as setting multiple attachments in “Attach” mode. In this mode, the total space occupied by the group of widgets may change.

4.13.1 Direction of Attachments Set by “Distribute”

“Distribute” sets attachments in a different order from “Group align”. In “Distribute”, attachments are always made in spatial order from bottom to top or from right to left, as shown in the Figure 4-21. Each widget is attached to its neighbor. In “Distribute”, unlike “Group Align”, the order of widget selection makes no difference.

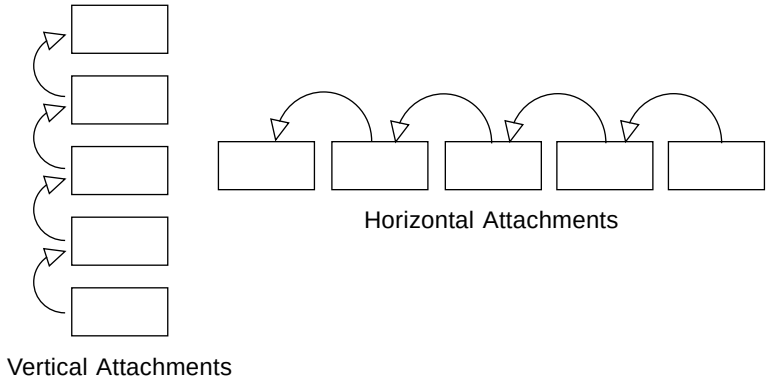


Figure 4-21 Direction of Attachments Set by “Distribute”

You can see in the Layout Editor screen that the arrows attaching the ToggleButtons edge to edge all point from right to left. It is easier to see the arrows if you pull down the View Menu and temporarily remove class name annotations from your screen.

4.13.2 Tip

Because “Distribute” only sets attachments as described above, circular attachments can easily result unless you plan ahead. For example, you can select widgets from right to left, “Group Align” them and then “Distribute” them, as you have just done. However, you cannot select them from left to right, “Group Align” them and then use “Distribute” on the same group of widgets. Doing this results in circular attachments.

If you make all other attachments from right to left or from bottom to top, “Distribute” never results in circularity.

4.14 Proportional Spacing: The Position Mode

Position attachments let you attach an edge of a widget at a position that is a percentage of the Form's total width or height. This capability lets the widgets in your interface take advantage of additional space when the window resizes. Positions are always measured from the top left corner of the Form. If the top or bottom edge of a widget has a position attachment of *n%*, that edge is positioned *n%* of the way down from the top of the Form. If the left or right edge of a widget has a position attachment of *n%*, that edge is positioned *n%* of the way across from the left edge of the Form.

The position is specified as a percentage value in the "Position" field. If you do not enter a value, SPARCworks/Visual assumes a value of zero. Position attachments are shown as hollow circles on the attached edge of the widget.

First, demonstrate the current window behavior:

1. Resize the window so it is wider than the present size.

Note that the RowColumn stays at the same distance from the Frame, as shown in Figure 4-22.

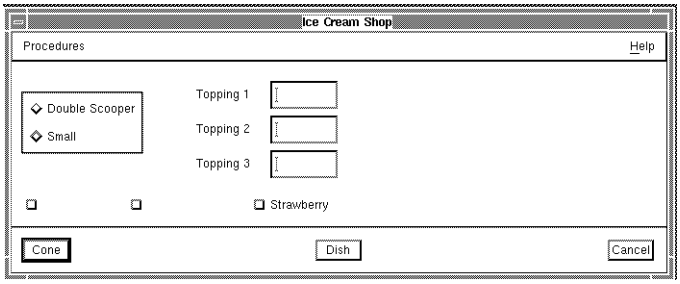


Figure 4-22 Behavior of RowColumn with Fixed Offset Attachment When Window is Resized

This is the behavior you expect when you set an attachment with a fixed offset. Any extra window width is just unused space. Many interfaces use this resize behavior. However, you can also use a position attachment to make the RowColumn move over to take advantage of available window space.

To do this, specify a 45% position attachment on the left edge of the RowColumn:

2. **Select the “Attachments” page.**

3. **Select “Position” mode.**

4. **Double-click in the “Position” box and type:** 45

5. **Position the pointer just inside the left edge of the RowColumn so that the edge highlights and click.**

The position attachment appears as a hollow circle on the edge of the RowColumn, replacing the existing attachment and the arrow that represented it. This type of attachment places the RowColumn’s left edge 45% of the distance across the Form, regardless of the window size. To see the effects of this:

6. **Resize the window narrower, then wider.**

The RowColumn now moves right to fill any extra space, as shown in Figure 4-23. This type of resize behavior is the main advantage of position constraints.

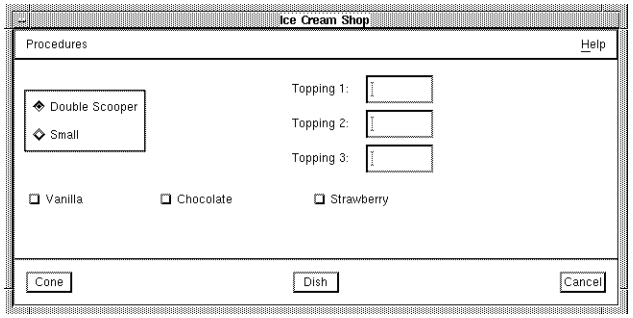


Figure 4-23 Behavior of RowColumn with Position Attachment When Window is Resized

The disadvantage of this type of attachment is that a position attachment is calculated only by the size of the Form and does not adjust to fit the sizes of other widgets. This is a problem if other widgets resize, as shown in Figure 4-24. When one of the labels inside the Frame becomes longer, the Frame can get closer to the RowColumn, or even overlap it.

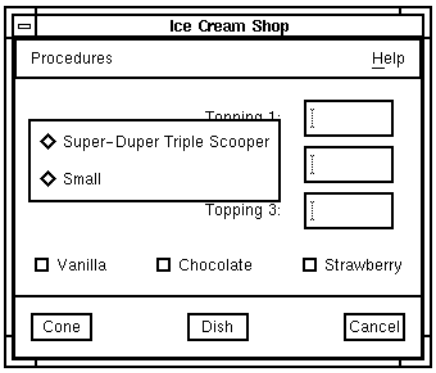


Figure 4-24 Behavior of RowColumn with Position Attachment When Another Widget Resizes

Compare this behavior to that shown in Figure 4-9, where the layout had a widget attachment with a 50-pixel offset. The choice of attachment type is up to you and should be based upon the types of widgets in your layout and any possible changes at run time. The topic of layout strategy is discussed further in the *Advanced Layout* chapter.

4.15 Self Mode

The “Self” mode is another way of setting a position attachment. Instead of typing a percentage value, you click on one edge of the widget and SPARCworks/Visual calculates a percentage based on the widget’s present location and the present size of the Form. When you use “Self”, you do not have to specify a percentage in the “Position” field and any value that is already in the field is ignored. You must, however, first place the widget where you want it to be using “Move” or one of the other commands.

“Self” works especially well in combination with “Distribute”. You have already set up the buttons at the bottom of the Form with a fixed gap between them. By setting “Self” attachments on both sides of each button, you can preserve the evenness of spacing while letting the buttons take advantage of extra window space that may become available.

By default, “Self” attachments snap to the grid. Therefore, in order to take advantage of the precise spacing set by “Distribute”, you should disable the grid.

1. **Set the grid slider to 0.**

2. **Click on the “Self” toggle.**

In “Self” mode, positions are calculated relative to the total size of the Form. Since you have been changing the window size, you should:

3. **Reset the Form.**

Resetting the Form calculates its best size based on the attachments currently set on its children.

4. **Click on the right edge of the right ToggleButton.**

The “Self” attachment appears as a filled circle on the edge of the ToggleButton.

5. **Click on the left edge of the right ToggleButton.**

6. **Click on the right edge, then the left edge, of the middle ToggleButton.**

7. **Click on the right edge, then the left edge, of the left ToggleButton.**

“Self” attachments appear as filled circles. When you have clicked on all six edges, your layout looks like Figure 4-25.

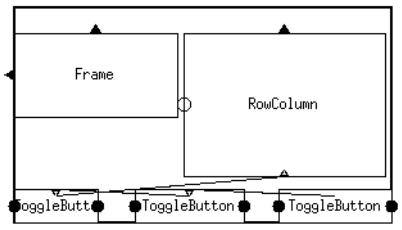


Figure 4-25 “Self” Attachments

In Motif, a “Self” attachment is the same as a position attachment and therefore SPARCworks/Visual cannot tell them apart after you reset the Form. In this event, “Self” attachments appear as hollow circles and behave exactly like Position attachments.

8. **Reset the Form.**

9. **Save your design.**

This concludes the tutorial portion of this chapter. The rest of this chapter discusses some additional layout features.

4.16 The Resize Mode

"Resize" works like "Move" but sets attachments on the bottom and right side of a widget. To use "Resize":

1. **Click on the "Resize" toggle.**
2. **Using mouse button 1, drag the lower right corner of a widget to the position you want.**

"Resize" is useful with BulletinBoards and DrawingAreas if you want to fix the size of a widget. In a Form, "Resize" works by attaching the lower right corner of the widget to a specific *x,y* location relative to the upper left corner of the layout widget. When combined with attachments on the upper left corner of the widget, this fixes the widget's size. In a BulletinBoard or DrawingArea, "Resize" simply sets the width and height resources of the widget.

This option is not normally used with Form layouts, because most widgets behave better when you let them calculate their own best size. Figure 4-26 shows a typical example of how widgets can resize themselves.

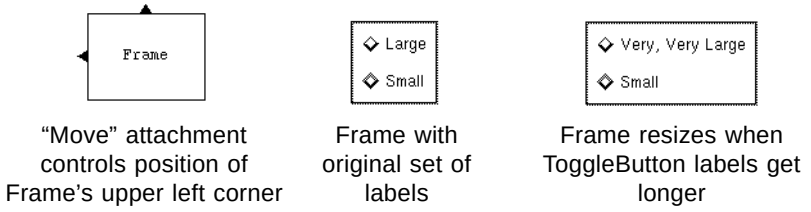


Figure 4-26 Frame Widget Resize Behavior

The Frame in Figure 4-26 is constrained by a "Move" command only. If the user changes the label text for one of the ToggleButtons, the Frame is free to resize itself because its bottom and right sides are unconstrained.

If the Frame also has attachments set by "Resize", however, it cannot resize, as shown in Figure 4-27.

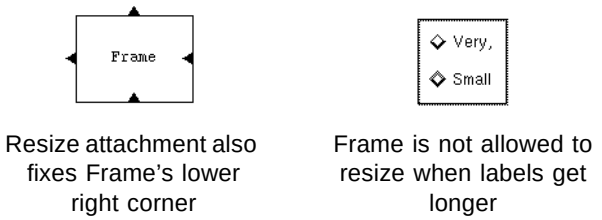


Figure 4-27 Effects of Using Move and Resize Attachments Together

Because the combination of "Move" and "Resize" attachments shown in Figure 4-27 fixes all four sides of the Frame, it cannot subsequently expand to accommodate a larger label. Motif handles this situation by displaying only part of the label string.

"Resize" is used mainly with the BulletinBoard and DrawingArea as these widgets do not offer position attachments or widget attachments. "Resize" offers a way to force widgets to remain at a certain size and prevents them from overlapping. The disadvantage of "Resize" is that it eliminates the positive effects of automatic resizing. To get the best behavior with widgets that are likely to change size, use a Form and attach widgets to one another so that when one widget changes size other widgets move to accommodate it.

4.17 Using the Constraints Panel

The constraints panel, which was introduced in the previous chapter, can be used to view attachments on any child of the Form and to adjust attachments. The constraints panel is only recommended for viewing or fine-tuning attachments. Note that the constraints panel only shows attachments that originate from a widget. Use the Layout Editor for any large-scale changes.

1. **Click on "Close" on the Layout Editor screen.**
2. **Select the RowColumn in the construction area.**
3. **Pull down the Widget Menu and select "Constraints".**

This command displays the attachments set on the RowColumn, as shown in Figure 4-28.

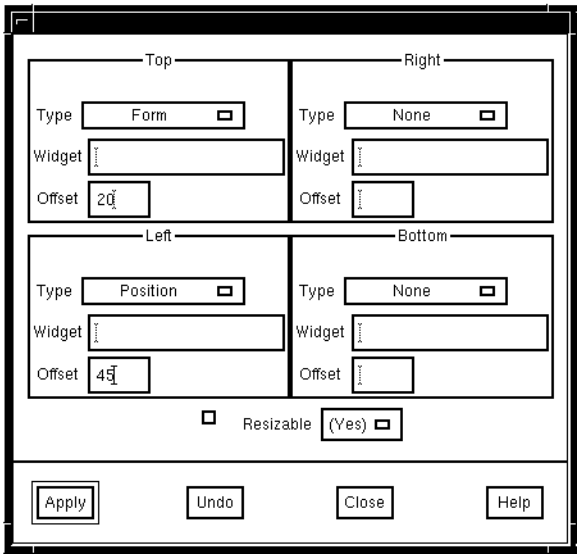


Figure 4-28 Constraints Panel for RowColumn

The top edge of the RowColumn has an attachment to the Form with an offset of 20. The left edge of the Form has a position attachment of 45%. The right and bottom edges of the RowColumn have no attachments.

You can use the constraints panel to:

- View the attachments on any child of a Form
- View the names of widgets to which each edge of the widget is attached
- Distinguish default offsets (shown in parentheses) from explicit offsets
- Change the type of attachment - “Form”, “Position”, or “Widget” - by selecting from the “Type” option menu
- Remove an attachment by selecting “None” from the “Type” menu (only if the widget still has at least one attachment in *x* and one in *y*)
- Adjust the offset or position by typing a new value into the “Offset” field

4. To effect any changes in the constraints panel, click on “Apply”.

4.18 Other Layout Widgets

The Layout Editor works in much the same way on the BulletinBoard and DrawingArea as on the Form. There are a few differences when you use it with other layout widgets. The following comments refer only to the BulletinBoard and DrawingArea.

As mentioned above, widget, self and position attachments are not available. The Layout Editor does not show arrows or arrowheads indicating attachments, but only the positions and sizes of widgets in the layout.

When you first display the Layout Editor, you may find that several widgets are initially laid out directly on top of one another. Use “Move” repeatedly to drag them to new positions so that they do not overlap.

The only available editing modes on the main screen are “Move” and “Resize”. You can also use “Group Align” and “Distribute” but not the “Align” mode on the main screen. “Group Align” and “Distribute” do not attach widgets to one another, but merely reposition them.

There is no danger of circular attachment as widgets cannot be attached to each other.

Internally, the Layout Editor does not set resources of the BulletinBoard and DrawingArea as it does for the Form. Instead, it determines layout by setting Core size and position resources of the child widgets. The constraints panel is not available for these layout widgets. To view the size and position resources, display the Core resource panel for the child widget.

4

Other Editors 5

5.1 Introduction

SPARCworks/Visual provides special editors which simplify certain common tasks. You can use these editors to perform the following tasks:

- Setting colors
- Setting fonts for text strings
- Using bitmaps or pixmaps for labels instead of text
- Creating pixmaps

All of these editors are used in this chapter to customize the tutorial interface.

SPARCworks/Visual also has an editor for creating compound strings, which is described in *The Compound String Editor* chapter.

5.2 Setting Colors

Because foreground and background colors are the basic elements of all widgets, these resources are located on the Core resource panel. Use the following instructions to set these colors for widgets in the hierarchy.

1. **Select the “Strawberry” ToggleButton icon in the hierarchy.**
2. **Click on the “core resources” button on the toolbar or pull down the Widget Menu and click on “Core resources...”.**

In previous chapters, you entered settings for resources directly in the text boxes on the right side of the resource panels. To use the editors described in this chapter, click on the buttons on the left side of the resource panels instead. These are PushButtons which display other editors when you click on them.

3. On the “Display” page, click on the “Foreground color” PushButton.

SPARCworks/Visual displays the Color Selector, shown in Figure 5-1:

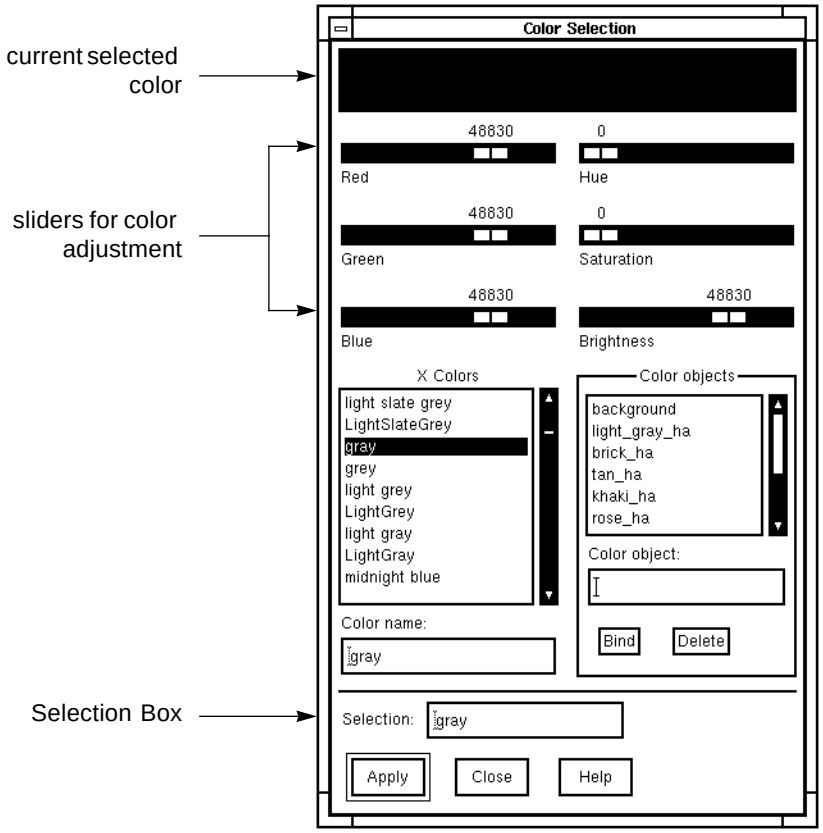


Figure 5-1 The Color Selector

5.2.1 Selecting from the X Color List

X provides many pre-named colors. These standard colors make a good starting point for selecting colors for the interface.

1. **Scroll down through the X colors displayed in the scrolled list.**

2. **Select a color.**

Choose a dark reddish color such as maroon. These colors are clustered about half-way down the list. The selected color is displayed at the top of the Color Selector.

3. **Click on “Apply” in the Color Selector.**

This applies your selection to the “Foreground color” resource on the Core resources panel. To apply it to the widget, you must:

4. **Click on “Apply” in the Core resources panel.**

The “Strawberry” ToggleButton changes color in the dynamic display. Don‘t worry if it looks like the background color changes instead of the foreground color; this is because the widget is selected in the hierarchy and the selection is reflected in the dynamic display by inverting the foreground and background colors.

To see the true colors:

5. **Select the Shell in the hierarchy.**

Now you can see the true foreground and background colors of the “Strawberry” ToggleButton in the dynamic display.

5.2.2 Using Color Components

Selecting from the X color list is only one of several ways to specify a color. Another method is to create the color using components. Use this technique to set the background color of the “Strawberry” ToggleButton:

1. **Select the “Strawberry” ToggleButton in the hierarchy.**

2. **On the Core resource panel, click on “Background color”.**

3. **Use the sliders to change the color.**

The sliders at the top of the Color Selector let you individually control the rgb components, hue, saturation and brightness of your color. You can use the sliders in any order. Changes are reflected immediately at the top of the Color Selector. Notice that the color name is a concatenation of values.

Since this is a background color, a light (non-saturated) color is recommended. This provides a good contrast for the labels, which are darker.

4. **When you are satisfied with the color, apply it by repeating Step 3 and Step 4 in Section 5.2.1.**

Do not change the color in the Color Selector before proceeding to the next section.

5.2.3 Color Objects

To create a visually appealing interface, it is essential to use colors consistently. SPARCworks/Visual assists you by providing *color objects*, which let you bind colors to names which you specify. Use this feature to apply the same background color to all the widgets in the central part of the tutorial interface:

1. **Select the "Strawberry" ToggleButton in the hierarchy.**
2. **In the Core resource panel, click on "Background color".**
3. **In the Color Selector, double-click in the "Color object" box.**
4. **Type**: `background`
5. **Click on "Bind".**

This binds the color to a color object named "background". The selection box shows the name *background* in angle brackets.

6. **Click on "Apply" in the Color Selector and in the Core resources panel.**

Apply this background color to other widgets in the hierarchy by entering the color object name as the setting for the background color resource:

7. **Select the "Vanilla" ToggleButton in the hierarchy.**
8. **In the Core resource panel, double-click in the "Background color" text field.**
9. **Type**: `<background>`

Note that you must use angle brackets. The angle brackets distinguish an object name from a string value. For example, a color object named <*red*> is distinct from the color *red*.

10. Click on “Apply”.

The color in the “Vanilla” ToggleButton changes. Now apply this background color to the “Chocolate” ToggleButton and any other widgets you want to share the same background color.

5.2.4 Rebinding Color Objects

When you use a color object, you can easily change the color on all widgets which reference that color object. This makes experimenting with colors easy.

1. In the Color Selector, click on “background” in the “Color objects” list.

The background color is displayed at the top of the Color Selector and “background” is displayed in the “Color object:” text field.

2. Using the sliders at the top of the Color Selector, change the color.

3. Click on “Bind”.

All the resources which refer to the color object change to the new color and the change is reflected immediately in the dynamic display.

Experiment with creating new colors, binding them to color objects and assigning these color objects to some color resources. You can also bind colors from the X colors list to color objects. By repeating these steps, you can build your own palette of colors. Remember that it is better to name a color object for the function it serves, such as “background”, than for the color it represents, since the color can change.

Color objects are saved with the design file. This means you can use the same names for color objects in different design files, even though the colors might be different. For example, the color object, *background*, might be yellow in one design and light blue in another. Within the same design, however, an object name such as “background” always refers to the same color.

5.3 Setting Fonts

The Font Selector lets you select font styles and sizes for the text which appears in your widgets. Your system determines which font styles are available; you can't create new fonts the way you can create new colors.

So far, you have used the default font for all text in the tutorial interface. In this section you will use the Font Selector to:

- Select a font
- Set a font on a single widget
- Bind a font object to a particular font
- Apply a font object to multiple widgets

This section discusses the use of a single font in a text string. Complex font objects which let you use multiple fonts in a single label string are discussed in *The Compound String Editor* chapter.

5.3.1 Selecting a Font

To add more visual interest to the tutorial interface, you are going to change some of the fonts from their default to an oblique font, as illustrated in Figure 5-2.

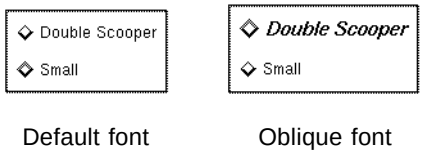


Figure 5-2 Toggle Buttons Before and After Setting Font

First, bring up the Font Selector:

1. **Click twice on the "Double Scooper" radio button to bring up the resource panel.**
2. **On the "Display" page of the resource panel, click on "Font".**

SPARCworks/Visual displays the Font Selector, shown in Figure 5-3.

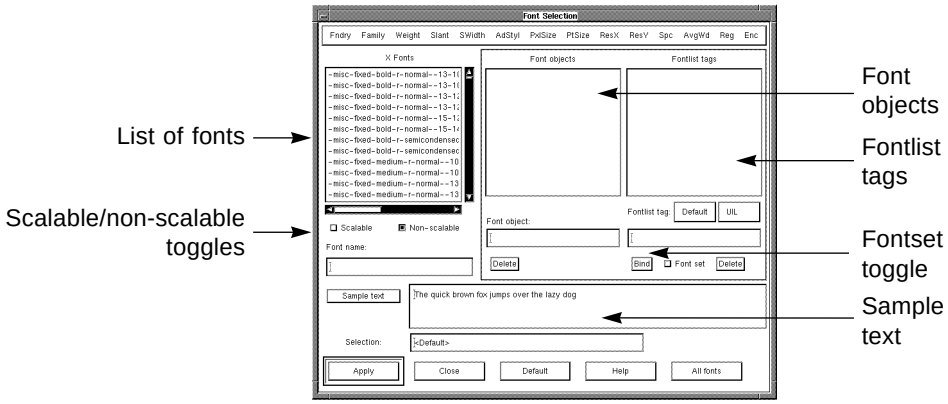


Figure 5-3 The Font Selector

5.3.2 Regions of the Font Selector Panel

The Font Selector lists all the fonts available on your system. Because different machines may have different fonts installed, your list may look different from the figure.

Since there can be hundreds of fonts in the list, the menu bar lets you filter the list according to different criteria such as the font family, weight and point size.

The toggles below the font list let you select scalable fonts, non-scalable fonts, or both.

When you select a font from the list, the name appears in the “Font name” and “Selection” fields.

The Font Selector also lists font objects and their associated fontlist tags. Simple font objects can be used as aliases for font names. *The Compound String Editor* chapter discusses the use of more complex font objects.

5.3.3 Filtering the Font List

You are going to change the font of the “Double Scooper” label to 14-point bold oblique Helvetica. (If this font is not installed on your machine, select an alternative.) Because there are so many fonts in the list, it helps to filter the list. The “Font name” and “Selection” fields reflect your choices as you make them.

1. Pull down the Family Menu from the menu bar and select “Helvetica”.

This eliminates all fonts from the list which are not Helvetica.

2. Pull down the Weight Menu and select “bold”.

3. Pull down the Slant Menu and select “o” for “oblique”.

4. Pull down the PtSize (point size) Menu and select “140”.

Note that, because point sizes are specified in tenths of a point, this selects a 14-point font.

These steps reduce the font list to about four entries. A typical font name entry is shown in Figure 5-4. Although many fields are required to specify a font uniquely, most of them are only interesting to advanced typographers. The fields you are most likely to use are family, weight, slant and point size. You may also need to specify the display resolution such as 75 dpi (dots per inch) or 100 dpi. These fields are identified in Figure 5-4.

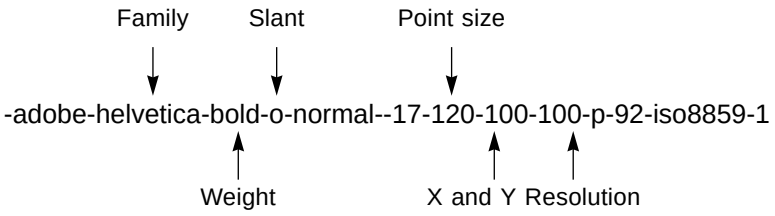


Figure 5-4 A Typical Font Name

The example in Figure 5-4 specifies a display resolution of 100 dpi, both horizontal and vertical, which is appropriate for most workstation displays.

5. Pull down the ResX Menu and select a resolution of 100 or 75 dpi.

Select the resolution appropriate to your display. If in doubt, use 100 dpi.

5.3.4 Applying the Font

The selections you have made are sufficient to specify a font for the interface. Now you can apply it to the “Large” radio button.

1. Click on “Apply” in the Font Selector.

This applies your selection to the “Font” resource in the resource panel.

2. Click on “Apply” in the resource panel.

The “Double Scooper” radio button is now labeled in a large, bold, oblique font.

3. Select the “Small” radio button.

4. Pull down the PtSize (point size) Menu and select “100”.

You can press the “Sample text” button to re-display the example text whenever the filtering has been changed. SPARCworks/Visual doesn’t do this automatically as some fonts can take a very long time to load.

5. Click on “Apply” in the Font Selector and in the resource panel.

The “Small” radio button is now labeled in a bold, oblique font, smaller than the “Large” label. Before proceeding, reset the Font Selector panel so that it shows all the fonts:

6. Click on the “All fonts” button at the bottom of the Font Selector.

This resets all elements of the font filter to “*” and so all available fonts are displayed again.

5.3.5 Scalable Fonts

The font you used on the radio buttons is a non-scalable font. This means that it is only available in certain fixed point sizes. X also supports scalable fonts, which can be any size you like. Try selecting some scalable fonts:

1. Clear the “Non-scalable” toggle in the Font Selector.

This eliminates all non-scalable fonts from the list and so the list is now empty.

2. Set the “Scalable” toggle.

This adds the scalable fonts to the list. You can specify a size for any of these fonts in two ways: by adjusting either the pixel size or the point size. The pixel size is the first numeric field in the font descriptor and the point size is the second numeric field. Both of these fields are initially set to zero for all scalable fonts, indicating a default size.

3. **Select the "Vanilla" ToggleButton.**

4. **Click on one of the font names in the list.**

Your selected font appears in the "Font name", "Selection" and "Sample text" fields.

5. **Click in the "Font name" field (not the "Selection" field).**

6. **Edit the second numeric field (point size) to 240 (24 points).**

7. **Press <*Return*> or click on the "Sample text" button to display a sample of this text size.**

When using scalable fonts, specify a non-zero value for either the pixel size or the point size, but not both. X adjusts the remaining zero value to fit the explicitly specified value. If you specify non-zero values for both fields, however, SPARCworks/Visual displays an error message if they do not match.

Typical point size values, specified in tenths of a point, are larger than typical pixel size values. A point size of 240 roughly corresponds to a pixel size between 30 and 35. The exact proportion depends on the resolution of your screen and the specific font.

X has two ways of scaling fonts: outline scaling and bit scaling. If the sample text is very jagged, the font is bit-scaled. To list only outline-scaled fonts, pull down the Fndry Menu and select "bitstream".

After experimenting, set the Font Selector to the non-scalable fonts again:

8. **Click on the "Non-scalable" and "Scalable" toggles.**

5.3.6 Simple Font Objects

For the two radio buttons, you set the fonts individually. If you use this method to set the same font for multiple widgets, any later changes must also be made for each widget individually. Also, the code generated by SPARCworks/Visual for the application makes a separate call to your system to load the same font for each label, which is inefficient.

Therefore, if multiple widgets use the same font, you can simplify both maintenance and the code by creating a simple font object. A *font object* is an alias for a list of fonts. A *simple font object* is an alias for a one-element list.

1. **Use the pulldown menus to select the 12-point bold oblique helvetica font.**

If more than one font appears in the list of X fonts, highlight one.

2. **In the "Font object" field, type:** `option_labels`

Remember that it is better to name a font object for the function it serves rather than for the size or style it represents, since these specifications can change.

3. **Click on "Bind".**

This creates a font object called "option_labels". It only has one font in its list: the 12-point helvetica font. This has an associated fontlist tag "<Default>". The use of fontlist tags is discussed in *The Compound String Editor* chapter.

Notice that the "Selection" field automatically updates to show the "<option_labels>" name. The angle brackets (<>) indicate that it is a font object rather than a font name. This font is applied to the resource panel when you click on "Apply".

4. **Click on "Apply" in the Font Selector.**

This applies the font object to the ToggleButton resource panel.

5. **Click on "Apply" in the resource panel.**

This applies the font object to the "Vanilla" ToggleButton. Now you can apply the same font object to the other labels in the interface:

6. **Select the "Chocolate" ToggleButton in the hierarchy.**

7. **Click on "Apply" in the Font Selector.**

8. **Click on “Apply” in the resource panel.**
9. **Select the “Strawberry” ToggleButton and repeat steps 7 and 8.**
10. **Select the “Topping 1” Label and repeat steps 7 and 8.**
11. **Select the “Topping 2” Label and repeat steps 7 and 8.**
12. **Select the “Topping 3” Label and repeat steps 7 and 8.**

The interface now looks as shown in Figure 5-5.

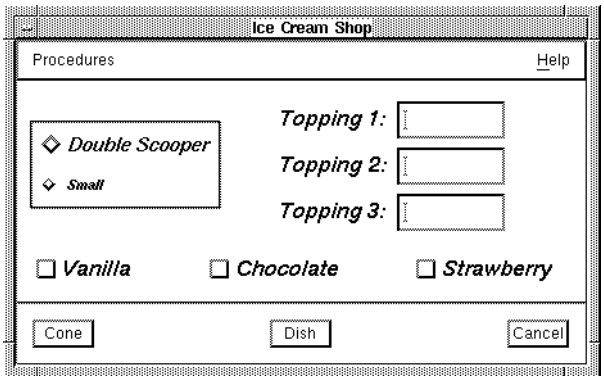


Figure 5-5 Interface with Fonts Applied

5.3.7 Changing the Font Object

The text on the ToggleButtons and Labels is now shown in the font to which the font object is bound. To change this font style, just bind the font object to another font.

1. **Pull down the Slant Menu and select “r” for “regular”.**
2. **Pull down the Family Menu and select “Times”.**

The font list now displays the 12-point bold Times fonts. If the list is empty, select a different font family.

3. **Click on “Bind”.**

The font object changes to correspond to the Times font and all the Labels and ToggleButtons change immediately in the dynamic display.

Font objects are saved with the design file. This means you can use the same names for font objects in different design files, even though the font might be different. For example, the *option_labels* font object might be Helvetica in one design and Times in another.

5.4 Selecting Pixmaps

You can use pixmaps instead of text strings on labels and buttons. SPARCworks/Visual provides two editors for creating and applying pixmaps. First you will use the Pixmap Selector to learn the basics of applying pixmaps to widgets. Then you will use the Pixmap Editor to create some custom pixmaps.

SPARCworks/Visual lets you use bitmaps created using the standard X bitmap editor. It also lets you use pixmaps in the public domain *Xpm* format and provides an editor for you to build pixmaps in this format. The *Xpm* library is included with the SPARCworks/Visual release.

Note that you can use pixmaps created with any other utility provided they are in *Xpm* format. You can also edit X bitmaps using the SPARCworks/Visual pixmap editor. When you do this, SPARCworks/Visual converts the bitmap to a two-color pixmap. You can keep it in two colors or add more colors to it.

As a first step, replace the label of a ToggleButton with a bitmap.

1. **Click twice on the “Cone” PushButton in the hierarchy.**

2. **On the “Display” page of the resource panel, click on “Pixmap”.**

SPARCworks/Visual displays the Pixmap Selector. This is shown with example entries in Figure 5-6:

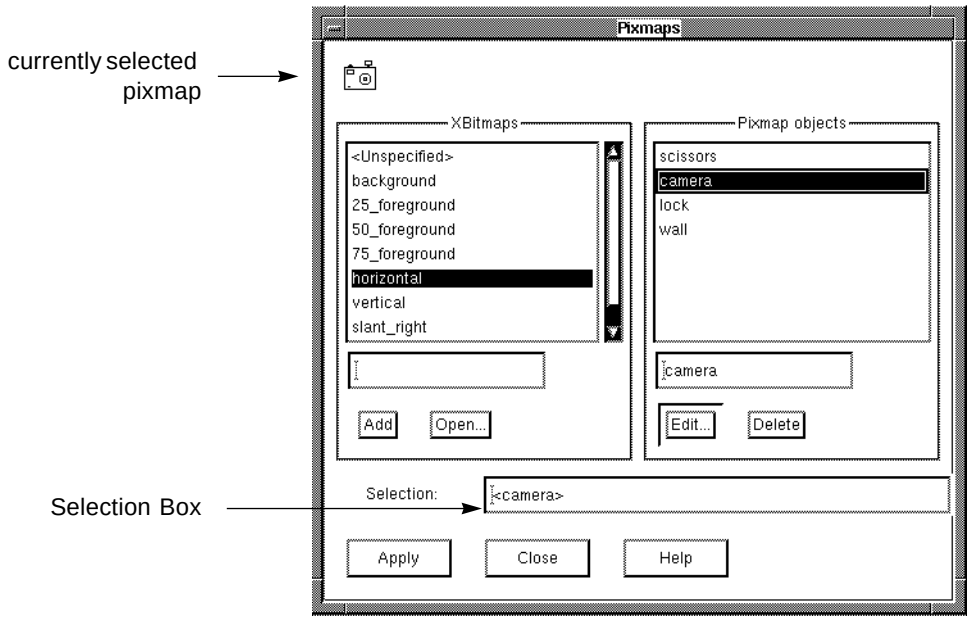


Figure 5-6 The Pixmap Selector with Example Entries

3. Select any bitmap from the list of X bitmaps.

The selected bitmap is displayed at the top of the Pixmap Selector.

4. Click on “Apply” in the Pixmap Selector.

This applies your selection to the “Pixmap” resource in the resource panel.

5. Click on “Apply” in the resource panel.

Now the ToggleButton has both a text label and a bitmap label, although only the text label appears in the dynamic display. To display the bitmap label, you must change the resource that controls which type of label is displayed.

6. On the “Settings” page of the resource panel, change the “Type” setting to “Pixmap”.

7. Click on “Apply” in the resource panel.

The ToggleButton now displays the bitmap instead of the text label. Since the bitmap does not convey any useful information in this case, change the “Type” resource back to “String”.

If you have additional X bitmap files on your system, you can also use these. Clicking on “Open...” displays a file selector which lets you locate bitmap files and add them to the list of X bitmaps.

You can also enter names of bitmaps in the text box under the list of X bitmaps, then click on “Add” to add the name to the list. If the bitmap doesn’t exist yet, you can still add its name to the list and apply it to resources. Later, in development or at run time, you can supply the bitmap.

5.5 Editing Pixmaps

SPARCworks/Visual provides an editor for creating pixmaps. The Pixmap Editor also lets you:

- Bind pixmaps to pixmap objects
- Load pixmaps files in XPM3 format
- Load X bitmap files and convert them to pixmaps for editing
- Write pixmaps to files in XPM formats

All pixmaps used in SPARCworks/Visual must be bound to pixmap objects. First, you will create a pixmap for the “Cone” PushButton, as shown in Figure 5-7:

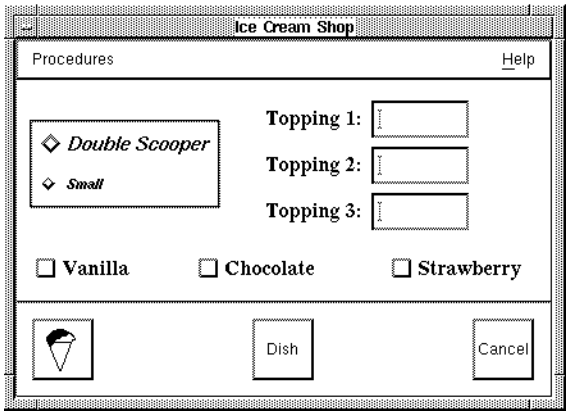


Figure 5-7 The Tutorial Interface with Pixmap Button

If you are continuing from the *Selecting Pixmaps* section on page 123, you already have the Pixmap Selector displayed and so you can skip the next two steps.

1. **Click twice on the “Cone” PushButton in the hierarchy.**
2. **On the “Display” page of the resource panel, click on “Pixmap”.**
3. **In the text box in the “Pixmap objects” portion of the Pixmap Selector, type:** `cone`
4. **Click on “Edit...”.**

SPARCworks/Visual displays the Pixmap Editor, shown in Figure 5-8:

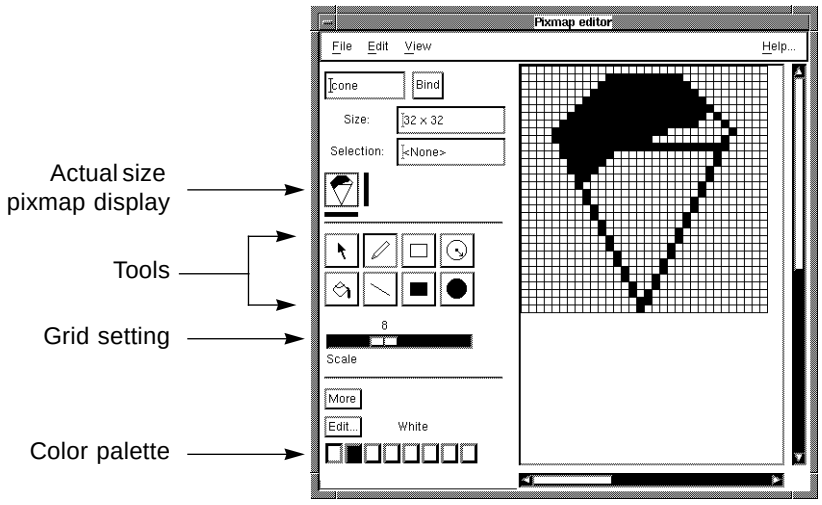


Figure 5-8 The Pixmap Editor

First you can set the size of the pixmap you'll be creating.

5. **Click in the "Size" text box and drag to highlight all the text.**
6. **Type**: `40 x 40`
7. **Press <*Return*>.**

The actual-size pixmap display and the grid in the drawing area should resize. To draw:

8. **Click on the black color box in the lower left corner.**
9. **Click on the filled rectangle tool.**
10. **Click in the drawing area and drag to create a black filled rectangle.**

The actual-size pixmap display updates to show what you've just drawn.

Experiment with the drawing tools until you feel comfortable using them. The arrow is a selector, not a drawing tool. Don't create the pixmap for the "Cone" yet; it is helpful to learn the options on the Edit Menu and add some colors to the color palette before you undertake a drawing task. The Edit Menu in the menu bar should not be confused with the "Edit" pushbutton in the color palette. You will be using both.

5.5.1 The Editing Options

The options in the Edit Menu operate on the selected portion of the pixmap. To select, use the arrow in the tool palette.

1. Click on the arrow.

2. Click in the drawing area and drag.

The selected portion of the pixmap includes pixels in the rectangular border marked with crosses and all the pixels within the border. Note that the dimensions of the selected area are echoed in the text box next to “Selection”. These values may be modified by the user. The format used is the dimension of the selected area followed by the coordinates of the top left corner.

3. Pull down the Edit Menu.

Experiment with the options in this menu until you feel comfortable using them. Experiment with changing the pixmap size or the selected portion by entering numbers in the text boxes. Try changing the scale of the drawing area using the slider. The grid defining individual pixels disappears at a scale of 5 or less. When you are familiar with these features, proceed to the next step.

5.5.2 The Color Palette

Although the Pixmap Editor lets you create a color palette containing up to 256 colors, it starts with only two colors. Now you will add some colors.

1. Click on the third box in the color palette.

2. Click on “Edit...” in the color palette.

SPARCworks/Visual displays the Color Selector. You can also double click on a box to bring up the color selector.

3. Select a color.

Use any method to specify a color. You can use X colors, create your own color using the sliders, or use a color object.

4. Click on “Apply”.

The selected color is displayed in the color palette. By repeating these steps, you can build a customized color palette. If you want more colors in the palette, click on “More”.

Now that you have more colors, experiment more with the drawing tools. When you are ready, use the tools and colors to create a pixmap showing an ice cream cone.

5. **Draw an ice cream cone.**

5.5.3 Changing Colors

Even after you have used a color in the pixmap, you can change the color and see the results immediately.

1. **In the color palette, click on the color you want to change.**
2. **In the Color Selector, select a color.**
3. **Click on “Apply”.**

The new color replaces the old color in the color palette and in the pixmap.

5.5.4 Pixmap Objects

To display your finished pixmap, you must bind it to a *pixmap object* first. Because you entered the name of the pixmap object originally in the Pixmap Selector, all you have to do now is bind the pixmap to the object.

1. **Click on “Bind” in the Pixmap Editor or press <Return> in the “Bind” field.**

In the Pixmap Selector, the pixmap object name appears in the list of pixmap objects and in the selection box.

2. **Click on “Apply” in the Pixmap Selector.**
3. **Click on “Apply” in the resource panel.**
4. **On the “Settings” page, change the “Type” setting to “Pixmap”.**
5. **Click on “Apply” in the resource panel.**

Pixmap objects work very much like color objects. You can use the same pixmap in more than one place. Changes you make to the pixmap are reflected in the dynamic display as soon as you bind again.

Pixmap objects are saved with the design file. This means you can use the same names for pixmap objects in different design files, even though the pixmap might be different. For example, the pixmap object, *cancel*, might be a cancel stamp in one design and the international “No” symbol in another.

If you want, try creating and adding pixmap objects for the “Dish” and “Cancel” buttons. Hint: the lettering for the “Cancel” stamp shown in Figure 5-9 was done with the line tool, not the pencil tool.

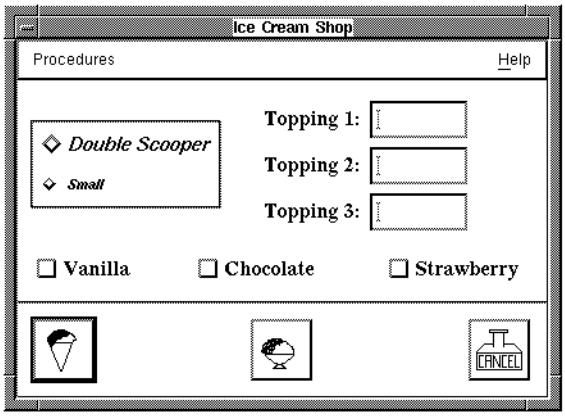


Figure 5-9 The Tutorial Interface with Three Pixmap Buttons

5.5.5 *Reading Pixmap and Bitmap Files*

If you have X pixmap files on your system, you can also use these in your interface. Selecting “Read file...” in the File Menu displays a file selector which lets you load a pixmap file into the Pixmap Editor. SPARCworks/Visual reads pixmaps in XPM3 format. It also reads X bitmap files and converts the bitmaps to pixmaps for editing in the Pixmap Editor.

5.5.6 Writing Pixmap Files

Selecting “Write XPM File...” lets you write the pixmaps you create to files. SPARCworks/Visual writes XPM1 and XPM3 format. You should normally save pixmaps in XPM3 format. XPM1 is provided for compatibility with older versions of third-party pixmap handling utilities.

5.5.7 Saving Your Work

Every time you bind or write to an XPM file, you effectively save the current state of the pixmap you are creating. It’s a good idea to do this frequently as you work.

5

Additional Windows and Links 6

6.1 Introduction

You have now finished setting up the main window of your interface. Most interfaces, however, have multiple windows. This chapter shows how to add a second window to your interface. The second window is a simple help screen, as shown in Figure 6-1:

Figure 6-1 Help Screen

This help screen will appear when the user invokes the “About This Layout” command in your interface’s Help Menu and will disappear when the user clicks on the “OK” button. This behavior is similar to that of SPARCworks/Visual’s copyright screen. Before you start, you may want to pull down SPARCworks/Visual’s Help Menu and select “About SPARCworks/Visual”. Note that the copyright screen appears when you click on “About SPARCworks/Visual” and disappears when you click on its “OK” button.

To achieve these results, you will:

1. Create an additional window for your interface.
2. Design a simple help screen within the second window.
3. Create a “Show” link to display the help screen when the “Help” command is given from the Help Menu.
4. Create a “Hide” link to remove the help screen when the user clicks on the “OK” button.

6.2 Creating a Second Window

You can create a new window at any time, regardless of which widget is selected in the design hierarchy.

To add a dialog to your interface:

1. Click on the Shell icon in the widget palette.

SPARCworks/Visual clears the construction area and displays the hierarchy for the new window. So far, this consists only of the Shell. Note that the dynamic display for the first window is still visible. As you build the secondary window, you can see both dynamic displays at the same time.

2. Click on the DialogTemplate icon.

3. Click on the Label icon.

4. Click on the PushButton icon.

The hierarchy for the subwindow and its default dynamic display are shown in Figure 6-2. Because this screen is so simple, you can use a Label instead of a container widget with children for the work area. The DialogTemplate centers the PushButton in the button box with the work area above it. There is no menu bar.

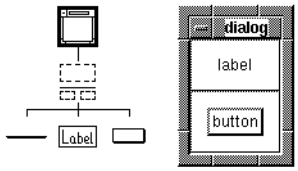


Figure 6-2 Hierarchy and Default Dynamic Display for Second Window

Set the text on the Label and PushButton:

5. **Click twice on the Label in the design hierarchy to bring up the Label resource panel.**

6. **On the "Display" page, double-click in the "Label" box and type:**

```
This is a "Help" screen.
Are you finding
it helpful?
```

Use <*Return*> to put newlines into a multi-line label. Do not put a newline at the end of the last line.

7. **Click on "Apply".**

8. **Click on the PushButton in the design hierarchy.**

9. **Double-click in the "Label" box and type:** `OK`

10. **Click on "Apply".**

11. **Click on "Close".**

6.3 Shell Types

Although every window starts with a Shell widget, there are different types of windows and different types of Shell widgets. SPARCworks/Visual provides the following Shell widgets:

- Application Shell – The main window of the application, which is the first one displayed when the application runs

- Top Level Shell – A window other than the Application Shell which remains visible when the Application Shell is iconified and can be iconified independently
- Dialog Shell – A window which cannot be iconified independently of the Application Shell

The behavior described above applies to *mwm*. With *twm*, although Shell behavior is the same, it looks different because *twm* can turn Dialog Shells into pseudo-icons to reduce their size. The pseudo-icons are just a visual convenience for cleaning up your display. Internally, they are distinguished from true icons and they look different on your screen.

All windows in the design close when the Application Shell is closed.

6.3.1 Examples of Shell Types in the SPARCworks/Visual Interface

To see some possible uses of different Shell types, look at the SPARCworks/Visual interface itself. The main screen with the widget palette and construction area is the Application Shell. Dialogs, resource panels and the Layout Editor screen are all Dialog Shells. You cannot iconify these windows separately using the window manager.

To remove them from the display, you must close them, either using the window manager or by clicking on a “Close” button, which closes the window internally via an “Activate” callback. All open Dialog Shells disappear when the main window is iconified and reappear when it is restored.

The “Palette Icons Help” panel is an example of a Top Level Shell. While it does not come up automatically when SPARCworks/Visual starts, it can still be iconified independently once you have displayed it. Its popup subwindows, like all Dialog Shells, are children of the main Application Shell and do not close or iconify with the “Palette Icons Help” panel.

6.3.2 Shell Type in the Dynamic Display

The Shell type is not reflected in the dynamic display. All windows created for dynamic display are really Dialog Shells and cannot be iconified independently. You can configure SPARCworks/Visual to use Top Level shells rather than Dialog Shells. See the Application Defaults section in the Appendices. The generated code creates the type of Shell you specify for each window at run time.

6.3.3 Application Shell Requirement

You should designate at least one Shell in each design as the Application Shell. If you have no Application Shell in your design, the application will not display any windows. SPARCworks/Visual shows you a warning message at code generation time if you do not have an Application Shell.

You can have more than one Application Shell in your design. In this case, the *main()* program generated by SPARCworks/Visual creates all the Application Shells but displays only one of them. SPARCworks/Visual cannot tell which one you want displayed first. If you have more than one Application Shell in your design, you may have to write your own *main()* program or edit the generated one to start with the correct Application Shell. To get similar results without ambiguity, use only one Application Shell for your first window, use Top Level Shells for other primary windows and use callbacks or links to display all windows but the first. Using more than one Application Shell is not recommended.

In the *Using the Resource Panels* chapter, you designated the first Shell in this design as an Application Shell. The new Shell should be a Dialog Shell.

To view its resource panel:

1. **Click twice on the Shell.**

If the “Dialog shell” toggle at the top of the Resource Panel is not already set:

2. **Click on the “Dialog shell” toggle.**

You can also change the title on this Resource Panel.

3. **Select the “Display” page.**
4. **Double-click in the “Title” box and type**: `Help`
5. **Click on “Apply”.**
6. **Click on “Close”.**

6.3.4 Navigating Between Windows

When you add a Shell to your design, a corresponding icon appears in the window holding area at the top right of the main SPARCworks/Visual window, as shown in Figure 6-3. To move from one window's hierarchy to another, click on the Shell icon associated with that window in the window holding area. Because most Dialog Shell icons look alike, and because icons are not necessarily shown in this area in the order in which you created them, it helps to assign explicit variable names to all Shell icons and turn on the "Show dialog names" option in the View Menu so that you can tell them apart.

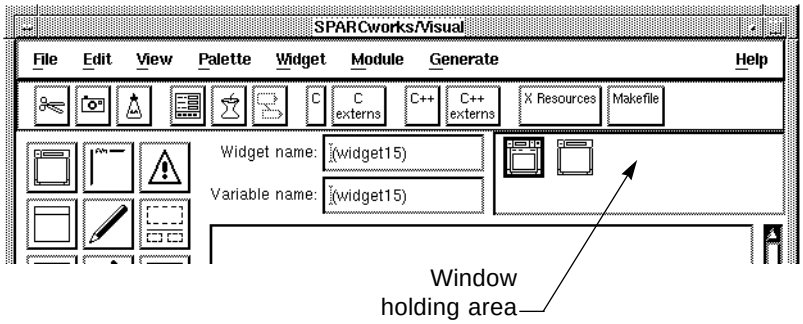


Figure 6-3 Upper Part of SPARCworks/Visual Screen

Assign a name to the second Shell in your design.

1. **If the Shell is not already selected in the hierarchy, select it.**
2. **Double-click in the "Variable name" field.**
3. **Type**: `help_window`

To register the new name:

4. **Type <Return> or select any other widget in the hierarchy.**

To display the Shell names in the window holding area:

5. **Pull down the View Menu and select "Show dialog names".**

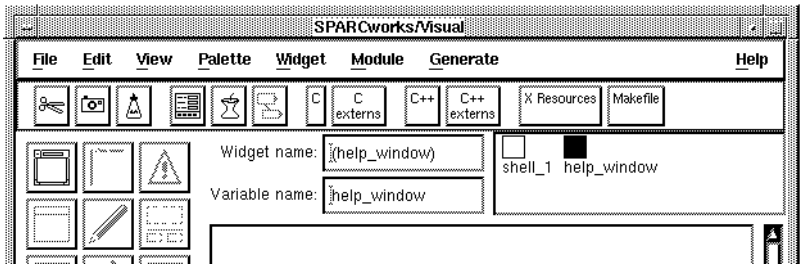


Figure 6-4 Window Holding Area with Dialog Names

6.4 Links

SPARCworks/Visual has predefined callback procedures called *links*. There are four links available:

- *Show* – makes a widget and its children appear on the screen
- *Enable* – enables a button or command
- *Disable* – disables (grays out) a button or command
- *Hide* – makes a widget disappear. The widget is not destroyed, just hidden

6.4.1 Distinction between Links and Callbacks

Only PushButtons, ArrowButtons and CascadeButtons can have links. All links are triggered by “Activate”. A link can show, hide, enable, or disable any widget in the design. One button can have multiple links.

Unlike callbacks, links work in the dynamic display and can therefore be used for prototyping window behavior. When you generate code, you can either include links, which work exactly as they do in the dynamic display, or substitute more complex callbacks for the simple links.

You are now going to set a common configuration of links to display the help screen you have just built and make it disappear again at the proper time. To do this you will:

- Set a “Show” link on the “About This Layout” button in the Help Menu
- Set a “Hide” link on the “OK” PushButton in the help screen

6.4.2 Widget Naming Requirements

The button widget on which links are set does not have to be explicitly named. However, the *target widget* of a link - that is, the widget to be shown, hidden, enabled, or disabled - must have an explicit variable name. If the target widget is a Shell, its immediate child must also be explicitly named. If any of these names change, the link is no longer effective. Note also that the widgets on either side of a link must not be designated as *static* or *local* variables.

The "OK" PushButton is currently visible in the construction area and so begin by setting the "Hide" link on this PushButton.

1. **Select the PushButton.**
2. **Double-click in the "Variable name" field.**
3. **Type**: `ok_button` **and press <Return> to register the new name.**
4. **Pull down the Widget Menu and select "Edit links".**

This displays the panel shown in Figure 6-5.

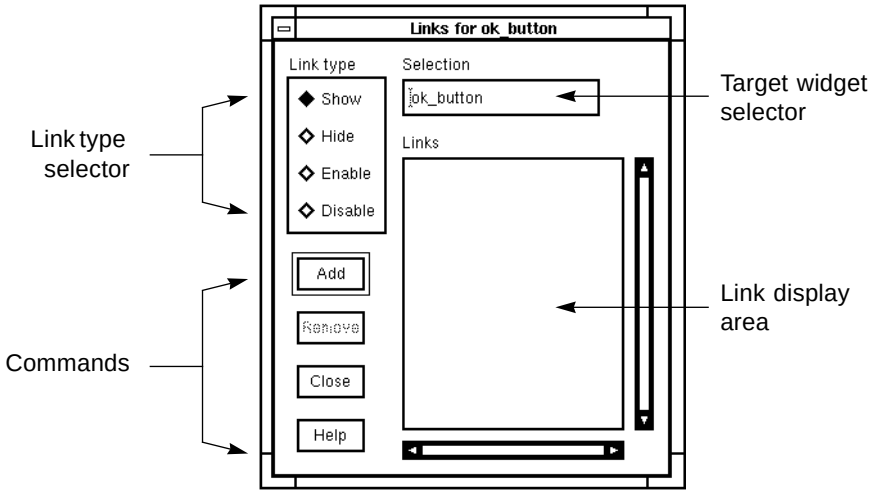


Figure 6-5 Default Links Panel

The target of the "Hide" link should be the widget *help_window* so that when the "OK" button is activated, the entire help screen disappears.

To select the target widget:

5. **Select the Shell in the design hierarchy.**

The name of the Shell, *help_window*, appears in the “Selection” field of the Links panel. However, the “Add” command is still disabled. This is because you have not yet named the DialogTemplate which is the immediate child of the Shell. As discussed above, the child of a Shell must be named explicitly before you can set a link to the Shell.

You can leave the Links panel open while you name the DialogTemplate:

6. **Select the DialogTemplate.**
7. **Double-click in the “Variable name” field and type:** `dialog_2`
8. **Select the Shell.**

The Shell is now a valid target widget and so “Add” is enabled.

Now select the type of link:

9. **Click on the “Hide” toggle.**
10. **Click on “Add”.**

The new link appears in the link display area, as shown in Figure 6-6.

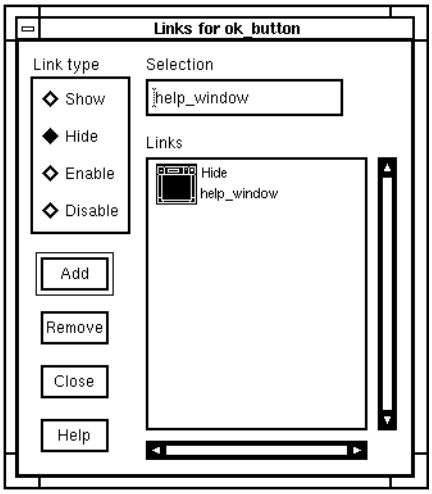


Figure 6-6 Links Panel with New “Hide” Link

11. Click on “Close”.

To demonstrate the new link:

12. Click on the “OK” button in the dynamic display.

The help screen vanishes. You can restore it by resetting the Shell.

You can also set up a “Show” link to display the help screen when a button is pressed in the main window. To do this:

13. Click on the *shell_1* icon in the window holding area.

The hierarchy for the main window is displayed in the construction area.

Set the new link on the PushButton in the Help Menu:

14. Select the *help_button* widget, the PushButton child of the second Menu.

The Links panel, unlike resource panels and the Layout Editor, does not automatically start adding links to the currently selected PushButton. To edit links for the currently selected button, you must:

15. Pull down the Widget Menu and select “Edit links”.

The Links panel now displays the name and the links (none, so far) of the current PushButton. Select the target widget, which is the Shell for the help screen:

16. **Click on the *help_window* icon in the window holding area.**

In the Links panel:

17. **Click on the "Show" toggle.**
18. **Click on "Add".**

The new link appears in the link display area.

To demonstrate the behavior of these two links:

19. **Click on the *shell_1* icon in the window holding area.**
20. **Pull down the Help Menu in the dynamic display and select "About This Layout".**

The Show link on this pushbutton makes the help screen appear.

21. **Click on the "OK" button in the dynamic display of the help screen.**

The Hide link on this pushbutton makes the help screen disappear. You can repeat the previous two steps as many times as you want.

22. **Save your design.**

6.4.3 Removing Links

To remove a link:

- **Select the link's icon in the link display area and click on "Remove".**

6

Search 7

7.1 Introduction

This chapter describes the search and annotation features of SPARCworks/Visual. These features enable you to navigate your way around a large design by highlighting widgets which conform to a particular set of criteria.

7.2 Search

The search facility allows you to search for strings in preludes, callbacks, methods, translations, widget and/or variable names and string resources. The Search dialog is invoked from the Edit menu and is shown in Figure 7-1.

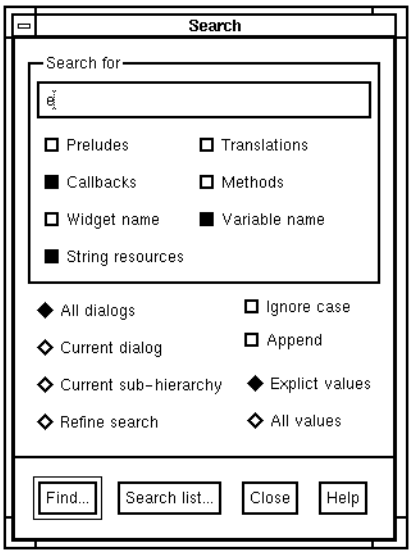


Figure 7-1 The Search Dialog

To use the search mechanism, follow these instructions:

23. **Display the Search dialog**
24. **Type the string you are looking for in the text area labelled “Search for”.**
25. **Select where you wish to find the string by selecting the relevant toggle buttons. You can select any number of these at once.**
26. **Select where you wish SPARCworks/Visual to look by selecting one of the following toggles:**
    - *All dialogs* – Look through all dialogs in the current design
    - *Current dialog* – Only search through the current dialog
    - *Current sub-hierarchy* – Only search through the hierarchy below and including the selected widget
    - *Refine search* – Only search through those widgets which are already in the search list dialog from a previous search

27. **Select whether you wish SPARCworks/Visual to ignore the case of letters in the string when looking for a match.**

28. **Select whether you wish to "Append" to an existing list of widgets which were found as the result of a previous search.**

29. **If you are searching string resources or widget or variable names, then select whether you want to search only values which you have explicitly set or all values including defaults.**

30. **Press "Find".**

Any widgets which match the search criteria are displayed in a separate dialog, the Search list dialog. Pressing "Search list" displays the list of widgets which have already been found - it does not repeat the search (this is useful if you have closed the Search list dialog and wish to view the same list again).

7.2.1 The Search List Dialog

The Search list dialog shows a list of widgets which match one or more of the search criteria. After selecting a widget from this list, the following options are available:

- *Go to* – The corresponding widget in the design hierarchy is selected. If the widget is in a part of the hierarchy which was folded, it is unfolded. Similarly, if the widget is not in the current dialog, the relevant dialog is selected first. If the string is found anywhere other than in the widget or variable name, the resource panel containing the string is opened (Callback Methods dialog, for example). Note that *Double-clicking* on an item is the same as pressing *Go to*
- *Next* – The next widget in the list is selected and the Go to action is invoked on the selection.
- *Clear* – Clears the list so that you can perform another search

Deleting a widget will remove it from the list, as will temporary deletions such as reset or cut and paste.

Applying the search criteria shown in Figure 7-1 to the tutorial example results in the Search List Dialog shown in Figure 7-2.

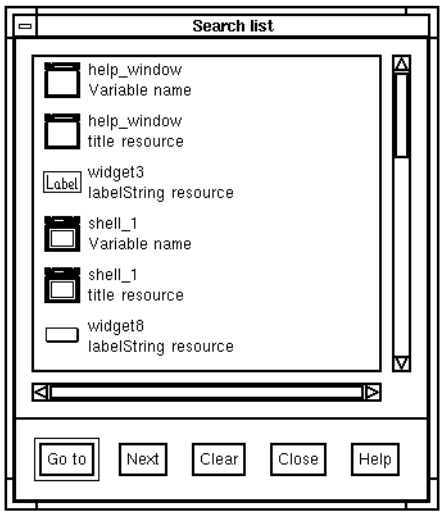


Figure 7-2 The Search List Dialog

7.3 Annotations

SPARCworks/Visual also provides a method of annotating the design hierarchy to indicate which widgets have been given certain attributes or match the search criteria. The View menu contains a pullright tear-off menu labelled Annotations, as shown in Figure 7-3.

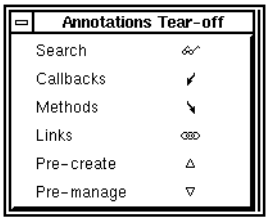


Figure 7-3 The Annotations Menu

There are six categories in this menu, each of which is a toggle button. To see which widgets in the design hierarchy have been given one of these attributes, select the toggle. The corresponding symbol is instantly placed next to each widget in the design hierarchy which matches the criteria of the associated symbol:

- For *Callbacks, Methods, Pre-create preludes and Pre-manage preludes* the criterion is that the widget has been given one of these. Selecting Methods will also find those widgets which have had a method declaration added as the result of a method having been defined for one of its children
- For *Links*, the criterion is that the widget is the source of a link
- For *Search*, the criterion is that the widget was found in a previous search, as described above

A section of the tutorial hierarchy with all the annotations set is shown in Figure 7-4.

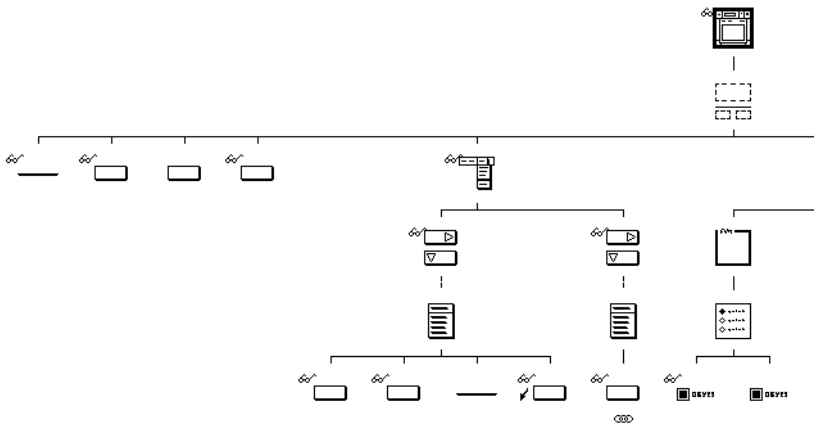


Figure 7-4 An Annotated Hierarchy

7.3.1 Configuring the Annotation Symbols

The annotation symbols are arranged around the widget icon in the hierarchy in such a way that all six symbols can be seen clearly. Where they appear in relation to the icon, how much space they use and the name of the pixmap are all specified in the SPARCworks/Visual resource file. You can change these. The relevant lines are as follows - the example here is the search symbol:

```
visu*annotate_search.annotatePosition:NorthWest
visu*annotate_search.annotateWidth:10
visu*annotate_search.annotateHeight:3
visu*annotateSearchPixmap:an_search.xpm
```

The first line above specifies the geographical location NorthWest. This is in relation to the widget icon and can be any of the eight primary or secondary compass points.

Generating Code 8

8.1 Introduction

Up to this point, you have used the interactive features of SPARCworks/Visual to build a working prototype of a user interface. Now you can use the code generation features to produce the files necessary to convert that design into a free-standing program. Code generation works in C, C++, or UIL.

In this chapter, you will:

- Generate a primary module for your design, including all code needed to prototype your interface
- Generate a stubs file for convenience in writing callback functions
- Compile, link and run your prototype
- Generate an editable X resource file containing resource settings which are not hard-wired into the code
- Write your *quit()* callback

This chapter also includes an analysis of the code which is generated into the various files and a discussion of strategies for arranging your files.

Prerequisites

You need some knowledge of C, C++, or UIL to understand the generated code files and supply code for callback functions. You also need some knowledge of the X Window System.

8.2 The Generate Menu

The Generate Menu is used to generate source code, X resource files and Makefiles from your design. The Generate Menu has five selections: “C”, “C++”, “UIL”, “X Resources” and “Makefile”. The first three options let you select the language you want to work in. These options apply only to code files. The fourth option, “X Resources”, generates the X resource file for your interface. X resource files are the same in all three languages. “Makefile” generates makefiles for use in building your applications.

Note that in Windows mode the Generate Menu has an additional selection to generate Windows resource files. This is discussed in the *Building the Application* section of the *Cross Platform Tutorial* chapter on page 296.

C is used for the examples in this chapter. The procedure for generating C++ is exactly the same and you may use C++ for the tutorial if you prefer. The procedure for UIL is very similar to the procedure for C with the exception of one additional step which is discussed later in this chapter.

To display the code generation dialog:

1. Pull down the Generate Menu.

2. Select “C”.

A pullright menu appears with the following options: “C”, “Stubs”, “Externs”, and “C pixmaps”. This menu has a dashed line at the top. The dashed line indicates that you can tear off the pullright menu into a separate window for convenience in reusing it. Do this, since you will be returning to this menu several times:

3. Click on the dashed line at the top of the pullright menu.

The pullright menu is transferred to a separate window, as shown in Figure 8-1. You can move the menu to any convenient place on the screen.

Figure 8-1 Tear-off Menu

The “C” option (<*Alt-C*>) generates the *primary module* for the interface - the file with all the C code needed to set up the design hierarchy you have built. The primary module may or may not include a *main()* procedure. The other three options on the menu generate optional auxiliary files. This is described in more detail later in this chapter.

Now, generate a primary module for your interface:

4. Select “C” from the tear-off menu.

This displays the Generate Dialog, as shown in Figure 8-2.

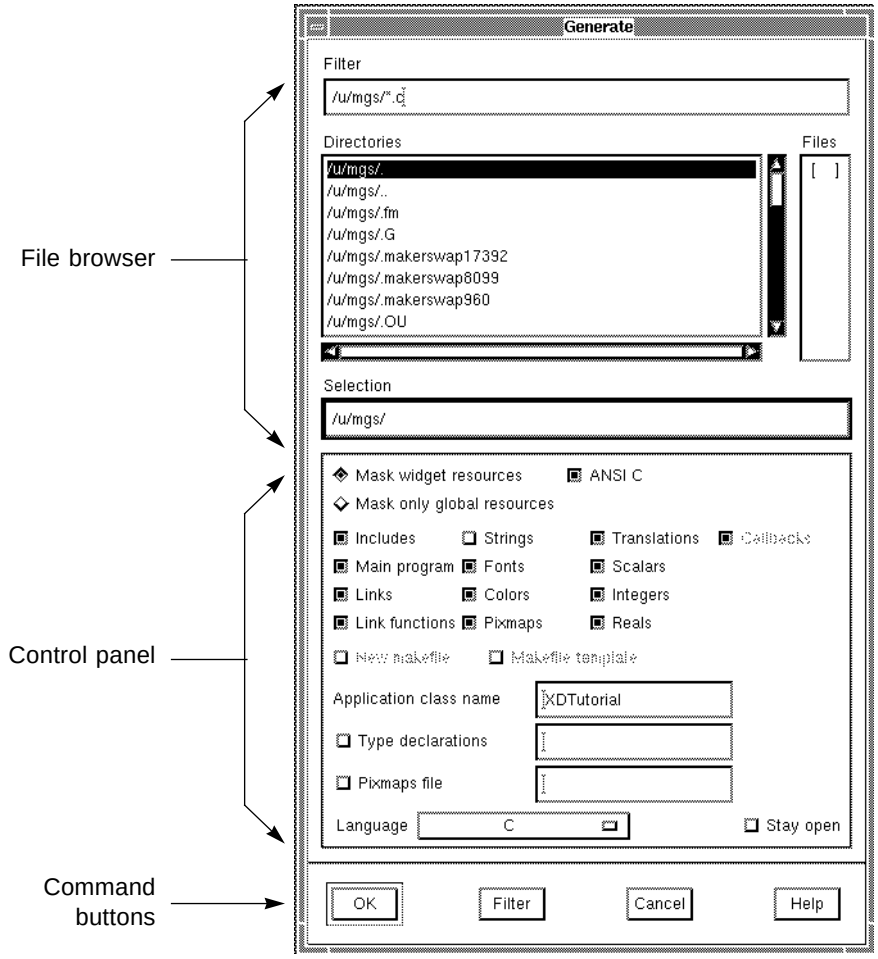


Figure 8-2 The Generate Dialog

8.2.1 The File Browser

The Generate Dialog has two sections: a file browser and a control panel. The file browser works the same as in “Save as...” and “Open”. Note that the “Filter” button for the file browser is located with the other buttons at the bottom of the whole panel.

♦ **Use the file browser to find your directory and assign your C file the name:** `icecream.c` **but do not click on "OK.**

By convention, all C files, including primary modules and stubs files, have the suffix `.c`.

8.2.2 The Control Panel

The control panel at the center of the Generate Dialog lets you control which resources are generated into your source code. If a resource is generated into the source code it is then hard-coded and cannot be modified through the resource file. Typically, any resources which are not generated into the source code are generated into the X resource file, where they can be edited by the end user. You can also control whether you generate links, *#include* statements, or a *main()* procedure. The Generate dialog is described in greater detail later in this chapter.

First, generate a code file using the default settings. Then inspect the resulting file and read on for a detailed discussion of the options on the control panel.

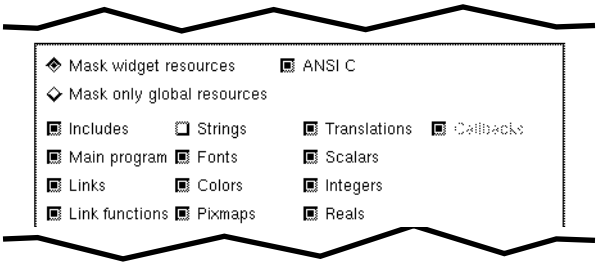


Figure 8-3 Default Toggle Settings in Generate Dialog

1. Set the toggles on the control panel as shown in Figure 8-3.

Note that all types of resources are generated except strings. This is to prevent the string resources from being hard-coded and allowing them to be modified through the X resource file described later in this chapter.

2. Unset the "ANSI C" toggle if your compiler does not require ANSI C.

3. Double-click in the "Application class name" box and type:

`XDTutorial`

The *application class name* is used to identify resource settings when you generate an X resource file. Assigning a name other than the default "XApplication" prevents confusion of your resource values with system-wide X resources.

4. Click on "OK".

When you click on "OK", the Generate Dialog disappears and SPARCworks/Visual generates your primary module.

8.3 Generating the Stubs File

The resulting code module has all the code you need except for the callback functions. Your design has just one of these: *quit()*. The most convenient way to define *quit()* is within a stubs file generated by SPARCworks/Visual. The stubs file contains function declarations with empty braces into which your code is written.

For now, generate a stubs file with an empty *quit()* function. The dummy function lets you compile, link and run your application as a prototype. Later, you will add functionality to *quit()* to complete your application.

1. Click on "Stubs" on the C tear-off menu.

The Generate Dialog appears. Note that the resource type toggles are grayed out, because resources are not generated into stubs files. The "Includes", "Links", and "Link functions" toggles are active and should be set as shown in Figure 8-4. "Includes" should be generated since you will compile the stubs file separately from the primary module. "Links" and "Link functions" should be generated only once. Other ways to arrange your files are discussed later in this chapter.

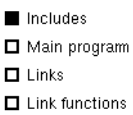


Figure 8-4 Toggle Settings for Stubs File

2. Set toggles as shown in Figure 8-4.

3. Use the file browser to assign your file the name: `stubs.c`

4. **Click on “OK”.**

The Generate Dialog disappears and SPARCworks/Visual generates your stubs file. You are now ready to compile, link and run your prototype. To do so, you will need a Makefile. However, you don’t need to write one as SPARCworks/Visual will generate one for you.

See the *Incremental Stubs File Generation* section of the *Coding Techniques* chapter on page 192 for a description of how SPARCworks/Visual will allow you to modify the generated stubs but still retain the ability to re-generate the file.

8.4 Generating a Makefile

Your makefile needs to compile both files, *icecream.c* and *stubs.c* and link the resulting object files with the required libraries.

1. **Pull down the Generate Menu.**
2. **Select “Makefile”.**
3. **Generate your Makefile to a file called “Makefile” into the same directory as your primary module and stubs files, as illustrated in Figure 8-5.**

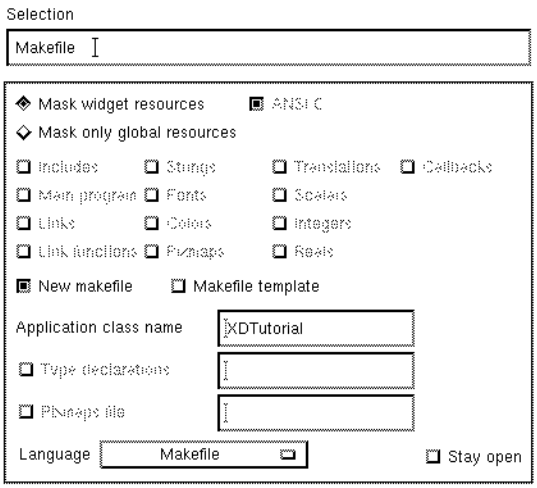


Figure 8-5 Makefile Generation

4. **To build your prototype, type:** `make`

5. **To run your prototype, type:**`icecream`

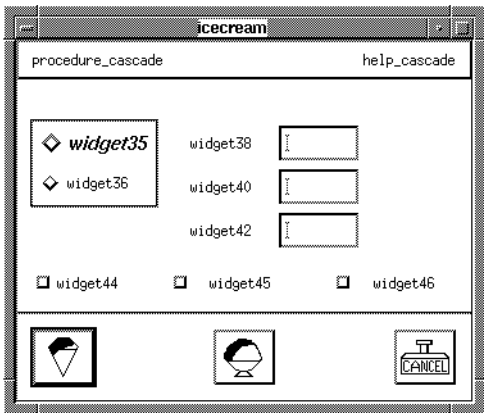


Figure 8-6 Interface Prototype Running with All Resources Except Strings

All resource settings are available except for text strings. This is because the strings were not generated into the source code file. Later you will generate them into an X resource file. When string resources are not set, Motif uses widget names as substitute labels.

As in the dynamic display, all the widgets in your prototype are functional. You can click on the buttons, pull down the menus, and so on. Your generated file also includes links. You can display the help screen by pulling down the menu at the right side of the screen and clicking on its single entry; you can make the help screen disappear by clicking on its button.

Although your prototype also calls *quit()* when you click on *exit_button*, *quit()* doesn't do anything because you have not yet supplied the code to make it functional. To terminate your prototype when you have finished examining it:

6. **Use the window manager to close the main window.**

8.5 Generating the X Resource File

As you have seen, resource settings need to be available or they are not applied when your interface runs. You can make them available in one of two places: the primary module (also known as the *source file*) or the *X resource file*. You generated all resources except strings into the source file. Therefore, non-string resources, such as the width of the text fields and the number of columns in the RowColumn, are displayed correctly. Generating resources into the source file is known as *hard-wiring* them.

Now, generate an X resource file containing all remaining resources - that is, just the strings.

1. Pull down the Generate Menu and select "X Resources..." (`<Alt-X>`).

This displays the Generate Dialog again (if it is not still on the screen). Look at the array of toggle buttons in the control panel (shown in Figure 8-7).

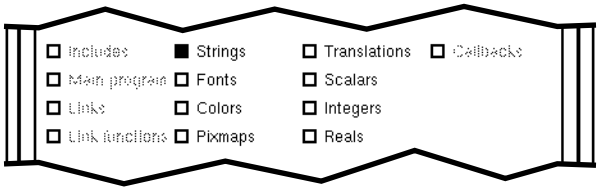


Figure 8-7 Resource Type Toggles in the Generate Dialog

The left column and the "Callbacks" toggle are disabled, since these toggles do not represent resources. Note that the settings of the remaining toggles are the inverse of their settings when you generated the primary module. SPARCworks/Visual inverts these settings for you so that the source file and the resource file complement each other. In most cases, designers want to generate into the resource file all those, and only those, resources which were not hard-wired.

2. Make sure that the resource type toggles are set as shown in Figure 8-7.

3. Make sure that the "Mask widget resources" radio button is on.

The significance of this radio button is discussed later in this chapter.

4. **Use the file browser to find your working directory and assign your X resource file the name:** `icecream.res`

5. **Click on "OK".**

This filename and directory location makes an editable copy of your X resource file in a convenient local place. However, X does not automatically recognize *icecream.res* as that application's resource file. One recommended method of telling X where to find this file is to copy the resource file to the designated application resource directory (*/usr/lib/X11/app-defaults* on POSIX systems). The filename in that directory should be the application class name, *XDTutorial*, without a suffix. This method avoids any confusion of this application-specific resource file with other files you might be using.

Because you may not have access permission to the application resource directory, a different method is described here.

6. **Set the environment variable XENVIRONMENT to the filename of the resource file:**

`setenv XENVIRONMENT icecream.res` (assuming a C shell)

There are other ways to get X to recognize your X resource file. Consult X documentation and use another method if you prefer.

7. **Run your application again:** `icecream`

This time, the correct strings should appear.

8. **Use the window manager to close the application's main window.**

8.6 Adding Callback Functionality

All that remains to complete the tutorial is adding functionality to *quit()*. To do this:

1. **Open `stubs.c` with a text editor.**

The code for quit() now looks like this:

```
void
quit (Widget w, XtPointer client_data, XtPointer xt_call_data )
{
```

```
    XmPushButtonCallbackStruct *call_data =
       (XmPushButtonCallbackStruct *) xt_call_data;
}
```

2. **Replace the text between the braces of quit() by:** `exit (0);`
3. **Exit the text editor.**
4. **Remake your executable and run the program.**

The “Exit” button is now functional. To quit your program:

5. **Pull down the Procedures menu of your interface and click on the “Exit” button.**

8.7 Analysis of the Primary Module

From top to bottom, your primary code module contains the following sections:

- Headers (optional)
- Global variable declarations
- Declarations of link functions, or the functions themselves (optional)
- Structures needed for your fonts and pixmaps and code setting up font and pixmap objects
- Widget creation code for your design hierarchy
- A *main()* procedure (optional)

This section analyzes the code in your file.

♦ **Open `icecream.c` with the text editor and inspect it as you read.**

The optional portions of the file can be included or excluded by setting toggles on the control panel. These toggles, and the advantages and disadvantages of including the optional sections, are discussed in the *Code Toggles in the Generate Dialog* section on page 166.

8.7.1 The Header Section

The primary module has the following header material, in this order:

- The module heading (if any)

- SPARCworks/Visual's heading
- The *#include* statements (optional)
- The module prelude (if any)

Your file does not yet have a module heading or module prelude. You can specify code to be inserted in these places from within SPARCworks/Visual. The procedure for doing so is discussed in the *Customizing the Generated Files: Preludes* section on page 176.

After some standard SPARCworks/Visual comments, there is a list of *#include* statements needed for the Motif code in your module. The *#include* statements are optional and are controlled by the "Includes" toggle.

8.7.2 Link Functions or Link Declarations

Next, the module contains code for the link functions. The following code fragment shows a typical link function:

```
void XDunmanage_link ( Widget w, XtPointer client_data, XtPointer
      call_data )
```

Generation of this code is optional and is controlled by the "Links" and "Link Functions" toggles.

8.7.3 Variable Declarations

In this section, all globally defined widgets in the design are declared. The following lines are typical:

```
Widget exit_button = (Widget) NULL;
Widget help_cascade = (Widget) NULL;
```

Only global widgets are declared here. By default, widgets are local in scope. Local widgets are defined in the function which creates their parent Shell and cannot be referenced elsewhere in your application. To make a widget global, you can:

- Specify it as global on the Core resource panel
- Give it an explicit variable name

Note that the variable names of Application Shells and Top level Shells are always global in SPARCworks/Visual and therefore should not be made local. In addition, any widget which is the target of a link must also be global.

8.7.4 Variable Names

Variable names must be unique. If you “Read” or “Paste” widgets into your design whose variable names duplicate names of existing widgets, SPARCworks/Visual silently removes the duplicate names and assigns new, local, names of the form *widget<n>*.

By convention, variable names of widgets should begin with a lower-case letter. This helps avoid conflict with Motif declarations.

8.7.5 Creation Procedures

By default, SPARCworks/Visual generates a *creation procedure* for each Shell widget in your design. The creation procedures are the heart of the generated code. Each creation procedure does the following:

- Creates the Shell widget itself
- Creates and manages all the children of the Shell and their children
- Sets all hard-wired resources for any child of the Shell
- Adds callbacks and (optionally) links to any child of the Shell which has them

The creation procedures do not display the Shell. Usually, windows are displayed by a function call in the *main()* procedure or in a callback routine.

By default, creation procedures have the form *create_<shell name>*, based on the variable name of the Shell. Your design has two Shells: a Dialog Shell, named *help_window*, and an Application Shell, named *shell_1*. It therefore has two creation procedures: *create_shell_1* and *create_help_window*. You can change the name of a creation procedure in a code prelude, discussed later in this chapter.

create_help_window has the following form:

```
void create_help_window (Widget parent)
{
    . . .
}
```

The function body has function calls which create the Dialog Shell itself:

```
help_window = XmCreateDialogShell ( parent, "help_window", al, ac );
```

Dialog Shells, unlike Application Shells, are dependent on another Shell, *parent*. This dependency results in the icon behavior discussed in the *Additional Windows and Links* chapter.

create_help_window also creates all the Shell's children. The DialogTemplate child, to which you gave an explicit variable name, is created and assigned to a global variable:

```
dialog_2 = XmCreateMessageBox ( help_window, "dialog_2", al, ac );
```

The Label, if you did not name it explicitly, is assigned to a local variable as illustrated below. (Note that the widget number may be different in your code.)

```
widget4 = XmCreateLabel ( dialog_2, "widget4",al,ac);
```

create_shell_1, the creation procedure for your Application Shell, has different arguments because it is a different type of Shell:

```
void create_shell_1 (Display *display, char *app_name, int app_argc,
      char **app_argv)
{
. . .
}
```

See the *Coding Techniques* chapter for a discussion of these arguments.

This function is similar to *create_help_window*, although it is considerably longer as your main window has more child widgets than the help window.

8.7.6 Callback Procedures

The primary module does not include callback functions themselves. However, it does add any callbacks you have specified to each widget's callback list. *create_shell_1* contains the following lines (not necessarily together) which create the *exit_button* and add the *quit* callback.

```
exit_button = XmCreatePushButton ( widget13, "exit_button", al, ac
      );
. . .
XtAddCallback (exit_button, XmNactivateCallback, quit,NULL);
```

An extern declaration of *quit()* is generated earlier in the source file.

The “Show” link on the widget *help_button* inserts an Activate callback to the SPARCworks/Visual function *XDmanage_link*. The code which creates *help_button* and adds a link to it looks much like the code which creates *exit_button* and adds its callback.

```
help_button = XmCreatePushButton ( widget19, "help_button", al, ac
      );
. . .
XtAddCallback (help_button,XmNactivateCallback, XDmanage_link,
      (XtPointer) &xd_links[0] );
```

8.7.7 The Main Program

A minimal *main()* procedure is generated at the end of your primary module if the “Main program” toggle is set in the Generate Dialog.
SPARCworks/Visual’s *main()* procedure does the following things:

- Opens a connection to the X server
- Initializes the X toolkit
- Calls the creation procedure for the first Application Shell
- Calls the creation procedures for all other Shells in the design
- Calls *XtRealizeWidget()* to display the first Application Shell
- Calls *XtAppMainLoop() (*which never returns)
- Calls *exit() (*This call is for neatness only, since this line of code is never executed)

As you have seen, this *main()* procedure is sufficient to run the interface and check its behavior. In many applications, very little additional code is needed in *main()* because most functionality is handled in callbacks. However, if you need to initialize other parts of your application, you must either write a replacement *main()* procedure or add initialization code to the one generated by SPARCworks/Visual. Termination code goes in the callback function which is invoked to exit the application.

♦ **When you have finished examining the generated code, close the file.**

8.8 Code Toggles in the Generate Dialog

Four toggles in the Generate Dialog determine whether the primary module includes the optional sections: “Includes”, “Main program”, “Links”, and “Link functions”. When you generate UIL, there is a fifth code toggle, for a “Callbacks” section, which is discussed below in the *Special Notes for UIL* section on page 167.

8.8.1 Includes

Sets the “Includes” toggle on to generate *#include* statements for the Motif header files needed for your interface.

8.8.2 Main Program

Sets the “Main program” toggle on to generate a minimal *main()* program. In most cases, you want to generate *main()* only once, copy it to another file and edit it for your application. To generate a new primary code module without *main()*, turn the “Main program” toggle off.

8.8.3 Links and Link Functions

Links created by the “Edit links...” command can be generated into your code file in three ways. To understand the possibilities, think of a link as having two parts: the link itself and the *link function*. The link registers a callback on the specified widget. The link function is the actual code which shows, hides, enables, or disables the widget. There are two separate toggles: “Links” and “Link functions”. These two toggles have three meaningful combinations:

- Both toggles on
- Links on, Link Functions off
- Both off

The fourth possible combination (Link Functions on, Links off) is meaningless. Because no code related to links is generated if Links is off, Link Functions has no effect in this case.

If you turn both toggles on, SPARCworks/Visual registers links on widgets which have them and generates the functions themselves into the file. If you are generating a stubs file, you can generate the link functions by setting both toggles on.

If you turn “Links” on and “Link functions” off, SPARCworks/Visual registers links on widgets which have them and generates *extern* references, if needed, to the link functions.

If you turn both “Links” and “Link functions” off, no links are generated. This option is useful for designers who use links only for prototyping. In this strategy, you generate both a link and an “Activate” callback on the same button as you build your interface. You can use the links for prototyping window behavior as you build the design. You then discard the links when you generate code and code the show, hide, enable, or disable behavior into your “Activate” callback along with any other functionality you require.

Links are generated only in the primary module. Link functions are only generated in a primary module or a stubs file.

8.8.4 Callbacks

When you generate C or C++, callbacks are always generated and the “Callbacks” toggle is disabled. Use of the “Callbacks” toggle with UIL is discussed in the section on UIL below.

8.9 Special Notes for UIL

When you work in UIL, the code generation procedure is basically the same as for C. However, because UIL is not as powerful a language as C, there are some features of SPARCworks/Visual which cannot be implemented in UIL. To get the full functionality of your design, you must generate a supplementary C file in addition to your UIL file.

To generate the primary file, select “UIL” from the UIL tear-off menu. To generate the supplementary C file, select “C for UIL”. You must type the name of your compiled UIL file in the “Uid file” field and set the “Uid file” toggle.

When you generate UIL, the “Callbacks” toggle is enabled. This lets you choose whether callbacks are registered in the UIL code or the C code. By default, they are registered in the UIL. If you use client data, however, you should generate the callbacks into the C code, because structure types cannot be defined in UIL.

“Includes”, “Main program”, and “Links” can be generated into the C file but not into the UIL.

UIL is a Motif-specific language and does not work with widgets outside the Motif toolkit. If your design contains a widget from another toolkit, you must use C or C++.

8.10 Other Auxiliary Files

SPARCworks/Visual can also generate some other auxiliary files which are useful in some applications.

8.10.1 Externs File

SPARCworks/Visual can generate a header file with *extern* declarations for all widgets which are global in scope, C++ class definitions and C structure definitions for your design. Global widgets include all widgets which you have explicitly named and those which you have designated as global on the Core resource panel.

To generate an Externs file, click on the “Externs” option on the tear-off menu and specify a filename in the Generate Dialog as usual. By convention, Externs files have the suffix *.h*.

To include the generated Externs file in your primary module, turn on the “Type declarations” toggle in the Generate Dialog and supply the name of the Externs file in the “Type Declarations” field. SPARCworks/Visual then generates an *#include* directive instead of explicit type definitions in the primary module. Global widgets are still allocated in the primary module when you do this.

The Externs file is also useful for including in your stubs file or other code files where you access global widgets or refer to type definitions.

8.10.2 Pixmaps File

This option works similarly to the Externs file. It generates a header file with *static* declarations of all pixmaps in your design. Using this option lets you keep the cumbersome definitions of pixmap structures in a separate file from your primary module.

To generate a Pixmaps file, click on the “Pixmaps” option on the tear-off menu and specify a filename in the Generate Dialog. By convention, Pixmaps files have the suffix *.h*.

To include the generated Pixmaps file in your primary module, turn on the “Pixmaps file” toggle in the Generate Dialog and supply the name of the Pixmaps file in the “Pixmaps file” field. SPARCworks/Visual then generates an *#include* directive instead of explicit pixmap definitions in the primary module.

8.11 Resource File Syntax

Open your X resource file with a text editor and look at it. Part of the file is shown below.

The syntax for generated resource files is as follows:

<application name><widget name>.<resource>: <value>*

For identification purposes, the widget’s variable name (not the widget name) precedes the list of its resources in a comment. If a group of widgets share a widget name, however, only one variable name from the group appears. A comment is also generated for any widget which has no resources generated into the file.

The file fragment below includes only String resources.

```
! procedure_cascade
XDTutorial*procedure_cascade.labelString: Procedures
! widget14
XDTutorial*widget14.labelString: Wash Dishes...
! widget15
XDTutorial*widget15.labelString: Count Money
! exit_button
XDTutorial*exit_button.labelString: Exit
```

```
XDTutorial*exit_button.accelerator: Ctrl<Key>E
XDTutorial*exit_button.acceleratorText: Control+E
! help_cascade
XDTutorial*help_cascade.labelString: Help
XDTutorial*help_cascade.mnemonic: H
! help_button
XDTutorial*help_button.labelString: About This Layout
XDTutorial*help_button.mnemonic: A
! widget21
```

An end user can change any of these strings by editing its value in the X resource file. For example, the second line could be changed to read:

```
XDTutorial*procedure_cascade.labelString: Closing Up
```

Resource values in the X resource file are overridden by values for the same resource in the *.Xdefaults* file in the user's home directory.

8.11.1 Shared Resource Values

To identify a widget completely, X requires a list of all the widget's ancestors in the hierarchy as well as the widget's own name. In the generated X resource file, SPARCworks/Visual uses a wildcard (*) instead of a list of specific ancestors. Thus, each widget is distinguished only by the application name and the widget name, and any widgets which share a widget name, also share any resources generated into the X resource file.

The following lines are taken from SPARCworks/Visual's own X resource file:

```
/* dialog buttons */
visu*apply_button.labelString: Apply
visu*cancel_button.labelString: Close
```

SPARCworks/Visual has several buttons, in several places, which have the widget name *apply_button*. All these buttons share the label string "Apply". Similarly, all buttons with the widget name *cancel_button* share the label string "Close". These strings can be changed on all buttons at once by editing one line of the X resource file.

By contrast, resources generated into the source file are always set separately for each widget, even if widgets share a widget name.

8.12 Control over Generation of Resources

SPARCworks/Visual gives you control over which resources are to be generated into which file. You can include groups of resources by type or individual resources. The following switches control the generation of resources:

- The resource type toggles in the Generate Dialog
- The "Mask widget resources/Mask only global resources" radio buttons
- The masking toggles next to each resource in the resource panels

8.12.1 Resource Type Toggles

The resource type toggles, shown in Figure 8-8, are located in the Generate Dialog control panel.

Figure 8-8 Resource Type Toggles

To include all resources of a given type in your file, turn its toggle on. To exclude that type, turn its toggle off.

Resource types, as defined by Motif, are as follows:

- Strings – Includes null-terminated character strings and *XmString*s
- Fonts
- Colors
- Pixmaps
- Translations – Defined by a command in the Widget Menu and discussed in the *Translations* chapter
- Scalars – Multiple-choice resources, including Booleans. On most resource panels, these are on the "Settings" page
- Integers – Any integer resource setting which is not a scalar
- Reals – Any floating-point numeric resource value

In many cases, designers want the list of hard-wired resource types, and the list of types generated into the X resource file, to be mutually exclusive. If you generate the X resource file immediately after the primary module, SPARCworks/Visual does this for you by reversing the resource type toggle settings. If all toggles except “Strings” were set when you generated your primary module, SPARCworks/Visual turns off all toggles except “Strings” for the X resource file.

8.12.2 Individual Resource Masking Toggles

The unlabeled toggles next to each resource in the resource panels give you control over generation of resources on an individual basis. These toggles work in combination with the resource type toggles in one of two ways, depending on the setting of the “Mask Widget Resources/Mask Only Global Resources” radio buttons, shown in Figure 8-9. The two possible interactions are discussed below.

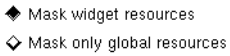


Figure 8-9 Masking Policy Toggles

8.12.3 Masking Policy

These radio buttons determine a masking policy for the resource type toggles. If you set “Mask widget resources”, the type toggles affect all resources. If you set “Mask only global resources”, the type toggles affect only global object resources - that is, font, color and pixmap objects.

If “Mask Widget Resources” is set, each resource is included or excluded based on the settings of both its toggles: the type toggle and the individual resource toggle. A resource is included if exactly one of the toggles is set. It is excluded if both toggles are on or both are off. Another way of saying this is that the toggles in the Generate Dialog establish a general rule; the toggles in the Resource Panels identify exceptions to this rule.

If “Mask Only Global Resources” is set, then the type toggles in the Generate Dialog control only font, color and pixmap objects, which are then controlled by two toggles, as discussed above. All other resources are controlled only by their individual masking toggles. Non-global resources are generated into the source file if their toggles are off and into the resource file if their toggles are on.

8.12.4 Examples

In many cases, designers want to generate most strings into an X resource file so that they can be edited easily. This makes it possible to produce a foreign-language version of the application simply by editing the X resource file. To do this, you generate strings into the X resource file. However, there may be a few strings, such as the company’s address, which you do not want users to be able to change. You can hard-wire these few string resources by setting their individual masking toggles.

Similarly, you might want to let users edit nearly all color resources except for your company colors. To do this, set the masking toggles on the individual resources which control the company colors. When you generate code, generate colors as a group into the resource file. SPARCworks/Visual hard-wires the tagged ones into the source code.

8.12.5 Default Settings

Default resource values, shown in brackets on the resource panels, are never generated into either file. In this case, Motif calculates the resource value at run time. The result may be different from the default value you saw while building the interface, depending on the platform you run the program on. Using default values is often helpful in making your application portable.

8.13 Arranging Your Files

SPARCworks/Visual allows considerable flexibility in arranging files to suit your preference. This flexibility requires some care on your part, since you must include all necessary pieces of code, yet avoid duplication, in order for your application to link successfully.

Another consideration is that your file setup should allow changes to your interface in SPARCworks/Visual after the first pass at generating code. Remember that any changes you make will require regenerating code and resource files. Your files and directories should be set up so that when you regenerate files you do not overwrite any coding work you have done.

With these considerations in mind, this section discusses strategies for organizing your code files.

8.13.1 Using Separate Directories

It is a good practice to keep a separate directory for each SPARCworks/Visual application. Make the directory before you start designing. Save your design file and generate all code files and resource files into that directory.

8.13.2 Keeping Generated Files Unchanged

To avoid errors, do all of your own coding outside the primary module and X resource file generated by SPARCworks/Visual. Code preludes, module preludes and the various switches on the Generate Dialog give you some control over the primary module from within SPARCworks/Visual. Similarly, resource preludes let you adjust the X resource file. If you do not edit these files outside SPARCworks/Visual, you can regenerate them when you make changes in your design without sacrificing any work you have done.

It is often helpful to use the generated *main()* program as a model for your own. The *main()* almost always needs to be edited. To do this without touching the generated code, copy *main()* to a separate file before editing it. Use the following steps:

1. **Generate code including *main()*, as you have just done.**
2. **Generate a stubs file.**
3. **Copy the *main()* program from your primary module to the stubs file. (Alternatively, you can copy *main()* to another file.) You must also copy the few lines of variable declarations just above *main()*.**
4. **Add your own code to the *main()* program generated by SPARCworks/Visual.**

5. **Regenerate the interface code, this time turning off the “Main program” toggle. Recompile both files and relink.**

8.13.3 Stubs File

Unlike other generated files, the stubs file is meant to be edited. SPARCworks/Visual will preserve changes to stubs files on re-generation. See the *Incremental Stubs File Generation* section of the *Coding Techniques* chapter on page 192 for more details.

8.13.4 Where To Put Links

If your application uses links, you must:

- Generate links into the primary module
- Generate link functions into exactly one file, either a primary module or a stubs file

To generate link functions, you must set *both* the “Links” and “Link functions” toggles for that file.

If your application uses generated code from more than one design file, you should generate links into all the primary modules but generate the link functions into only one file.

8.13.5 Where to Put Includes

If your *make* procedure involves compiling the primary module and the application code separately, you should turn on the “Includes” toggle both for the main C module and for the stubs file. This procedure was followed in the tutorial.

Another strategy involves writing a *#include* directive to include the generated code in your application code file and compile all the code together. If you do this, you should turn on “Includes” only once for the primary module and turn it off when you generate the stubs file.

8.13.6 Accessing Widgets in Callbacks

All callbacks are passed the address of the widget to which they belong. If you want the callback function to access other widgets in your design, you can do it in one of two ways:

- Pass the other widgets as part of the client data structure
- While building your design, use the Core resource panel to designate the other widgets as global. They are then generated into your Externs file, which you can *#include* in your callbacks file

8.14 Customizing the Generated Files: Preludes

SPARCworks/Visual lets you specify lines of code, called *preludes*, to be inserted into the primary module or X resource file at specific points. You must specify preludes from within SPARCworks/Visual before you generate the files. Preludes are saved with your design file and are generated each time you regenerate the files. There are several types of preludes, distinguished by where the code is inserted.

8.14.1 Module Preludes

The “Module prelude...” command in the “Module” menu lets you enter either a *heading prelude*, a *module prelude*, or a *resource prelude* in the dialog shown in Figure 8-10. To enter one of these, select the appropriate page from the option menu and type or edit lines of code in the text box. Click on “Apply” to register your module prelude or module heading.

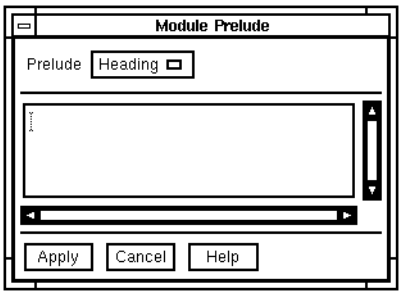


Figure 8-10 Module Prelude Dialog

The heading prelude is inserted at the beginning of the primary module and at the beginning of the stubs file. Typically, the module heading contains a comment with information such as the program name, ID, or version number.

The module prelude is inserted just after SPARCworks/Visual's generated *#include* statements, if any. The module prelude can be used to supply *#define* or *#include* statements or *extern* declarations which are needed by your code preludes. The module prelude is generated only into the primary module, not the stubs file.

The resource prelude is inserted at the beginning of the X resource file to specify loose resource bindings. Use the following syntax:

```
ApplicationName*resource: value
```

For example:

```
XDTutorial*background: white
```

Although these resource bindings apply to all widgets in the application, they are overridden by more specific resource settings on individual widgets or groups of widgets with a common name.

8.14.2 Code Prelude Dialog

While module preludes apply to the whole module, code preludes can be assigned to individual widgets. To specify code preludes, pull down the Widget Menu and select "Code preludes...". This produces the dialog shown in Figure 8-11.

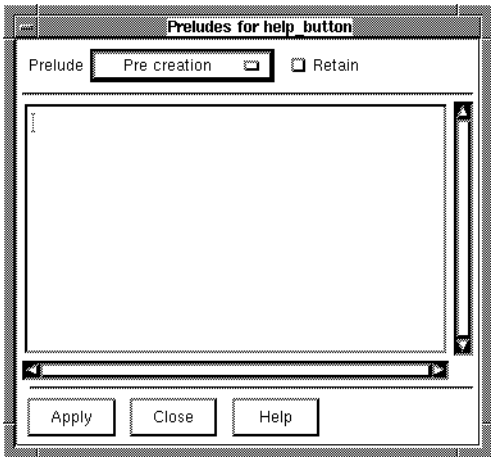


Figure 8-11 Preludes Dialog

Two kinds of code preludes can be used with C: *pre-create* and *pre-manage.* There are also three kinds of code preludes related to public, private and protected types for use with C++. For additional information, see the *C++ Code and Reusable Definitions* chapter.

8.14.3 Pre-creation Prelude

Pre-creation preludes are inserted into the generated code before the selected widget is created.

If the selected widget is not a Shell widget, the pre-create prelude is inserted in the creation procedure for the widget's parent Shell and you can provide any code without restriction. A common use of pre-create preludes is setting resources which can only be set at widget creation time.

If the widget is a Shell widget, the pre-create prelude becomes the creation procedure header. It can also include lines of code above the header, such as variable declarations. You can use a pre-create prelude on a Shell widget to:

- Pass additional parameters to the creation procedure for use in code preludes
- Change the name of the creation procedure

There are special rules which must be followed when you write pre-create preludes for Shell widgets. The rules, and ideas for using preludes with Shells, are discussed in the *Coding Techniques* chapter.

8.14.4 Pre-manage Prelude

The pre-manage prelude is inserted slightly later in the generated code, just before the widget's callbacks are added. One use of this prelude is to set up client data for the callbacks. Other uses include setting the value of a Text widget, filling a ScrollingList, adding buttons from a file, or any other dynamic initializations.

Because shell widgets are not managed, they cannot have pre-manage preludes.

8.14.5 The Retain Button

The Code Preludes Dialog has a "Retain" toggle button which works like the "Retain" toggle button on the Callbacks Dialog. "Retain" lets you apply the same list of preludes to multiple widgets without retyping them. Type and apply your preludes to one widget, turn on the "Retain" toggle, then select another widget and apply to transfer the list of preludes to the new widget.

8

Coding Techniques 9

9.1 Introduction

This section explains the various techniques which can be used when writing code to link the SPARCworks/Visual generated user interface code to your application and to further customize the generated code.

9.2 Callback Functions

The creation procedures generated by SPARCworks/Visual create the widgets for the dialogs of your application and set their initial resource values. However, it is the callback functions that make the application work.

Most callback functions have a similar structure. A typical callback function does some or all of the following:

- Extracts information from widgets, such as the text in a Text widget or the state of a ToggleButton
- Uses this information as parameters for calls to application functions
- Uses the results of these functions to change widget attributes. These attributes include not only values (such as the text in a Text widget) but also sensitivity (responsiveness to user input), visibility and even existence

9.2.1 Callback Function Parameters

A callback function receives three parameters:

- The widget from which the callback was invoked
- The call data
- The client data

The call data is a pointer to a data structure defined by the widget developer. Call data structures are described in the documentation from Motif or the developer of the widget toolkit.

The client data is a pointer that you can use to pass the address of any variable or structure. When you register a callback, you can specify the value for the client data parameter that is passed to the callback function.

In SPARCworks/Visual, the name of the client data is specified in the callback dialog as a single optional parameter of the callback function. This can be a pointer to a structure, which can be defined and initialized in a suitable prelude. For example, a typical prelude might be:

```
/* Pre-manage prelude for main dialog Shell */
       /* Define and initialize client data for the rungrep callback
       */
static rcdata_t rcdata = {
    &hitstring,
    &errorshell,
    &errorform,
    &errortext,
    &mainshell
};
/* End of Shell pre-manage prelude */
```

The callback is specified as:

```
rungrep((XtPointer)&rcdata)
```

The declaration of the structure *rcdata_t* would normally be in a header file that would be included in the generated code (via an in the module prelude) and in the callback function module. The callback function can then cast the client data to *(rcdata_t *)* and so access the data.

Note that the structure *rcdata* is defined to contain pointers to the widget variables, rather than the values of the variables themselves. This lets *rcdata* be initialized before the widgets are created. You can also define a structure into which the values of the widget variables are copied. However, this cannot be initialized until all the widgets have been created, which can be tricky.

9.2.2 Callbacks in C++

Ideally it would be desirable to add class member functions to widgets as callback functions. Unfortunately this is not possible because callback functions are called by a C library and they cannot provide the instance context (the *this* pointer) required by a class member function. SPARCworks/Visual provides an automatic way of calling a class member function from a callback. These are called callback methods and are discussed in the *Callback Methods* section of the *Structured Code Generation and Reusable Definitions* chapter on page 205.

9.3 Accessing Widgets

The functions that are used to manipulate widgets all take a parameter of type *Widget*. A *Widget* is a pointer to an opaque data structure (one you are not supposed to look inside). In the SPARCworks/Visual generated code, the value of this pointer is stored in the variable whose name is determined by the variable name of the widget. To access a widget, you must have access to this variable, or at least its value.

9.3.1 Global Widget Variables

The simplest technique is to have SPARCworks/Visual define the widget variables as global. You can then access them from a callback function by declaring them as *extern* in the callback function module. Including SPARCworks/Visual's generated Externs header does this for you.

SPARCworks/Visual declares named widgets as global by default. You can change this behavior by setting the Storage Class of the widget in the Core resource panel.

The strength of the global variable approach is its simplicity. However, having many global variables does nothing for the structure of your program and you must pay attention to naming conventions to ensure meaningful names and avoid duplicates.

9.3.2 Inclusion of Generated Code

You can reduce the need for global variables by including the primary module in the callback function module, using *#include*. The primary module should be generated without includes of the X and Motif header files.

If you do this, SPARCworks/Visual still declares named widgets as global. You may want to change their storage class to static, which makes them local to the callback function module.

This technique works well where a callback function needs access to widgets that are all or mostly within a single design. In more complex situations, you can add accessor functions to the callback function module. A callback function that needs to manipulate a widget which is local to another callback module can do so via the accessor functions.

9.4 Manipulating Widgets

There are many ways you can manipulate a widget. This section outlines a few of them. It is not a detailed description, but is only intended to point you to the appropriate functions and their documentation.

9.4.1 Toolkit Convenience Functions

The Motif toolkit provides a large number of convenience functions for getting and setting attributes of some widgets. These are all named after the widget class that they affect, such as *XmTextSetString(), XmTextGetString(), XmToggleButtonGetState()*. These are documented in the *Motif Programmer's Reference*.

Convenience functions are the first place to look. They are the easiest to use and are likely to be efficient.

One point to note is that convenience functions take a *Widget* parameter and expect this to be a pointer to a widget of the appropriate class. If the widget is of the wrong class, they commonly core dump. There are also both widget and gadget versions of some of the convenience functions and you may get a core dump if you use the wrong one.

9.4.2 Setting and Getting Resources

If there is no convenience function, you may have to get or set one or more of the resources of the widget directly using *XtGetValues()* or *XtSetValues()*. This is fundamental to widget programming and any book on Xt or Motif should cover it adequately.

Not all resources can be set after a widget has been created. The *Motif Programmer's Reference* documents the access controls on each resource of every widget class.

9.4.3 Enabling and Disabling Widgets

To disable a widget (that is, to make it insensitive to user input), or enable it again, use *XtSetSensitive()*. You should not set the resource *XmNsensitive* directly.

When a widget becomes insensitive, so too do all its descendants. Insensitive widgets are usually grayed out.

If you make a Text or TextField widget insensitive, the user cannot use key input to pan and scroll the text and so has only a limited view. It may be better to set the resource *XmNeditable* to *False*.

9.4.4 Showing and Hiding Widgets

There are two ways to make a widget appear or disappear: managing and mapping.

If a widget is unmanaged, it is as if it does not exist at all. Its parent does not reserve any space for it and it is not visible on the screen. A widget is unmanaged using *XtUnmanageChild()* or *XtUnmanageChildren()* and managed using *XtManageChild()* or *XtManageChildren()*. SPARCworks/Visual generates code to manage widgets after they have been created, but the Managed toggle in the Core resource panel changes this.

If a widget is managed but not mapped, its parent reserves space for it. However, it is still not visible; there is a blank hole. Widgets are normally mapped automatically when they become managed. This is controlled by the resource *XmNmappedWhenManaged*.

Mapping and unmapping is commonly used to change the visibility of widgets within a dialog without causing its layout to change. Managing and unmanaging cause layout changes.

You can make a complete dialog appear or disappear by managing or unmanaging the child of the Dialog Shell. If the dialog uses a Top level Shell, use *XtPopup()* and *XtPopdown()* on the Shell instead.

9.4.5 Creating and Destroying Widgets

SPARCworks/Visual generates code to create the widgets for your dialogs. The default *main()* program calls all the creation functions at start-up time. Since widget creation is relatively expensive, this may cause an unacceptable delay. It is common practice to defer creation of a dialog until the first time it is popped up. A static Boolean flag in the callback function that performs the popup can be used to determine if the dialog has already been created.

As well as generating code to create complete dialogs, SPARCworks/Visual can generate creation functions for dialog fragments, as described in the *Children Only Place Holders* section of the *Structured Code Generation and Reusable Definitions* chapter on page 209. You can call these from a callback function, such as to create another instance of some reusable component.

To destroy a widget (and all its children), use *XtDestroyWidget()*. It is inefficient to destroy a widget and then recreate it; you should unmanage it, then manage it again when it is needed.

9.5 Shell Preludes

The code preludes for a Shell differ slightly from those of normal widgets. The pre-create prelude is used to replace the function header for the Shell's creation procedure, which lets you pass in extra parameters such as for use as client data or in other code preludes.

A Shell's pre-manage prelude is inserted just after the local declarations in the procedure.

The generated body of the procedure refers to one or more variables, which, in the default procedure heading, are passed as parameters. While these variables must be in scope, you can choose to pass them as parameters or declare them as global variables. The following variables must be in scope:

Required for Application Shell widgets:

```
Display *display;
char *app_name;
int app_argc;
char **app_argv;
```

Required for Dialog Shell or Top level Shell widgets:

```
Widget parent;
```

In C for UIL the following are also required:

```
MrmHierarchy hierarchy_id;
MrmCode *class;
```

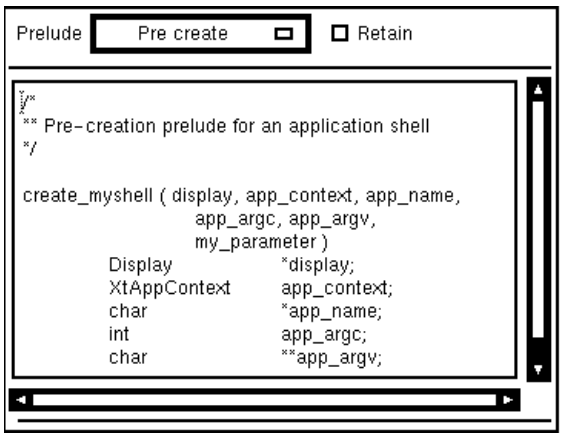


Figure 9-1 Example of Pre-Creation Code for Application Shell

If you do not provide a pre-create prelude for a Shell widget, the creation procedure name defaults to *create_<variablename>*, as explained above, with the compulsory parameters as the only parameters.

Note that if you provide a pre-create prelude for a Shell, the call of the creation procedure in the generated default *main()* program is unlikely to be correct.

9.6 The Managed Toggle

By default all widgets are generated as managed, with the exception of the "Apply" button in a SelectionBox that is not a child of a Dialog Shell. This state can be modified using the "Managed" toggle in the "Code generation" page of the Core resource panel. Usually this just means that the code to manage the widget is omitted from the generated code. For widgets or gadgets that are components of composite widgets, the generated code explicitly unmanages the widget if the toggle is off, since the toolkit always creates these widgets as managed. For the "Apply" button of a Selection Box, code to explicitly manage the button is generated if the toggle is on.

9.7 Drag and Drop

Motif 1.2 provides a sophisticated drag and drop mechanism that lets applications communicate data via the X selection mechanism. SPARCworks/Visual provides some simple support to let you specify drop sites in your application. Because the initialization of a drag is a dynamic function that would normally be done from within a callback or action function, SPARCworks/Visual does not provide any explicit support.

Basically, a drop site is a widget that is prepared to receive certain types of data from the transfer mechanism. SPARCworks/Visual provides its support through the Drop site page in the Core resource panel.

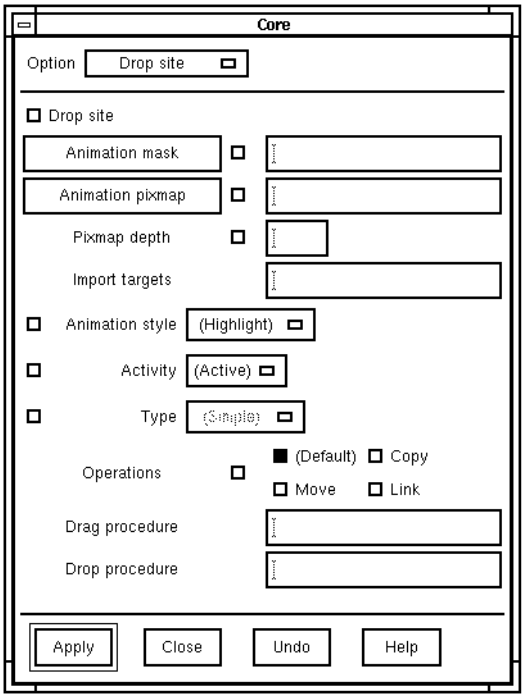


Figure 9-2 The Drop Site Page

To designate a widget as a drop site, simply set the “Drop site” toggle on and specify the import targets and drop procedure. The “Import targets” field is a list of strings that are converted into atoms to designate types that can be handled by the drop procedure. The list is specified as strings separated by commas or spaces.

By default Motif makes Label (and derived) widgets start a drag operation to transfer the *labelString* or *labelPixmap* if Button 2 is pressed over them. SPARCworks/Visual takes advantage of this by adding a drop procedure to the drop site widget that imports these types if specified in the import targets. The following tutorial lets you see how the drop site operates.

1. **Create as dialog containing an Application Shell with a RowColumn containing two Push Buttons.**

2. **Name the widgets: shell, rowcolumn, button1 and button2 respectively.**

3. **Pop up the Drop site page for button1.**

4. **Set the drop site toggle on and set animation style to shadow in.**

5. **In the "Import targets" field, type** `COMPOUND_TEXT`

6. **Press Apply but do not close the dialog.**

SPARCworks/Visual warns you that you have not specified a drop procedure, which you must do for your application to work.

7. **In the Drop procedure field, type:** `drop_button1`

The drop and drag procedure fields specify the names of functions to be called to handle the drop and dynamic drags respectively.

8. **Try dragging the text from any label in the tool bar (press button 2 and drag) across button1 in the dynamic display window.**

The button is shadowed in to indicate that it is a valid drop site for the target being dragged.

9. **Release the mouse button to drop the text into the widget.**

The drop procedure provided by SPARCworks/Visual simply copies the label into the widget.

10. **Select button2 and repeat Step 4.**

11. **In the "Import targets" field, type** `PIXMAP`

12. **In the Drop procedure field, type:** `drop_button2`

13. **Try dragging a pixmap from the tool bar across button2 in the dynamic display window.**

For further examples of using drop sites and for information on starting drags, refer to the Motif documentation.

Code is generated for C and C++, with a call to *XmDropSiteRegister()* being generated for widgets that are not normally drop sites. Text widgets are drop sites by default, which can import *COMPOUND_TEXT*. This can be disabled by setting the drop site toggle off, or modified by simply changing the appropriate resources.[1]

```
extern void drop_button1 (Widget, XtPointer, XtPointer );
extern void drop_button2 (Widget, XtPointer, XtPointer );
```

```
shell_p shell = (shell_p) NULL;
shell_p create_shell (Display *display, char *app_name, int
       app_argc, char **app_argv)
{
...
button1 = XmCreatePushButton ( rowcolumn, "button1", al, ac );
XtSetArg(al[ac], XmNanimationStyle, XmDRAG_UNDER_SHADOW_IN); ac++;
/* Set up the import targets atom list for button1 */
atom_list = (Atom *) XtMalloc ( 1 * sizeof ( Atom ) );
atom_list[0] = XmInternAtom ( display, "COMPOUND_TEXT", False );
XtSetArg(al[ac], XmNimportTargets, atom_list); ac++;
XtSetArg(al[ac], XmNnumImportTargets, 1); ac++;
XtSetArg(al[ac], XmNdropProc, drop_button1); ac++;
/* Register the drop site for button1 */
XmDropSiteRegister ( button1, al, ac );
ac = 0;
XtFree ( (char *) atom_list );
button2 = XmCreatePushButton ( rowcolumn, "button2", al, ac );
XtSetArg(al[ac], XmNanimationStyle, XmDRAG_UNDER_SHADOW_IN); ac++;
/* Set up the import targets atom list for button2 */
atom_list = (Atom *) XtMalloc ( 1 * sizeof ( Atom ) );
/* pixmap has pre-defined atom */
atom_list[0] = XA_PIXMAP;
XtSetArg(al[ac], XmNimportTargets, atom_list); ac++;
XtSetArg(al[ac], XmNnumImportTargets, 1); ac++;
XtSetArg(al[ac], XmNdropProc, drop_button2); ac++;
/* Register the drop site for button1 */
XmDropSiteRegister ( button2, al, ac );
ac = 0;
XtFree ( (char *) atom_list );
...
```

1. The comments which appear in the code are not generated.

```
}
```

You must write the drop procedures to handle the transfers.

9.8 Incremental Stubs File Generation

Callback stubs are generated for all callbacks and callback methods specified in your design. These are generated into a separate source file called a *Stubs File.* Along with the callback stubs, SPARCworks/Visual generates special comments.

When you subsequently generate the same stubs file, SPARCworks/Visual reads the special comments in order to work out which callbacks and methods have already been generated. *In this way you can add your own code to the stub and it will not be overwritten.*

SPARCworks/Visual then appends any new callbacks or methods to the end of the stubs file. Whenever a new stubs file is generated, the old version is copied to a file with the name you have specified and a `.bak` extension. Below is an example stubs file from the cross platform tutorial:

```
/*
** Generated by SPARCworks/Visual
*/
```

The Beginning of the Prelude

```
/*
** SPARCworks/Visual generated prelude.
** Do not edit lines before “End of SPARCworks/Visual generated
      prelude”
** Lines beginning ** SPARCworks/Visual Stub indicate a stub
** which will not be output on re-generation
*/
/*
**LIBS: -lXm -lXt -lX11
*/


#include <afxwin.h>
#include <afxext.h>
```

The End of the Prelude

```
/* End of SPARCworks/Visual generated prelude */
```

Special Comment to Indicate a Stub

```
/*
** SPARCworks/Visual Stub about_sh_c::DoSetText
*/

void
about_sh_c::DoSetText ( )
{
}
```

Special Comment to Indicate a Stub

```
/*
** SPARCworks/Visual Stub shell_c::DoExit
*/

void
shell_c::DoExit ( )
{
}
```

Special Comment to Indicate a Stub

```
/*
** SPARCworks/Visual Stub scrolled_win_c::DoInput
*/
void
scrolled_win_c::DoInput ( )
{
}
```

9.8.1 The Stubs File Comments

At the beginning of the file there is a prelude which SPARCworks/Visual reads and, effectively, throws away. The prelude is always regenerated anew. Before every stub SPARCworks/Visual generates a comment giving the name of the callback or method. In this way SPARCworks/Visual can calculate which stubs it still needs to generate, having read the existing stubs file. *You should not alter these comments in any way unless you wish* SPARCworks/Visual *to regenerate the stub.*

9.8.2 Regeneration of Callback Stubs

If you wish SPARCworks/Visual to regenerate one of the stubs, simply remove the comment preceding the stub and the stub itself. If you remove only one or the other, one of the following will occur:

- If you remove the stub but leave the comment, no stub will be generated
- If you remove the comment but leave the stub, you will have two copies of the stub

Remember that regeneration of a stub will lose the contents of the routine.

SPARCworks/Visual will not remove old stubs even though a special comment no longer matches a callback or callback method.

9.8.3 Regeneration of Whole File

You may wish SPARCworks/Visual to regenerate the whole file anew if, for example, you have deleted some callbacks or changed some names, the old ones are still being generated and the file is starting to become full of redundant code. In order to do this, simply remove, or change the name of, the stubs file. If SPARCworks/Visual cannot find a file with the name you have specified for the stubs file, it will generate a new file.

Structured Code Generation and Reusable Definitions 10

10.1 Introduction

This chapter describes how to use SPARCworks/Visual's structured code generation facilities to create reusable widget hierarchies. These reusable hierarchies, known as *definitions*, appear on the widget palette and can be added to the hierarchy like any other widget. This chapter is intended as an overall description of the material that is presented in the following tutorials.

10.2 Structured Code Generation

SPARCworks/Visual provides controls for structuring your generated code so that it is more flexible and can be reused more easily. Before reading this section, you should review the structure of the default generated code in the *Analysis of the Primary Module* section of the *Generating Code* chapter on page 161. In particular, note that the default code has a single creation procedure for each Shell in the design. Widgets are declared as local if they have not been named and global if they are named or are Application Shells.

The structured code controls let you:

- Designate any widget in the hierarchy to have its own creation function that returns the widget, including its descendants
- Designate any widget to have its own creation function that returns a structure containing the widget and its named descendants
- Designate any widget to be defined as a C++ class with descendant widgets as members

- Designate a widget as a place-holding container that serves only to house a collection of child widgets
- Explicitly specify a widget as global, local, or static

SPARCworks/Visual's controls for structuring code are located on the "Code generation" page of the Core resource panel.

10.3 Function Structures

The simplest case of structured code generation is to designate a widget as a *function structure*. This makes SPARCworks/Visual generate a separate function that creates that widget and its descendants. This function is called by the creation procedure for the enclosing widget.

To do this, select the "Code generation" page of the Core resource panel and select "Function" from the "Structure" option menu.

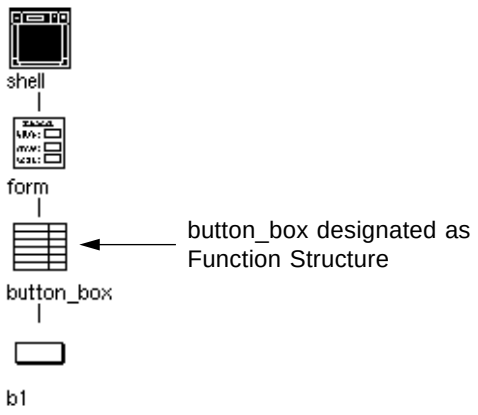


Figure 10-1 Example: Structure

The hierarchy shown in Figure 10-1 produces the following generated code, slightly simplified for clarity:[1]

```
Widget shell = (Widget) NULL;
Widget form = (Widget) NULL;
```

1. The comments describing the functions and procedures are not generated.

```
Widget button_box = (Widget) NULL;
Widget b1 = (Widget) NULL;

/* This is the creation function for the button_box. */
Widget create_button_box (Widget parent)
{
    Widget children[1];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */
    Widget button_box = (Widget)NULL;

    button_box = XmCreateRowColumn ( parent, "button_box", al, ac );
    b1 = XmCreatePushButton ( button_box, "b1", al, ac );
    children[ac++] = b1;
    XtManageChildren(children, ac);

/* The button box is created, but not managed, and returned. */
    return button_box;
}

/* The creation function for the Shell calls the button box creation function. */
void create_shell (Widget parent)
{
    Widget children[1];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */

    XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
    shell = XmCreateDialogShell ( parent, "shell", al, ac );
    ac = 0;
    XtSetArg(al[ac], XmNautoUnmanage, FALSE); ac++;
    form = XmCreateForm ( shell, "form", al, ac );
    ac = 0;
    button_box = create_button_box ( form );
```

```
/* The constraint resources for the button box are set in the parent's creation
function. */
    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
    XtSetValues ( button_box,al, ac );
/* The button box is managed at this point. */
    children[ac++] = button_box;
    XtManageChildren(children, ac);
}
```

This module now has two functions: one (*create_shell()*) for creating the whole hierarchy and one (*create_button_box()*) for creating the button box.

10.4 Data Structures

The next type of code structuring is the *data structure*. This is similar to a function structure, in that SPARCworks/Visual generates a separate creation procedure for the widget and its descendants. When a widget is designated as a data structure, SPARCworks/Visual also generates a *typedef* for a structure including that widget and its children. The creation procedure for the widget creates and sets up that type of structure and returns a pointer to it. A deletion function (*delete_<widget_name>*) is also generated so that the allocated memory can be freed.

To designate a widget as a data structure, select the "Code generation" page from the Core resource panel and select "Data structure" from the "Structure" option menu.

Using the same hierarchy as shown above, but with *button_box* designated as a data structure, the following code is produced, slightly simplified for clarity:[1]

```
/* First the type declarations are generated for the data structure. */
typedef struct button_box_s {
    Widget button_box;
    Widget b1;
```

1. The comments describing the functions and procedures are not generated.

```
} button_box_t, *button_box_p;
Widget shell = (Widget) NULL;
Widget form = (Widget) NULL;
button_box_p button_box = (button_box_p) NULL;

/* The creation procedure returns a pointer to a button_box structure. */
button_box_p create_button_box (Widget parent)
{
    Widget children[1];      /* Children to manage */
    button_box_p button_box = (button_box_p)NULL;

/* Space is allocated for the structure and the fields are filled in. */
    button_box = (button_box_p) XtMalloc ( sizeof ( button_box_t ) );
    button_box->button_box = XmCreateRowColumn ( parent,
       "button_box", al, ac );
    button_box->b1 = XmCreatePushButton
       ( button_box->button_box, "b1", al, ac );
    children[ac++] = button_box->b1;
    XtManageChildren(children, ac);
    return button_box;
}

/* A deletion function is supplied to free the allocated memory. */
void delete_button_box ( button_box_p button_box )
{
    if ( ! button_box )
        return;
    XtFree ( ( char * )button_box );
}

/* Again, the Shell creation function calls the button box creation function. */
void create_shell (Widget parent)
{
    Widget children[1]; /* Children to manage */
    Arg al[64]; /* Arg List */
    register int ac = 0; /* Arg Count */
    shell = XmCreateDialogShell ( parent, "shell", al, ac );
```

```
    form = XmCreateForm ( shell, "form", al, ac );
    button_box = create_button_box ( form );
    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
/* The button_box widget has to be referenced inside the structure. */
    XtSetValues ( button_box->button_box,al, ac );
    ac = 0;
    children[ac++] = button_box->button_box;
    XtManageChildren(children, ac);
    ac = 0;
}
```

10.5 C++ Classes

The use of *C++ classes* is very similar to data structures. SPARCworks/Visual does not wrap each widget in the hierarchy with a C++ class, but instead designates sections of the hierarchy as classes in their own right. Each widget designated as a C++ class has a class defined for it. Its named descendant widgets become members of that class and widget creation and widget destruction methods are supplied. In addition, if the class contains members that are themselves (pointers to) classes, a constructor and destructor method is generated to create and destroy these members. Note that the widgets are not created at the time of the class instance but by an explicit call to the widget creation function. Similarly, destroying the class instance does not destroy the widgets.

To designate a widget as a C++ class, select the “Code generation” page of the Core resource panel and select “C++ class” from the “Structure” option menu. Note that if you designate a widget as a C++ class, then generate C, the widget is treated as a data structure.

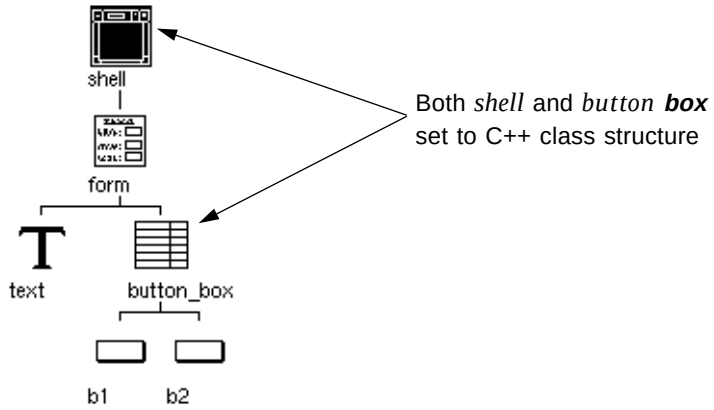


Figure 10-2 Example: C++ Class Structures

The C++ code generated from this example is shown below, simplified for clarity:[1]

```
// Classes are declared for button_box and shell.
class button_box_c: public xd_XmRowColumn_c {
public:
    virtual void create (Widget parent, char *widget_name = NULL);
protected:
    Widget button_box;
    Widget b1;
    Widget b2;
};

typedef button_box_c *button_box_p;

// The shell class has constructor and destructor functions because it
// contains a pointer to class (or data structure) member.
class shell_c: public xd_XmDialog_c {
public:
```

1. The comments describing the functions and procedures are not generated.

```
    virtual void create (Widget parent, char *widget_name = NULL);
    shell_c();
    virtual ~shell_c();
protected:
    Widget shell;
    Widget form;
    Widget text;
    button_box_p button_box;
};

typedef shell_c *shell_p;

shell_p shell = (shell_p) NULL;
```

// The creation function now becomes a method of the class. This method
// is declared public in the SPARCworks/Visual base class, which is supplied
// with the release.

```
void button_box_c::create (Widget parent, char *widget_name)
{
    Widget children[2];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */

    if ( !widget_name )
        widget_name = "button_box";

    button_box = XmCreateRowColumn ( parent, widget_name, al, ac );
```

//_xd_rootwidget is a protected member of the class that stores the widget
// that is at the root of the sub-hierarchy. This lets the base class
// operate on the widget.

```
    _xd_rootwidget = button_box;
    b1 = XmCreatePushButton ( button_box, "b1", al, ac );
    b2 = XmCreatePushButton ( button_box, "b2", al, ac );
    children[ac++] = b1;
```

```
    children[ac++] = b2;
    XtManageChildren(children, ac);
    ac = 0;
}
// The Shell's creation method calls that for the button box.
void shell_c::create (Widget parent, char *widget_name)
{
    Widget children[2];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */

    if ( !widget_name )
        widget_name = "shell";

    XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
    shell = XmCreateDialogShell ( parent, widget_name, al, ac );
    ac = 0;
    _xd_rootwidget = shell;
    XtSetArg(al[ac], XmNautoUnmanage, FALSE); ac++;
    form = XmCreateForm ( shell, "form", al, ac );
    ac = 0;
    text = XmCreateText ( form, "text", al, ac );
// The button box class is instantiated in the constructor method and so at
// this point only the widgets need to be created.
    button_box->create ( form, "button_box" );

    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac], XmNtopWidget, button_box->xd_rootwidget());
       ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
    XtSetValues ( text,al, ac );
    ac = 0;
```

```
    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
    XtSetValues ( button_box->xd_rootwidget(),al, ac );
    ac = 0;
    children[ac++] = text;
    children[ac++] = button_box->xd_rootwidget();
    XtManageChildren(children, ac);
    ac = 0;
}

shell_c::shell_c()
{
// Instantiate the child classes.
    button_box = new button_box_c;
}
shell_c::~shell_c()
{
// Free the child classes.
    delete button_box;
}
```

If a widget is designated a C++ class and C code is generated, the widget is treated as if it were a data structure.

By default, the generated class is derived from one of the supplied SPARCworks/Visual base classes. You can override this by specifying the base class in the field below the C++ Access option menu. The SPARCworks/Visual base classes supplied with the release provide minimal support, sufficient for the generated code to execute correctly. You can modify and extend those classes to provide reusable methods that suit your approach to GUI development.

Descendant widgets appear as protected members of the class if they are named, or if they are themselves data structures or C++ classes. It is therefore important to name the C++ class widget itself and any of its descendants that you want to access as class members. You can alter the default access control by selecting the required level (Public, Protected, or Private) from the C++ Access option menu.

Using an unnamed widget for the C++ class widget itself does not cause an immediate error. However, this is not recommended as numbers assigned by SPARCworks/Visual can change when you edit your hierarchy.

10.5.1 Callback Methods

The X toolkit functions which invoke callback functions expect a callback function in the following form:

```
void my_callback (Widget, XtPointer, XtPointer)
```

An ordinary member function is not suitable as a callback function because the C++ compiler passes it an extra first parameter - the *this* pointer - that lets it find the instance data for the object. If you use an ordinary member function as a callback function, the member function interprets the widget pointer as the instance data pointer and does not work as expected.

SPARCworks/Visual uses a common technique to work around this. A static member function (which does not expect a *this* pointer) is declared and used as the callback function:

```
static void my_callback (Widget, XtPointer client_data, XtPointer
       call_data)
```

The client data parameter is used to pass in a pointer to the instance. The static member function merely calls an ordinary non-static member function using that instance pointer and passes on the widget and call data parameters. The non-static member function has the following form:

```
virtual void my_callback (Widget, XtPointer call_data)
```

SPARCworks/Visual generates both function declarations, all the code for the static callback function and a stub for the regular member function which is written by you. Note, because this function is declared as *virtual*, you can override it in a derived class to modify the behavior. For a discussion of this technique, see "*Object-Oriented Programming with C++ and OSF/Motif*" by Douglas Young.

10.5.2 Callback Method Access Control

By default, callback methods have public access. You can control the access for individual callback methods using the “Access” option menu on the Callbacks dialog. Select “Public”, “Private”, or “Protected”.

10.5.3 Pure Virtual Callback Methods

You can set the “Pure virtual” toggle to declare the non-static member function as pure virtual. For example, if you set this toggle for a callback method *OnNew()* in a menubar class, SPARCworks/Visual would declare the method as:

```
class menubar_c: public xd_XmMenuBar_c {
...
public:
...
    virtual void OnNew( Widget, XtPointer ) = 0;
};
```

Because the function is pure virtual, you do not have to provide an implementation of *menubar_c::OnNew() and menubar_c* becomes an abstract class. That is, you cannot create an instance of *menubar_c* but only use it as a base class for others.

10.5.4 Editing Callback Methods

Callback method attributes are set initially when the callback method is first specified. This can be done on any widget that has a class ancestor. However, they can only be changed on the root widget for the class. For example, the *OnNew()* callback method of the class *menubar_c* can only be edited via the *menubar* widget itself.

Method declarations are accessed for a Class structured widget from the Declare Methods callback button in the Core resources dialog, or from the Method Declarations item in the widget menu.

10.5.5 Deleting Callback Methods

When you remove a callback method from a widget you are only removing the use of the method. If you want to remove the declaration as well you must remove it from the Method Declarations of the enclosing class.

10.5.6 Method Preludes

You can add additional data or function members to a C++ class using the "Code preludes" dialog. Select "Public methods", "Protected methods", or "Private methods" and type your declarations into the text box. C++ code preludes are generated into the class declaration, both in the primary module and in the Externs file.

10.5.7 Creating a Derived Class

To add a function to a class it is often better to write a new class derived from the generated class. The logical gap between the subclass and generated base class can be used to add members and provide implementations for virtual functions.

By default, SPARCworks/Visual derives the name of a C++ class from the variable name of the root widget and so the class for the widget *menubar* is *menubar_c*:

```
class menubar_c: public xd_XmMenuBar_c {
...
};
```

When SPARCworks/Visual generates code to create an instance of the class, it uses the same name:

```
menubar = new menubar_c;
```

You can change the default behavior so that SPARCworks/Visual declares the generated class under one name and creates the instance under another. For example:

```
menubar = new mymenubar_c;
```

To make this change, use the "Instantiate as" field on the Code Generation page of the Core resource panel.

10.5.8 Modifying the Base Classes

By default, SPARCworks/Visual derives a generated class from a base class appropriate to the type of the root widget. For example, a class with a MenuBar at the root of its widget hierarchy is derived from *xd_XmMenuBar_c*. The name of the base class can be changed in the Core resource panel.

The SPARCworks/Visual distribution contains a sample implementation of a set of base classes. These can be used as they stand or modified to add extra functionality appropriate to a particular application area.

A Makefile is included to build the sample base classes. SPARCworks/Visual makes two assumptions about the base classes:

There is a data member *_xd_rootwidget* of type *Widget*.

There is an accessor function *xd_rootwidget()* that returns the value of *_xd_rootwidget* to be retrieved.

These assumptions, together with a few items of basic class restrictions, are encapsulated in the class *xd_base_c*:

```
class xd_base_c
{
public:
    xd_base_c() {_xd_rootwidget=NULL;}
    Widget xd_rootwidget() const {return _xd_rootwidget;}

protected:
    Widget _xd_rootwidget;
private:
    void operator=(xd_base_c&); // No assignment
    xd_base_c(xd_base_c&);      // No default copy
};
```

SPARCworks/Visual places no other constraints on the base classes used. In other words, any set of base classes can be used provided that they are derived from *xd_base_c* (or another base class that satisfies SPARCworks/Visual's assumptions).

Note that actual parameters for the base class constructor can be supplied with the class name. If parameters are supplied (if the base class string contains a '('), the class is forced to have a constructor and the parameter string is passed to the base class. For example, setting the “Base class” string to mymenubar_c (“Hello World”) for the widget menubar will cause SPARCworks/Visual to generate:

```
class menubar_c : public mymenubar_c {
public:
    menubar_c();
    ...
};
...
menubar_c::menubar_c ()
    :mymenubar ( "Hello World" )
{
}
...
menubar = new menubar_c;
```

10.6 Children Only Place Holders

The *Children Only* structure option lets you designate one widget (the Children Only widget) as a container structure for another structure. Children Only widgets provide context for their descendants in the hierarchy, but no code is generated for them. Consider the following example:

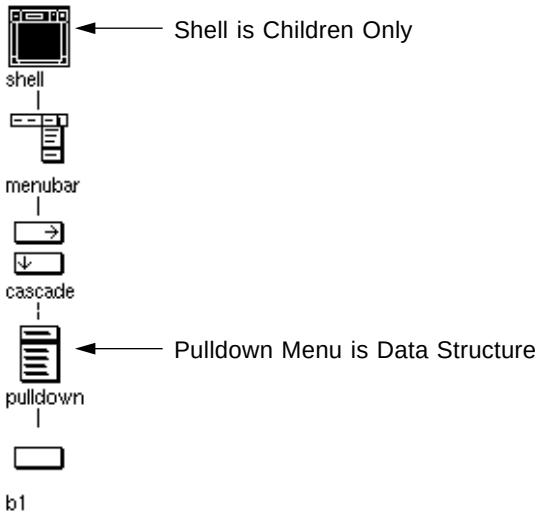


Figure 10-3 Use of Children Only Structure

When you generate code from the design shown in Figure 10-3, SPARCworks/Visual produces code for the pulldown menu structure only. This feature lets you generate fragments of the design that can be controlled by your application program.

10.7 Structure Colors

The “Structure colors” option in the View Menu is useful when you are building a design that uses the structured code generation features. This option color-codes widgets that are designated as function or data structures, C++ classes, or Children Only place holders.

“Structure colors” has a pullright submenu. Select “Show colors” on the submenu to display your structures in the appropriate colors. Click on the dashed line at the top of the submenu to tear it off as a reference to the color code.

Widgets that are not designated as any kind of structure are displayed against the usual background color.

10.8 Structured Code Generation and UIL

When generating UIL for a design that contains structures of some kind, the approach is basically similar for that of C and C++. Independent hierarchies are generated into the UIL file and separate creation functions are generated into the code file. The creation function fetches the appropriate widgets from the UIL hierarchy and fills in the data structure fields as appropriate.

10.9 Changing Declaration Scope

Widgets are normally declared locally in the enclosing creation function unless they are structured in some way, or named. In this case they are declared in the enclosing structure if there is one, or as global variables. This default behavior can be modified by setting the storage class of a widget in the Core resource panel. Setting the storage class to Local forces a widget that would otherwise be declared globally or within a structure to be local to the creation function. Setting the storage class to Global forces an unnamed widget or a named element of a structure to be global. Global status is especially useful for widget-type resources and links as discussed in the *Unreachable Widgets* subsection below. The Static option is similar to Global but the declaration is static to the module.

There is no way to force an unnamed widget into a data structure. Unnamed children of a data structure widget are created and managed locally to the data structure's creation procedure.

10.9.1 Unreachable Widgets

When you use the structured code generation in conjunction with widget-type resources such as *XmNdefaultButton* for a BulletinBoard, or with links, you can specify designs that reference widgets that are not in scope. These are considered unreachable widgets. SPARCworks/Visual attempts to detect these cases and warns you at code generation time. However, unreachable widgets cannot always be detected when used with links. Also, if you use unreachable widgets in conjunction with Children Only structures or dynamic run time creation of hierarchies, unexpected failures may result.

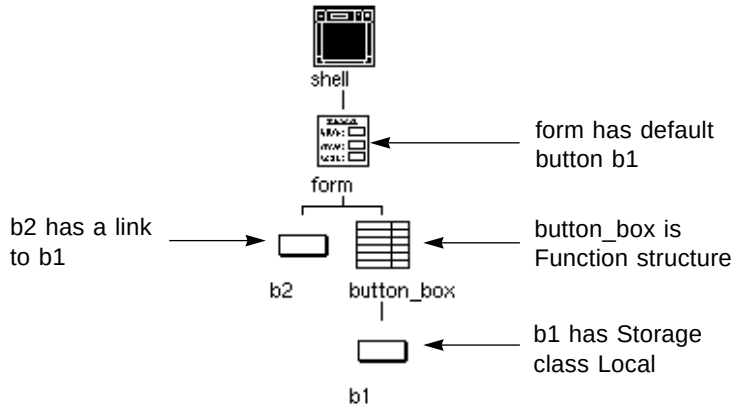


Figure 10-4 Hierarchy with Unreachable Widgets

An unreachable widget is illustrated in Figure 10-4. *b1* must be available to the Form's creation function so that it can be used as the default button argument and as the link destination. However, since *b1* is local to the *button_box* function, it is not in scope in the Form's creation function. SPARCworks/Visual detects this situation and displays the following warning at code generation time.

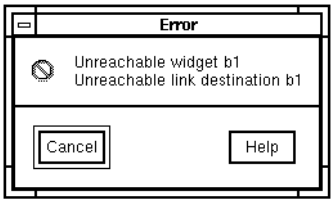


Figure 10-5 Unreachable Widget Error

Code is still generated but it may not compile or run as expected. The simplest solution to this is to force the appropriate widget to be global by using the Storage Class option.

10.10 Definitions

Once a hierarchy of widgets has been encapsulated as a structure (either a C++ class or a C structure), you can re-use it in other designs by turning it into a *definition*. A definition is a reusable hierarchy of widgets which is added to the SPARCworks/Visual widget palette. Selecting a definition from the palette creates an instance of the definition in the design. This instance can be further modified and in turn be made into a definition.

10.10.1 Prerequisites

A widget hierarchy can become a definition provided that:

1. The root widget has a variable name.
2. The root widget has been designated as a C++ class or a structure.
3. The root widget is not part of another definition.
4. The widget hierarchy does not contain a definition.
5. The widget hierarchy does not contain any global or static widgets.

10.10.2 Designating a Definition

Designating a definition requires that the design file is saved with the widget marked in it as being a definition. To mark the widget as a definition use the Definition toggle in the Widget menu. Creating a definition freezes the widgets within it. Their resource panels are disabled and you cannot add widgets or change widget names. You can edit the widgets that make up a definition only by temporarily removing the definition. This should be done with caution to avoid conflicts with designs that use the definition. For details, see the *Modifying a Definition* section on page 217.

To make the definition available for use in other designs SPARCworks/Visual needs an external reference to it. This is provided by means of a definitions file which is edited using the Edit Definitions dialog.

10.10.3 Definition Shortcut

The “Define” button in the Palette Menu is a quick way of adding a new definition. It designates the currently selected widget as a definition, saves the design and adds the definition to the palette. The header filename for the definition is taken from the type declarations filename in the code generation dialog. No icon is used.

10.11 The Definitions file

The definitions file is read by SPARCworks/Visual to establish the set of definitions which are to appear on the palette. The definitions filename is specified setting the *definitionsFileName* resource. The default value is *$HOME/.xddefinitionsrc.*

If you need to work on multiple projects, each of which uses a different set of definitions, you can change the definitions file by setting the resource. For example:

```
visu.definitionsFileName:/home/project6/xddefs
```

The value of this resource can include environment variables:

```
visu.definitionsFileName:$PROJECT_ROOT/xddefs
```

To change to the new setting, exit and restart SPARCworks/Visual.

10.11.1 Editing the definitions file

To modify the definitions file use the Edit Definitions button in the Palette menu.

This displays the dialog shown in Figure 10-6.

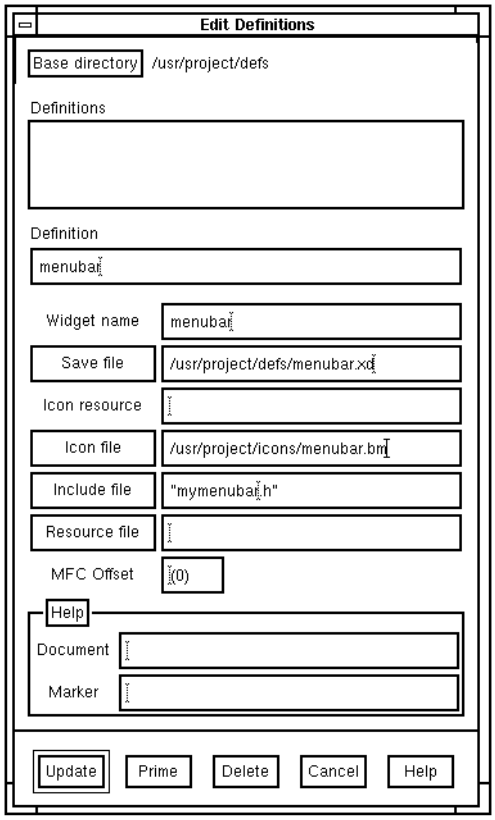


Figure 10-6 Adding a Definition to the Palette.

You can use this dialog to add a new definition, delete a definition, or edit an existing definition. To add a definition, you must supply:

- *Definition* – A definition name
- *Widget name* – The variable name of the root widget of the definition
- *Save file* – The name of a saved design file (*.xd*)

You can also specify:

- *Icon resource* – A resource name which will be used to locate the pixmap file for the definition. See the *Specifying the Icon File* section of the *Configuration* chapter on page 390 for further details

- *Icon file* – A file containing a bitmap or xpm pixmap to be used as the palette icon if one is not found using the Icon resource
- *Include file* – The name of the header file that declares the corresponding structure or class. This file is included in generated code when instances of the definition are used. It should correspond to the name specified when the externs file was generated for the definition
- *Resource file* – The name of the resource file which contains values for the definition. It is included in the generated resource file when instances of the definition are used. It should correspond to the name specified when the resource file was generated for the definition
- *Help information* – A document and tag pair which can be used to provide help to users. See the *Online Help for Definitions* section on page 219 for more details.
- *MFC Offset* – This field is only present when SPARCworks/Visual is in Windows mode. In Windows applications controls are given a unique number by which they are identified. SPARCworks/Visual attempts to generate unique numbers, and in most circumstances there will not be a problem. However when adding widgets to an instance which has a very large number of controls already, it is possible for the numbers to overlap. The MFC offset is added to the id of a control which is being added to an instance. By increasing this number you can make sure that the control's id does not clash with any of the controls in the definition

Attributes not set at creation time can be set later. For example, you can test and debug a definition before designing its icon.

You can use the "Prime" button to fill in several of the fields for the currently selected widget.

10.11.2 Base Directory

If a definition is specified with a relative file name (a name that does not start with /), SPARCworks/Visual adds the *base directory* to the front of the file name. If a base directory is not specified, the directory that contains the definitions file is used.

To specify a base directory, display the Edit Definitions dialog, click on “Base Directory”, select a new directory and click on “Apply”. The new base directory is saved in your definitions file and is immediately used in the current session of SPARCworks/Visual. The base directory cannot be changed if the current design contains instances.

10.12 Modifying a Definition

Widgets in the definition are frozen. You cannot add or delete widgets, rename them, or reset resources. To modify a definition, you must temporarily undefine it. When you need to modify a definition, use the following steps:

1. **Open the save file that contains the definition.**
2. **Select the root widget of the definition.**
3. **Pull down the Widget Menu and turn off the “Definition” toggle.**

Turning off the toggle unfreezes the widgets in the definition so you can make any necessary changes. After making your edits:

4. **Select the root widget and set the “Definition” toggle on again.**
5. **Save the design.**
6. **Regenerate the code file and externs file.**

10.12.1 Impact of Modifying a Definition

Changing a definition affects every design file that uses it. Each time you open a design that uses a definition, SPARCworks/Visual also opens the file that contains the definition and merges information from the two files. If the definition has been modified, SPARCworks/Visual tries to reconcile the new definition with the design that used the old definition.

If any changes cannot be reconciled, SPARCworks/Visual displays an error message and saves any irreconcilable parts of the design in temporary SPARCworks/Visual clipboard files. At this stage there are several ways to proceed:

- Paste the clipboard file into your design and manually resolve its contents with the new definition
- Discard the clipboard file contents altogether

- Exit from SPARCworks/Visual without saving and modify or revert the definition so that it is compatible with the designs that use it

To minimize the risk of incompatibilities:

- Avoid changing the names of widgets in the definition
- Replace a widget in the definition only with a subclass widget of the same name. For example, replacing a Label “foo” with a PushButton “foo” is normally safe

10.13 Instances

To create an instance of a definition simply click on the appropriate button in the palette. The instance is shown with a colored background.

10.13.1 Modifying and Extending an Instance

Creating an instance of a definition corresponds to creating an instance of the structure (either a C structure or a C++ class). You can modify an instance after you have created it provided that the modifications can be reflected in the generated code. For example, you can set resources on widgets or add children to widgets only if they are accessible (i.e. if they are named and, for C++, they have an appropriate access mode). You cannot remove widgets or change their names. The root widget is an exception. Because the root widget of the instance is always accessible (through the member function *xd_rootwidget()*), it can always be modified.

10.13.2 Creating a Derived Structure

It is frequently useful to create a new structure that is derived from the definition. To do this simply set the Structure option on the Code generation page of the Core resources dialog. The derived structure can only be set to the same value as the definition, i.e. it is not possible to derive a C++ class from a C structure.

10.13.3 Overriding a Callback Method

Inherited methods can be overridden so that the derived class has different behavior from the base class.

10.14 Definitions and Resource Files

Resource values for widgets that are components of definitions can be either hard-coded or specified in resource files.

10.14.1 Instances and Definition Resource Files

When you specify a resource file for a definition, SPARCworks/Visual includes that file in the resource file for any design that contains an instance of the definition. The Xlib mechanisms that read the resource file interpret this directive and use it to find the resource file for the definition.

10.15 Online Help for Definitions

To record information about a definition and communicate with other developers who are using it, you can provide online help for definitions. The online help is accessed in the SPARCworks/Visual interface by using the <*Tab*> and arrow keys to get to the icon or button for the definition, then pressing the <*osfHelp*> key (usually <*F1*>).

Help files are stored in subdirectories of the SPARCworks/Visual help directory. The help directory is determined by the *helpDir* resource. By default, it is $VISUROOT*/help where $*VISUROOT *is the path to the* SPARCworks/Visual *installation root directory.*

10.15.1 Text Help Documents

Text help documents are ordinary text files. The first line of the file is used as the title for the help and the remainder as the help body.

The name of the file is formed by concatenating the document name and marker name. These are joined using the value of the visu.*userHelpCatString* resource. By default this resource is set to ".". SPARCworks/Visual looks for this file in the *UserDocs* subdirectory of the SPARCworks/Visual help directory.

10

C++ Code Tutorial 11

11.1 Introduction

This chapter describes how to use SPARCworks/Visual's C++ code generation facilities to add structure to application code and to create reusable widget hierarchies that correspond to C++ classes. These reusable hierarchies, known as *definitions*, appear on the widget palette and can be added to the hierarchy like any other widget. Although this chapter primarily covers C++, most of the material covered is also relevant to structured code generation in C with the exception of the sections on callback methods. Where they diverge the differences are noted.

This chapter is a tutorial. It contains step-by-step instructions that show you how to:

- Create a C++ class corresponding to a widget hierarchy
- Use class methods to handle callbacks
- Use derived classes and preludes to add extra members to the generated class
- Modify or replace the base classes from which the SPARCworks/Visual classes are derived
- Turn a class into a reusable definition and place the definition on the widget palette
- Modify a definition
- Create and modify an instance of the definition
- Use a derived class to extend an instance of a definition

- Override callback methods
- Generate and use resource files for definitions
- Create online help for definitions

For best results, read this chapter at your compiler while running SPARCworks/Visual and do the steps as you read. Several times during the course of the tutorial, you are asked to exit SPARCworks/Visual and restart in a different directory. These instructions ensure that code is generated to the directories expected by the provided Makefiles.

Further information on the subject of structured code generation can be found in the *Structured Code Generation* chapter.

11.2 Creating a C++ Class

A C++ class in SPARCworks/Visual corresponds to any widget with its children. When you designate a widget as a C++ class, SPARCworks/Visual generates a class with that widget and its named descendant widgets as data members. This class can be extended by adding data members and member functions and thus provides a single location for properties that relate to the whole hierarchy.

11.2.1 Designating a C++ Class

Use the following steps to create a widget hierarchy containing a MenuBar widget, then designate the MenuBar as a C++ class. Note that this example would *not* be compatible with Windows code generation.

1. **Create a new directory, *libmenu*. Make this your current directory and start SPARCworks/Visual.**
2. **Build the widget hierarchy shown in Figure 11-1.**

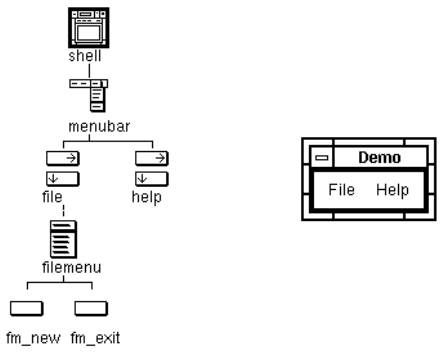


Figure 11-1 MenuBar widget Hierarchy

3. **Set the variable name on the MenuBar widget to *menubar* and set the other variable names as shown in Figure 11-1.**

Only explicitly named widgets are created as members of the class; unnamed widgets are local to the function that creates them. Naming the widgets makes them directly accessible from member functions of the class. Since they are protected members by default, they are also accessible from member functions of any derived class.

4. **Set the "Label" resource for each CascadeButton and PushButton to an appropriate string: "File", "Help", "New" and "Exit".**

5. **Use the Shell resource panel to designate the Shell widget as an Application Shell.**

This completes the example hierarchy. Now designate the menu bar as a C++ class:

6. **Select the MenuBar widget in the widget hierarchy.**

7. **Display the "Code generation" page of the Core resource panel.**

8. **Select "C++ class" from the "Structure" option menu, as shown in Figure 11-2.**

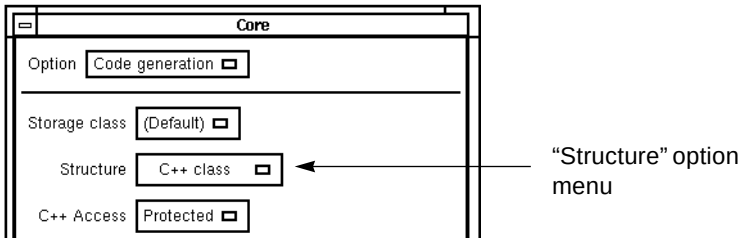


Figure 11-2 Designating a Widget as a Class

9. Click on "Apply".

By default, SPARCworks/Visual uses the variable name of the widget as the basis for a default class name and "Instantiate as" name and so the widget named *menubar* produces the class named *menubar_c*. The base class for *menubar_c* depends on the widget class; here it is *xd_XmMenuBar_c*. These names can be changed, as described later in this chapter.

11.2.2 Widget Member Access Control

The generated *menubar_c* class will contain the class widget (*menubar*) and all its named descendent widgets as members. Although, by default, they are protected members, you can change the access control on any widget by using the Core resource panel. Use the following steps to make the Help menu a public member of the class.

1. Select the CascadeButton named *help* in the widget hierarchy.

2. Display the "Code generation" page of the Core resource panel.

This is shown in Figure 11-3.

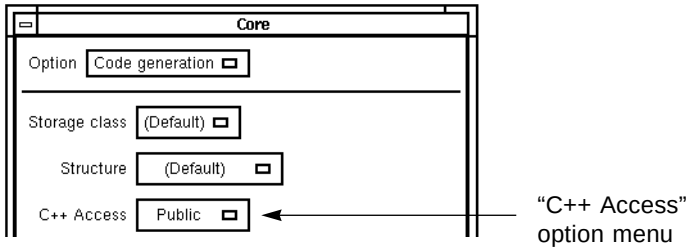


Figure 11-3 Member Access Control

3. **Select "Public" from the "C++ Access" option menu, then click on "Apply".**

11.2.3 C++ Class Code Generation

The code generated for the class consists of a class declaration in the generated Externs file and an implementation in the main C++ code file. To generate these files for the example:

1. **Display the Generate dialog for C++.**
2. **Set the "Type declarations" toggle and specify *menubar.h* as the declaration file. Set all other switches as shown in Figure 11-4.**

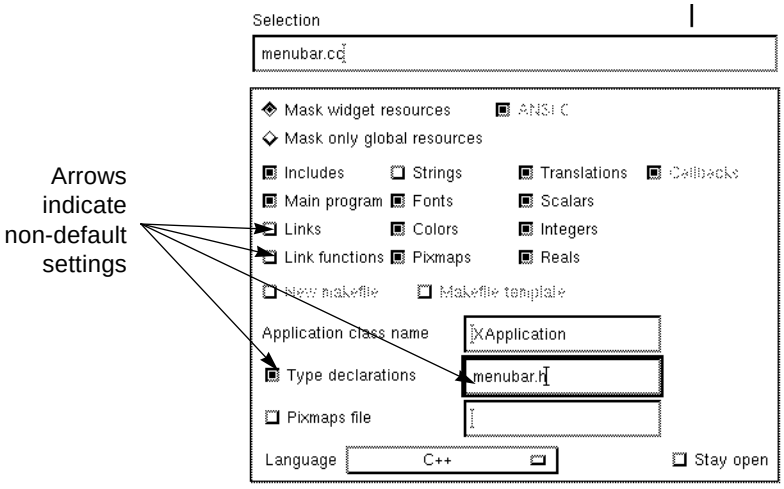


Figure 11-4 Code Generation for the *menubar_c* class

3. Generate C++ for the design into *menubar.cc*.

4. Generate C++ externs for the design into *menubar.h*.

The C++ externs file, *menubar.h*, contains the declaration for the class:

```
...
class menubar_c: public xd_XmMenuBar_c {
public:
    virtual void create (Widget parent, char *widget_name = NULL);
    Widget help;
protected:
    Widget menubar;
    Widget file;
    Widget filemenu;
    Widget fm_new;
    Widget fm_exit;
};

typedef menubar_c *menubar_p;
```

```
...
```

The new class for this MenuBar is based on an existing class, *xd_XmMenuBar_c*. The MenuBar and its named widget descendants are protected members, except for the widget *help*, which you designated as public.

The main C++ file, *menubar.c*, contains the creation function for the new class. This function creates the MenuBar widget and its descendants. Note that this is *not* done in the constructor for *menubar_c*. This gives you the option of creating the widgets later than the class instantiation.

```
...
#include <menubar.h>
...
void menubar_c::create (Widget parent, char *widget_name)
{
    Widget children[2];/* Children to manage */
    Arg al[64];/* Arg List */
    register int ac = 0;/* Arg Count */
    XmString xmstrings[16];/* temporary storage for XmStrings */
    if ( !widget_name )
        widget_name = "menubar";
    menubar = XmCreateMenuBar ( parent, widget_name, al, ac );
    _xd_rootwidget = menubar;
    xmstrings[0] = XmStringCreateLtoR("File",
        (XmStringCharSet)XmFONTLIST_DEFAULT_TAG);
    XtSetArg(al[ac], XmNlabelString, xmstrings[0]); ac++;
    file = XmCreateCascadeButton ( menubar, "file", al, ac );
    ac = 0;
...
    children[ac++] = file;
    children[ac++] = help;
    XtManageChildren(children, ac);
    ac = 0;
}
```

The *menubar.c* file also includes a creation function for the complete hierarchy. This function creates any widgets not in the class: in this case, just the Shell. It then creates an instance of the *menubar_c* class and calls *menubar_c::create()* to create the widget members of the class:

```
void create_shell (Display *display, char *app_name, int app_argc,
      char **app_argv)
{
    Widget children[1]; /* Children to manage */
    Arg al[64]; /* Arg List */
    register int ac = 0; /* Arg Count */
    XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
    XtSetArg(al[ac], XmNtitle, "Demo"); ac++;
    XtSetArg(al[ac], XmNargc, app_argc); ac++;
    XtSetArg(al[ac], XmNargv, app_argv); ac++;
    shell = XtAppCreateShell ( app_name, "XApplication",
       applicationShellWidgetClass, display, al, ac );
    ac = 0;
    menubar = new menubar_c;
    menubar->create ( shell, "menubar" );
    XtManageChild ( menubar->xd_rootwidget());
}
```

11.2.4 Compiling the Generated C++ Code

Since you set the “Main program” toggle when you generated code, *menubar.c* also contains a main program and so the application can be built as it stands.

The C++ code generated by SPARCworks/Visual is straightforward to compile. The only special feature is that the base classes from which the generated classes are derived, such as *xd_XmMenuBar_c*, must be available. The $VISUROOT/*xdclass/lib* directory contains source for the default base classes. $VISUROOT/*xdclass/h* contains header files.

If the *libxdclass* library has not yet been built, use the following steps to build it using the supplied Makefile:

1. Go to the *xdclass/lib* directory in your SPARCworks/Visual installation.

2. **Set VISUROOT to the path of the root of your SPARCworks/Visual installation.**

3. **Type**:

```
make
```

When this completes, the *libxdclass* library is ready to use.

4. **Compile the menubar program.**

The header files for the base classes must be in your include path and the *libxdclass* library must be linked in. Typical build commands are:

```
CC -I. -I$VISUROOT/xdclass/h -I/usr/dt/include \
        -I/usr/openwin/include -c menubar.cc
CC -I. -I$VISUROOT/xdclass/h -I/usr/dt/include \
        -I/usr/openwin/include -L/usr/dt/lib \
        -L/usr/openwin/lib -o menubar menubar.o\
        $VISUROOT/xdclass/lib/libxdclass.a\
        -lXm -lXt -lX11
```

where *$*VISUROOT represents the path of your installed SPARCworks/Visual.

5. **To run the application, type**:

```
menubar
```

The application looks and behaves exactly as it would if there were no classes in it.

11.3 Callback Methods

So far, this example has showed how to designate a widget hierarchy as a class and the form of the code that is generated. At this stage, it does not exploit the fact that the widget hierarchy is a class.

Note that this section, and the following three sections, are specific to C++ programming, when using C structures the conventional callback mechanism applies. If you are not doing C++ programming you might like to skip to the *Creating a Definition* section on page 241.

11.3.1 Callbacks and Member Functions

SPARCworks/Visual provides a simple mechanism that allows you to specify class member functions as callback functions. In SPARCworks/Visual these are known as callback *methods*. The technique used is discussed fully in the *Callback Methods* section of the *Structured Code Generation and Reusable Definitions* chapter on page 205.

11.3.2 Specifying a Callback Method

The callbacks dialog lets you specify the member functions that are invoked in response to events. When you specify a callback method for a particular widget, the method which is invoked is that which belongs to the most immediate class-designated ancestor of the widget (perhaps the widget itself). For example, callback methods on the menu buttons in the MenuBar example invoke member functions of the *menubar_c* class.

Use the following steps to declare a class method on the *fm_new* button:

1. **Display the resource panel for the *fm_new* button.**
2. **Select the "Callbacks" page.**
3. **Click on the "Activate" button to display the callbacks dialog.**
4. **Select the "Methods" toggle.**
5. **In the "Name" field, type**:

```
OnNew
```

6. **Click on "Update".**

This adds *OnNew* to the list of local methods, as shown in Figure 11-5:

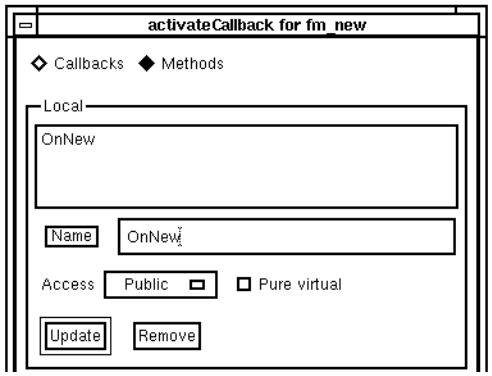


Figure 11-5 Specifying a Callback Method

This designates the member function *menubar_c::OnNew()* as the method that handles the Activate callback of the button *fm_new*. When you do this, SPARCworks/Visual also declares the method on the parent widget *menubar* if you haven't already declared it.

Use a similar procedure to enter a callback method on the *fm_exit* button:

7. **Select the *fm_exit* button in the widget hierarchy.**
8. **Display the resource panel and select the "Callbacks" page.**
9. **Click on the "Activate" button.**

You can now enter the callback method for the *fm_exit* button.

10. **In the "Name" field, type**:

```
OnExit
```

11. **Click on "Update".**

The class now has two callback methods, *menubar_c::OnNew()* and *menubar_c::OnExit()*.

11.3.3 Generating Code for Callback Methods

Using a callback method from within a class causes SPARCworks/Visual to generate declarations for two additional member functions, a complete implementation for one of them and a stub for the other.

1. **Generate the C++ and C++ externs again.**

2. **Generate C++ stubs into *menubarS.cc*.**

Look at the class declaration in *menubar.h*. Two new member functions have been added for each callback method:

```
class menubar_c: public xd_XmMenuBar_c {
public:
...
    static void OnExit( Widget, XtPointer, XtPointer );
    virtual void OnExit( Widget, XtPointer );
    static void OnNew( Widget, XtPointer, XtPointer );
    virtual void OnNew( Widget, XtPointer );
};
```

Note that only the static versions of these functions have the argument list expected by an Xt callback. Therefore, when Xt invokes the callback method *menubar_c.OnNew()*, the C++ compiler selects the static version based on the argument list. Note the following line in the creation function in *menubar.cc*:

```
XtAddCallback (fm_new, XmNactivateCallback, OnNew, (XtPointer)
       this);
```

The code for the static function is also generated into *menubar.cc*. This function simply invokes the non-static virtual member *OnNew(Widget, XtPointer)*, using the instance pointer passed in as client data:

```
void menubar_c::OnNew( Widget widget, XtPointer client_data,
       XtPointer call_data )
{
    menubar_p instance = (menubar_p) client_data;
    instance->OnNew ( widget, call_data );
}
```

You provide the code for the non-static virtual member function *OnNew(Widget, XtPointer)*. A stub for this function is generated to *menubarS.cc*:

```
void
menubar_c::OnNew (Widget w, XtPointer xt_call_data )
{
    XmAnyCallbackStruct *call_data = (XmAnyCallbackStruct *)
       xt_call_data;
}
```

SPARCworks/Visual generates code according to this pattern for all the callback methods that are used in a hierarchy. In this example, similar code is generated for *OnExit()*.

OnNew() and *OnExit()* are invoked from the *fm_new* and *fm_exit* PushButtons but the functions are methods of the *menubar_c* class. This means that all the callback functions that define the behavior of widgets in the class are kept in one place. It also means that all callback functions have access to the instance data for the class and can use it to share information.

11.3.4 Implementing a Callback Method

The application behavior is added by implementing the callback methods. Use the following steps to implement the *OnExit()* method:

1. Open the stubs file, menubarS.c, with a text editor.

2. Complete the implementation of *menubar_c::OnExit()* as:

```
void
menubar_c::OnExit (Widget, XtPointer)
{
    exit(0);
}
```

3. Compile the application.

Use the same build command as before but add *menubarS.c* to the file list. A typical build command is:

```
CC -I. -I$VISUROOT/xdclass/h -I/usr/dt/include \
        -I/usr/openwin/include -c menubar.cc
CC -I. -I$VISUROOT/xdclass/h -I/usr/dt/include \
        -I/usr/openwin/include -c menubarS.cc
```

```
CC -I. -I$VISUROOT/xdclass/h -I/usr/dt/include \
        -I/usr/openwin/include -L/usr/dt/lib \
        -L/usr/openwin/lib -o menubar menubar.o\
        menubarS.o $VISUROOT/xdclass/lib/libxdclass.a\
        -lXm -lXt -lX11
```

where $VISUROOT is set to the path of your installed SPARCworks/Visual.

4. To run the application, type:

```
menubar
```

5. Select the "Exit" button from the File Menu.

Verify that the program exits.

11.3.5 Editing Methods Attributes

Callback methods have two attributes: their access level and whether they are designated pure virtual. The access level determines whether the method is accessible from derived classes and external code. A method can be designated pure virtual to indicate that it has no implementation in the base class, which must be sub-classed to provide an implementation. See the *C++ Classes* section of the *Structured Code Generation and Reusable Definitions* chapter on page 200 for further details.

The attributes are set initially when the callback method is first specified. This can be done on any widget that has a class ancestor. However, they can only be changed on the root widget for the class. For example, the *OnNew()* callback method of the class *menubar_c* can only be edited via the *menubar* widget itself.

1. Select the *menubar* widget in the widget hierarchy.

2. Select "Core resources" in the Widget Menu to display the Core resource panel.

3. Select the "Callbacks" page.

4. Click on the "Declare methods" button to display the callback methods dialog.

This shows a list of the callback methods declared for the class (rather than a list of the methods invoked for any particular event). You can use this panel to edit the callback methods and to declare methods that are not invoked by events in this class but which may be invoked in a derived class.

5. **Select the method *OnNew()* and set the "Pure virtual" toggle as in Figure 11-6.**

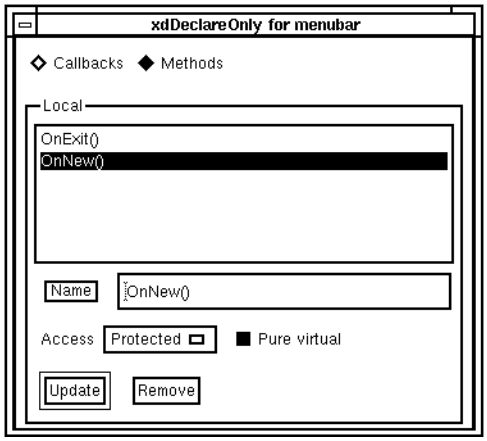


Figure 11-6 Editing a Callback Method

6. **Click on the "Update" button to apply the change.**

7. **Regenerate the C++ externs file, *menubar.h*.**

You do not have to provide an implementation for a pure virtual member function although it is legal to do so. Therefore:

8. **Edit the stubs file, *menubarS.c*, to remove the stub for *OnNew()*.**

9. **Build the menubar program, as before.**

The C++ compiler produces an error message like:

```
"menubar.c" line 111: Error: Cannot create a variable for abstract
      class menubar_c
```

The error occurs because the *menubar_c* class contains a pure virtual function and therefore cannot be instantiated. It is now only useful as a base class. Later in this chapter you will use this class as a basis for a derived class.

11.4 Adding Class Members

This section and the following one present two techniques for adding members to SPARCworks/Visual's generated classes. Note that these sections are specific to C++ programming. The two techniques are:

- Using X-Designer's preludes mechanism
- Generating a derived class

While both techniques require writing code, you can do it without editing the generated code. This means that you can still modify the design in SPARCworks/Visual and regenerate code without losing your additions.

11.4.1 Adding Class Members as a Prelude

The easiest way to add a small number of members is to use the preludes mechanism. This lets you type fragments of code in SPARCworks/Visual and have them passed into the generated code.

1. **Select the *menubar* widget in the widget hierarchy.**
2. **Pull down the Widget Menu and select "Code preludes".**
3. **Select "Protected methods" from the option menu at the top of the dialog.**

This displays a page on which you can enter protected members for this class. You can enter both methods and data members. Note that the Code Preludes dialog also has pages for public and private methods.

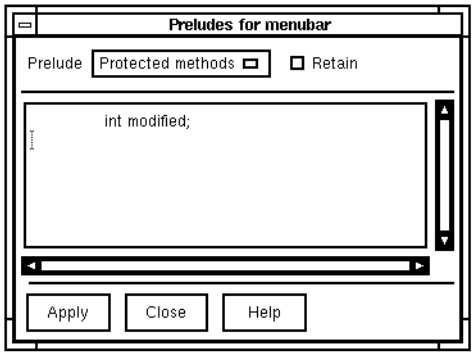


Figure 11-7 Adding a Protected Member to a Class

4. **In the text box, type:**

```
int modified;
```

5. **Click on “Apply” and then “Close”.**

To see the result of this operation:

6. **Generate C++ externs.**
7. **Examine *menubar.h* and verify that the class *menubar_c* now has the additional member.**

11.5 Creating a Derived Class

The Code Preludes dialog is designed for making small insertions to the generated code. To add substantial functionality, it is often better to write a new class derived from the generated class. The logical gap between the two classes can be used to add members and provide implementations for virtual functions. Note that this section is specific to C++ programming.

By default, SPARCworks/Visual derives the name of a C++ class from the variable name of the root widget and so the class for the widget *menubar* is *menubar_c*:

```
class menubar_c: public xd_XmMenuBar_c {
...
};
```

When SPARCworks/Visual generates code to create an instance of the class, it uses the same name:

```
menubar = new menubar_c;
```

You can change the default behavior so that SPARCworks/Visual declares the generated class under one name and creates the instance under another. For example:

```
menubar = new mymenubar_c;
```

To make this change, use the “Instantiate as” field on the Code Generation page of the Core resource panel:

1. **Select the *menubar* widget in the widget hierarchy.**
2. **Display the “Code generation” page of the Core resource panel.**

3. **Set the “Instantiate as” name to *mymenubar_c*, as shown in Figure 11-8.**

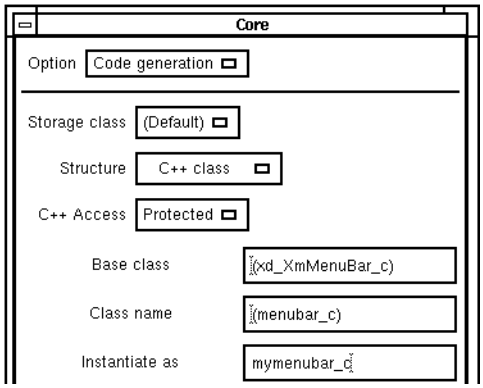


Figure 11-8 Changing “Instantiate as” Name

4. **Click on “Apply” and then “Close”.**

5. **Regenerate C++ and look at the generated code.**

The original class, *menubar_c*, is declared exactly as before. However, when SPARCworks/Visual generates code to create an instance of the class, it uses the “Instantiate as” name:

```
menubar = new mymenubar_c;
```

11.5.1 Writing the Derived Class

SPARCworks/Visual doesn’t generate code for the *mymenubar_c* class. You must provide a header file which declares the class and code to implement any methods it contains. There are no limitations on the new class except that it must be derived from *menubar_c*. For this example use the sample code given below.

1. **In a new file named *mymenubar.h*, write the class declaration for the derived class *mymenubar_c*.**

Use the following code:

```
#ifndef _mymenubar_h
#define _mymenubar_h
```

```
#include <menubar.h>
class mymenubar_c: public menubar_c {
public:
   // Constructor
    mymenubar_c();
  //Provide implementation for inherited pure virtual
    void OnNew(Widget, XtPointer);
};
#endif
```

Because the new class is derived from *menubar_c*, it inherits all widget members and member functions you declared for that class in X-Designer. You can add any number of new members. Here we add a constructor function and an implementation of the *OnNew()* virtual callback method.

2. In a new file named *mymenubar.c* write the class implementation for the derived class *mymenubar_c*.

Use the following code:

```
#include <mymenubar.h>
mymenubar_c::mymenubar_c()
{
    modified = TRUE;
}
void
mymenubar_c::OnNew(Widget, XtPointer)
{
   // Reset modified flag
    if (modified)
        modified = FALSE;
}
```

This completes all the code for the class. Note that the generated C++ code module *menubar.c* needs to include the header file for the derived class *mymenubar_c*. This is done using the “Type declarations” in the “Generate” dialog for C++.

3. **Display the Generate dialog for C++.**

4. **Set the "Type declarations" field to *mymenubar.h* and click on "OK" to generate the C++ code.**

The correct code generation settings are shown in Figure 11-9:

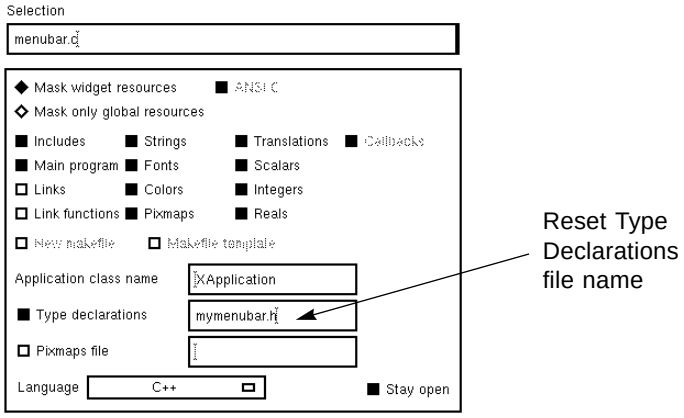


Figure 11-9 Changing the Declarations Header

5. **Regenerate the C++ externs.**

6. **Compile and link all the code modules, including *mymenubar.c*.**

The application now uses the class *mymenubar_c* and invokes its *OnNew()* method when the "New" button in the "File" menu is pressed. To check this, you can extend *mymenubar_c::OnNew()* to print out a message.

You should note that actual parameters for the constructor can be supplied with the class name. For example, setting the "Instantiate as" string to mymenubar_c ("Hello World") will cause SPARCworks/Visual to generate:

```
menubar = new mymenubar_c ( "Hello World" );
```

11.6 Creating a Definition

Once a hierarchy of widgets has been encapsulated as a class, you can re-use it in other designs by turning it into a *definition*. A definition is a reusable group of widgets which can be added to the SPARCworks/Visual widget palette. Selecting a definition from the palette creates an instance of that structure in the design. When SPARCworks/Visual generates code containing an instance, the definition's create function is called to create the widgets.

11.6.1 Prerequisites

A hierarchy of widgets can become a definition provided that:

- The root widget has a variable name
- The root widget has been designated as a C++ class
- The root widget is not part of another definition
- The widget hierarchy does not contain a definition
- The widget hierarchy does not contain any global or static widgets

11.6.2 Designating a Definition

Use the following steps to designate the MenuBar class as a definition:

1. **Select the *menubar* widget in the hierarchy.**

2. **Pull down the Widget Menu and set the "Definition" toggle.**

The *menubar* widget and its descendants are enclosed in a colored box to indicate that they constitute a definition.

3. **Save the design as *menubar.xd*.**

You must save the design containing the definition before adding it to the palette. SPARCworks/Visual uses the saved design file each time the definition is used. Although you can have multiple definitions in a single design file, it is easier to keep track if each file contains only one definition.

Creating a definition freezes the widgets within it. Their resource panels are disabled and you cannot add widgets or change widget names. You can edit the widgets that make up a definition only by temporarily removing the definition. This should be done with caution to avoid conflicts with designs that use the definition. For details, see the *Modifying a Definition* section of the *Structured Code Generation and Reusable Definitions* chapter on page 217.

11.7 Adding a Definition to the Palette

This section explains how to add the new definition to the widget palette.

1. Select “Edit definitions” from the Palette Menu.

This displays the dialog shown in Figure 11-10.

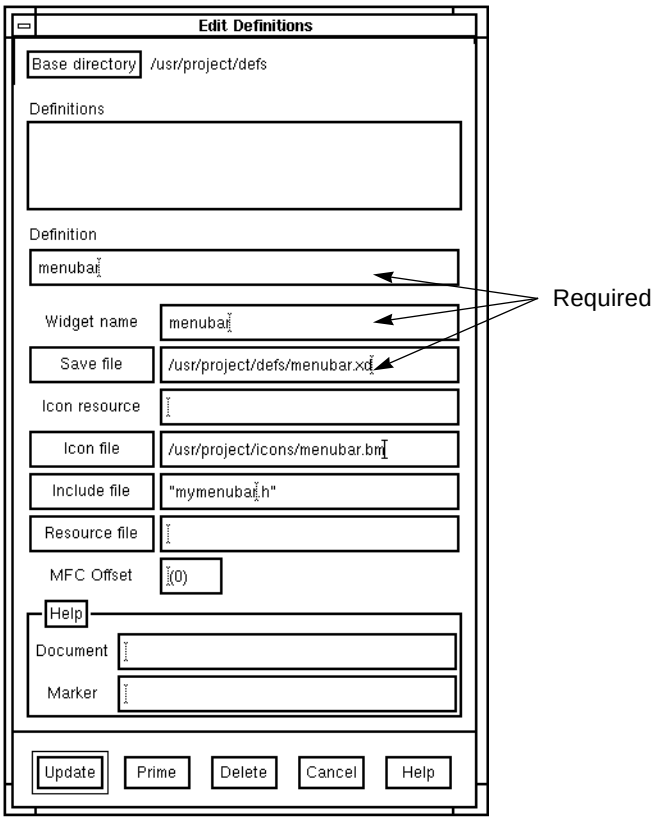


Figure 11-10 Adding a Definition to the Palette

You can use this dialog to add a new definition, delete a definition, or edit an existing definition. To add a definition, you must supply:

- *Definition* – A definition name
- *Widget name* – The name of the root widget of the definition
- *Save file* – The name of a saved design file (*.xd*)

You can also specify:

- *Icon resource* – A resource name which will be used to locate the pixmap file for the definition. See the *Specifying the Icon File* section of the *Configuration* chapter on page 390 for further details

- *Icon file* – A file containing a bitmap or xpm pixmap to be used as the palette icon if one is not found using the Icon resource
- *Include file* – The name of the header file that declares the corresponding structure or class. This file is included in generated code when instances of the definition are used. It should correspond to the name specified when the externs file was generated for the definition
- *Resource file* – The name of the resource file which contains values for the definition. It is included in the generated resource file when instances of the definition are used. It should correspond to the name specified when the resource file was generated for the definition
- *Help information* – A document and tag pair which can be used to provide help to users. See the *Online Help for Definitions* section of the *Structured Code Generation and Reusable Definitions* chapter on page 219

Attributes not set at creation time can be set later. For example, you can test and debug a definition before designing its icon.

2. Enter the parameters for the Edit Definitions dialog as shown in Figure 11-10. Where a pathname is required, specify your own pathname.

You can use the “Prime” button to fill in several of the fields for the currently selected widget. Specify your Externs file (*mymenubar.h*) in the “Include file” field. Note that the “Instantiate as” name you specified on the MenuBar applies each time the definition is used.

Specify an icon file (*menubar.xpm*) as the “Icon file”. The icon is optional. If you do not specify an icon, SPARCworks/Visual uses a PushButton bearing the definition name in the widget palette and the widget icon for the root of the definition in the widget hierarchy. You can use the SPARCworks/Visual pixmap editor to design an icon for your definition. It should use an area of color “none” which is used to show the selection in the widget hierarchy. For details, see the *Configuration* chapter.

Note that you can also specify the icon via the SPARCworks/Visual resource file. To do this, specify the name of a SPARCworks/Visual resource in the “Icon resource” field and set that resource to a filename in the SPARCworks/Visual resource file.

The other fields in the Edit Definition dialog are discussed later in this chapter.

3. Click on “Update”.

The icon you specified appears on the SPARCworks/Visual widget palette. It becomes active whenever you select a widget that can have a MenuBar child.

11.7.1 Definition Shortcut

The “Define” button in the Palette Menu is a quick way of adding a new definition. It designates the currently selected widget as a definition, saves the design and adds the definition to the palette. The header filename for the definition is taken from the type declarations file name in the code generation dialog. No icon is used.

11.8 Configuring Definitions

The set of definitions that you are using is determined by a file known as the definitions file. By default this file is $HOME/.xddefinitionsrc, but its name can be configured using the resource visu.definitionsFileName.

11.9 Generating Code for a Definition

The code for a definition has two parts: the declaration in the public header file (the externs file) and the code module containing the implementation. The code module does not have to be public in order to compile applications containing instances; it can be made available in compiled form in a library.

Use the steps in this section to generate only the code for the definition, i.e. without the Shell or other widgets and without a main program.

To mark the Shell widget so that no code is generated for it:

1. **Select the *shell* widget in the hierarchy.**
2. **Display the Core resource panel and select the “Code generation” page.**
3. **Set the “Structure” option menu to “Children only” and then click on “Apply” followed by “Close”.**

After you do this, SPARCworks/Visual ignores the Shell and any of the Shell’s children that are not designated as C++ classes, functions, or data structures. Code is generated only for the MenuBar and its descendants.

You must also suppress generation of the main program:

4. **Generate C++ code into *menubar.c*, with the "Main program" toggle off.**
5. **Generate C++ externs into *menubar.h*.**
6. **Save the design in *menubar.xd* and exit from SPARCworks/Visual.**

This completes the process needed to create a definition and the code for the corresponding class. It can now be used in an application. The normal way to make the implementation available for reuse is as a library:

7. **Compile the code and create a library *lib.a* in your *libmenu* directory.**

Typical build commands to compile the three source files (*menubar.c*, *menubarS.c* and *mymenubar.c*) are:

```
CC -c -I. -I$VISUROOT/xdclass/h *.c
ar r lib.a *.o
```

where $VISUROOT is set to the path of your installed SPARCworks/Visual.

11.10 Creating an Instance

A definition can be used in the same way as a widget on the palette. Clicking on the palette button creates an instance of the definition. SPARCworks/Visual copies the definition's hierarchy into the tree where it can be modified and extended. In the generated code SPARCworks/Visual will include a call to the definition's creation function to create the instance. Use the following steps to build a new design using the *menubar* definition.

1. **Create a directory *cmd* parallel to the directory *libmenu* and make *cmd* the current directory.**
2. **Start SPARCworks/Visual.**

Note that the *menubar* definition icon appears on the widget palette.

3. **Click on the following palette icons: Shell, MainWindow and the menubar definition.**

This produces the widget hierarchy shown in Figure 11-11. The components of the instance are enclosed in a colored box to indicate that they form a single entity. All widgets except the root widget are given the same names they had in the original definition. The root widget is assigned a default name of the form *widget<n>*. For reliable code, assign it an explicit name.

4. **Name the root widget of the instance, the Shell and the MainWindow as shown in Figure 11-11.**

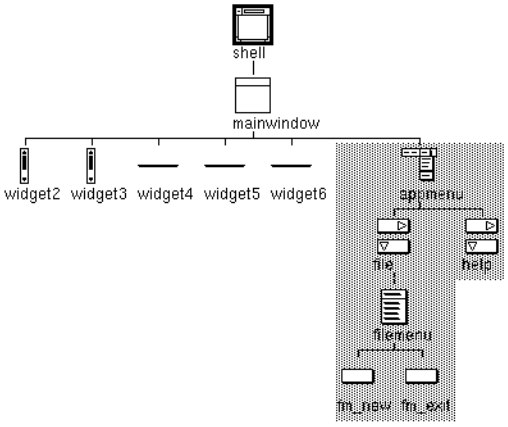


Figure 11-11 Hierarchy Containing an Instance of a Definition

Use the Shell resource panel to designate the Shell widget as an ApplicationShell.

11.11 Modifying and Extending an Instance

You can modify an instance after you have created it provided that the modifications can be done in the generated code. For example, you can set resources on widgets or add children to widgets only if they have public access. You cannot remove widgets or change their names. The root widget is an exception. Because the root widget of the instance can be accessed through the member function *xd_rootwidget()*, it can always be modified.

In our example, all components of the definition are protected except for the *help* button. This means only the *help* button's label can be changed. Similarly, it is possible to add extra widgets under the *help* menu but not under the *file* menu.

1. **Select the *help* widget in the widget hierarchy.**

2. **Click on the Menu icon in the widget palette and then on the PushButton icon.**

This adds a single item menu under the *help* CascadeButton.

3. Set the widget names as shown in Figure 11-12.

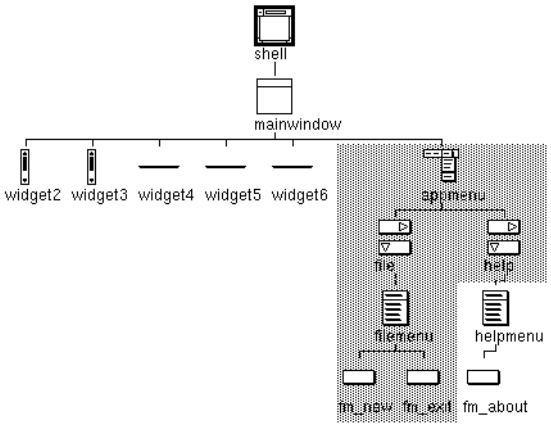


Figure 11-12 Extending an Instance of a Definition

Currently, you cannot modify other widgets in the definition. For example, you cannot add buttons to *filemenu*. However, if you create a subclass from a definition, you can modify protected as well as public widgets. This technique is discussed in the next section.

11.12 Creating a Derived Class

Because most members of the class corresponding to the definition are protected, they cannot be accessed in an instance of the definition. There are two ways to address this:

- modify the original definition to make the members public
- designate the instance as a class (Because the instance class is derived from the definition class, it has access to the protected members)

The second approach maintains better encapsulation and lets you exploit the callback methods.

1. Select the MenuBar widget, *appmenu*, in the widget hierarchy.

2. Display the "Code generation" page of the Core resource panel.

Select “C++ Class” from the “Structure” option menu and then click on “Apply”.

The MenuBar widget is designated as a class, as in Figure 11-13. This class is derived from the class that corresponds to the definition.

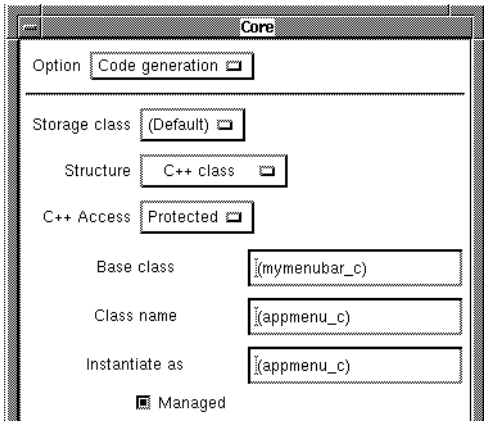


Figure 11-13 Creating a Derived Class from a Definition

Because member functions of the class can access the protected members of *mymenubar_c*, extra widgets can now be added anywhere in the hierarchy.

3. **Select the *filemenu* widget in the widget hierarchy.**
4. **Add two extra PushButtons to the menu and label them “Open...” and “Save...”.**
5. **Set the variable names of the new buttons to *fm_open* and *fm_save*.**
6. **Use mouse button 1 to drag the new buttons into the positions shown in Figure 11-14**

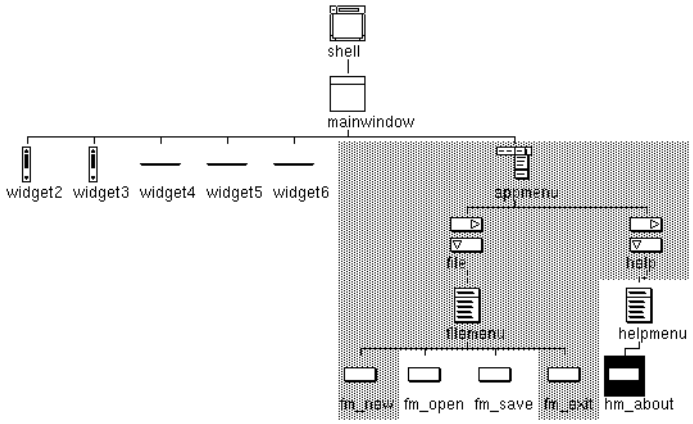


Figure 11-14 Extending a Derived Class

The order of a definition cannot be changed. This means that, while you can move widgets into the definition, you cannot move widgets within it.

7. **Click twice on *fm_open* in the hierarchy to display the resource panel.**
8. **On the "Callbacks" page of the resource panel click on "Activate".**
9. **Select the "Methods" page of the Callbacks dialog and specify the following callback:**

```
OnOpen
```

10. **Click on "Update".**
11. **Repeat steps 8 through 11 to set the Activate callback methods for *fm_save* to *OnSave*.**

This technique is also valid for C structures. SPARCworks/Visual will generate a new structure which is an extension of the definition's structure.

11.13 Overriding a Callback Method

The class *appmenu_c*, which corresponds to the MenuBar, has four callback methods: *OnNew()* and *OnExit()*, which are inherited, and *OnOpen()* and *OnSave()* which are defined by *appmenu_c* itself. However, the inherited methods can be overridden so that the derived class has different behavior from the base class. Note that this section is specific to C++ code.

1. **Select the widget *appmenu* in the widget hierarchy.**
2. **Display the "Callbacks" page of the Core resource panel.**
3. **Click on the "Declare methods" button and select the "Methods" page.**

The "Declare only" callbacks for the class are displayed.

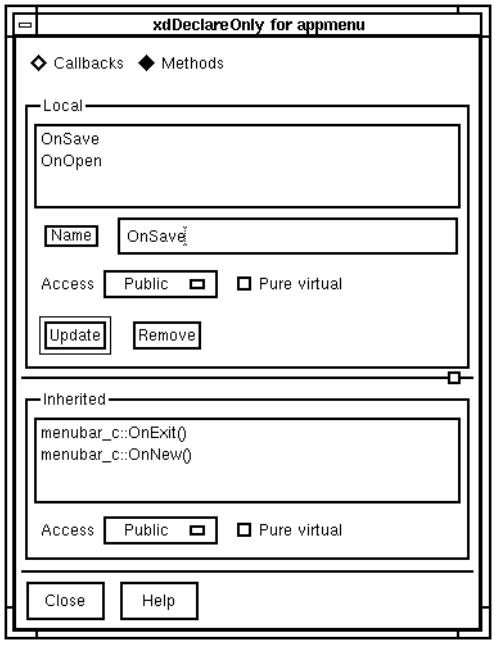


Figure 11-15 Declare Only Callbacks

Note that *OnSave()* and *OnOpen()* are local to *appmenu_c*, whereas *OnExit()* and *OnNew()* are inherited from *menubar_c*.

4. **In the “Name” field, type:**

```
OnExit
```

You can click on the “Name” button to get a browser rather than typing the name “OnExit” yourself.

5. **Click on “Update”.**

OnExit() is added to the list of local methods. It can now be overridden.

11.13.1 Implementing Overridden Methods

Overridden methods are implemented by completing the stubs generated by SPARCworks/Visual:

1. **Generate the code for C++ into *app.c*. Set the “Type declarations” toggle and specify *app.h* as the declaration file. Set all other toggles as shown in Figure 11-16.**

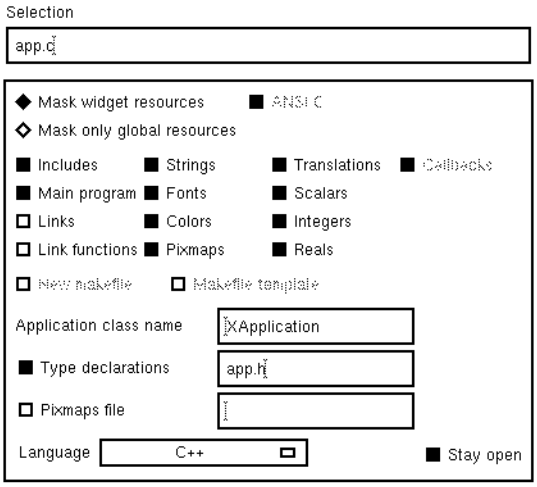


Figure 11-16 Code Generation for the Application

2. **Generate C++ externs into *app.h*.**

3. **Generate C++ stubs into *appS.c*.**

4. Complete the stubs file, *appS.c*, as follows:

```
...
#include <iostream.h>
void
appmenu_c::OnExit (Widget w, XtPointer xt_call_data)
{
    if (modified)
        XBell(XtDisplay(w), 100);
    else
        exit(0);
}
void
appmenu_c::OnSave (Widget, XtPointer xt_call_data)
{
    cout << "Saving..." << endl;
    modified = FALSE;
}

void
appmenu_c::OnOpen (Widget, XtPointer xt_call_data)
{
    cout << "Opening..." << endl;
    modified = TRUE;
}
```

This implementation of *OnExit()* overrides the implementation in the definition. The function *XBell()* rings the bell on the X server. *OnSave()* and *OnOpen()* are stub functions that print appropriate messages and update the *modified* flag.

The functionality of the application's menubar is now:

- The *modified* flag is initially set *TRUE* in the constructor for the class. It is set *TRUE* by "Open" and set *FALSE* by "New" and "Save"
- "Exit" terminates the application unless the *modified* flag is set, in which case it rings the bell

- "Open" and "Save" produce informative messages on stdout

5. **Compile and link the application code. The library containing the menubar class implementation must be linked in.**

Typical build commands are:

```
CC -c -I. -I../libmenu -I$VISUROOT/xdclass/h app.c
CC -c -I. -I../libmenu -I$VISUROOT/xdclass/h appS.c
CC -o app app.o appS.o ../libmenu/lib.a \
$VISUROOT/xdclass/lib/libxdclass.a \
-lXm -lXt -lX11
```

where $VISUROOT is set to the path of your installed SPARCworks/Visual.

6. **Run the application and verify that the menu behaves as expected.**

7. **Save the application as app.xd and exit from SPARCworks/Visual.**

11.14 Definitions and Resource Files

Resource values for widgets that are components of definitions can be either hard-coded or specified in resource files. When you specify a resource file for a definition, SPARCworks/Visual includes that file in the resource file for any design containing an instance of the definition.

1. **So far in this example, all resources have been hard-coded. Use the following steps to regenerate the *menubar* definition with string resources in a resource file:**

2. **Change to the *libmenu* directory and start SPARCworks/Visual.**

3. **Open the design file *menubar.xd*.**

4. **Display the "Generate C++" dialog and set the "Strings" toggle off.**

5. **Generate the C++ code.**

This removes hard-coded string resources such as the labels on buttons. Now generate a resource file containing the string resources:

6. **Display the "Generate X resource file" dialog and generate the resources into *menubar.res*.**

7. **Save the design.**

8. **Compile the code for the library as before.**

11.14.1 Editing the Definition

In the preceding steps, you changed the generated code for the definition so that string resources are kept in a resource file. Now edit the *menubar* definition and specify a resource file:

1. **Select “Edit definitions” from the Palette Menu.**

2. **On the Edit Definitions dialog, select *menubar* from the scrolled list of definitions.**

3. **In the “Resource file” field, type:**

```
../libmenu/menubar.res
```

This is shown in Figure 11-17.

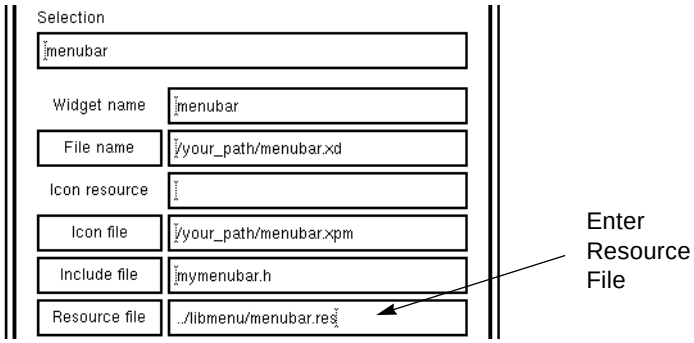


Figure 11-17 Setting the Resource File on the Edit Definitions Dialog

4. **Click on “Update”.**

This step saves the change to the definition in your definitions file (*$HOME/.xddefinitionsrc*). Note that you can use the Edit Definition dialog to specify a resource file at any time. The original *menubar.xd* design does not have to be loaded to perform this step.

5. **Exit X-Designer.**

11.14.2 Instances and Definition Resource Files

When you specify a resource file for a definition, SPARCworks/Visual includes that file in the resource file for any design that contains an instance of the definition. The Xlib mechanisms that read the resource file interpret this directive and use it to find the resource file for the definition. Use the following steps to try this in the *app.xd* application.

1. **Change to the *cmd* directory and start SPARCworks/Visual.**
2. **Open the design file *app.xd*.**
3. **Generate the X resource file into *app.res*.**

The X resource file for the application contains the following directive:

```
! Generated by SPARCworks/Visual
#include "../libmenu/menubar.res"
```

Xlib interprets this *#include* directive as giving a pathname relative to the directory containing the application resource file. For details, see the Xlib documentation.

Cross Platform Development 12

12.1 Introduction

This chapter describes the cross-platform capabilities of SPARCworks/Visual which are enabled in Windows mode. It contains sections on:

- Starting SPARCworks/Visual in Windows mode
- Specific controls
- Configuration resources
- Reading old designs
- Windows compliance

This chapter should be read in conjunction with the following tutorial chapter and the Structured Code Generation chapter and tutorial which precede it.

Note – This chapter is only relevant if you have purchased the SPARCworks/Visual XP add-on package. This package allows you to create interfaces which are common to both Motif and Windows environments.

12

12.2 Overview

The best way to develop an application which is to be ported to different platforms is to encapsulate the platform specific parts in some way so that the body of the application is isolated from the implementation details. SPARCworks/Visual uses its C++ code structuring capabilities to generate a set of classes for the user interface that have the same public interface, but two (totally) different implementations. In SPARCworks/Visual these implementations are known as *flavors*. There are three flavors of C++ code that can be generated:

1. *Motif* – The vanilla flavor is the same as generating C++ code when not using the cross platform capabilities. The base classes are a very simple set provided, with source, with the SPARCworks/Visual release.

2. *Windows MFC* – The target base classes on the Windows platform are the Microsoft Foundation Classes. These provide a fairly high level set of controls and functions which can be used to build user interfaces.

3. *MotifMFC* – A set of base classes supplied with source as part of the SPARCworks/Visual release. They are very similar to the Motif flavor base classes except that they are named to match the Microsoft Foundation Classes and provide a little of the basic functionality. They are not intended to provide the whole of the MFC interface, only enough to allow the developer some measure of shared code in the user interface. The real goal is in sharing code in the rest of the application.

The full capabilities of SPARCworks/Visual's C++ model can still be used for cross platform applications. This means that you can use sub-classing and inheritance to add additional functionality. These techniques can, for example, be used to support cross platform versions of your user-defined widgets.

12.2.1 Windows Mode and Compliance

Unfortunately for the user interface developer, the Motif and X toolkits bear little resemblance to the Microsoft Windows toolkit. Although the visual appearance is similar, the actual use of the toolkit is very different. This requires SPARCworks/Visual to impose some restrictions on the developer before Windows code can be generated for a design. SPARCworks/Visual will impose these restrictions when it is in *Windows mode*. When

SPARCworks/Visual is in Windows mode it needs to permit the developer to work on designs which do not comply with these restrictions (so that an existing design can be read in for example). As a result SPARCworks/Visual will check that a design is *Windows compliant*. If a design is not Windows compliant then C++ code can only be generated for the Motif flavor.

The issues of compliance and importing old designs are covered in later sections of this chapter.

12.2.2 Widgets and Resources

In addition to the compliance issues which actually stop a design from being generated for Windows, there are many attributes of a design which simply have no effect in the Windows implementation (for example the alignment of a label on a button), because the Windows toolkit does not support the concept. SPARCworks/Visual indicates this by means of a color cue. In the resource panels the text input fields, settings option menus and callback buttons use a pink (by default) color to indicate that although setting this resource will have an effect in the Motif flavors, it will have no effect in the Windows flavor.[1]

Similarly the variable name field can be pink to indicate that the widget will not map to any equivalent Windows object. For instance, in Windows a menubar is simply an attribute of a Dialog, there is no menubar object. Consequently SPARCworks/Visual will show the variable name pink for menubar widgets.

SPARCworks/Visual will also use this technique to indicate that some links will not be reproduced in the Windows code.

All these features are covered in more detail later in this chapter and in the tutorial following it.

12.3 Starting in Windows Mode

There are three methods of invoking SPARCworks/Visual so that it is running in Windows mode. Select the method which suits you best, bearing in mind that you may have to share your copy of SPARCworks/Visual with other users who do not want to run SPARCworks/Visual in Windows mode.

1. Pinking is only discernible on a color display.

12.3.1 The Resource File

Windows mode can be specified by a resource in the file containing your SPARCworks/Visual application resources:

```
visu.windows:true
```

12.3.2 The Command Line Switch

Windows mode can also be specified by a command line switch when invoking SPARCworks/Visual:

```
visu -windows
```

12.3.3 Separate Version of SPARCworks/Visual

The third method of invoking SPARCworks/Visual in Windows mode gives the appearance of a separate application which is always in Windows mode. This method uses the application resource, described above.

1. **Create a hard link to your SPARCworks/Visual shell script in the $VISUROOT/lib directory.**
2. **Give this file a name such as visu*windows*.**
3. **Create a hard link to your SPARCworks/Visual binary in the $VISUROOT/bin directory. Use the same file name as in Step 2.**
4. **Add the windows flag to your application resource file, using the name of your symbolic link, as follows**:

   ```
   visuwindows.windows:true
   ```

In this way, you can simply type visu*windows*. This will invoke SPARCworks/Visual in Windows mode. Other people can then continue to use SPARCworks/Visual safe in the knowledge that the default will not be Windows mode.

This is the technique that SPARCworks/Visual uses to bring up the version for smaller screens - such as the VGA screen. The program small_visu is nothing more than a symbolic link to the SPARCworks/Visual binary which picks up an alternative set of resources from the SPARCworks/Visual resource file.

12.4 The SPARCworks/Visual Window

The SPARCworks/Visual Window looks slightly different when in Windows mode. The main visual differences are the addition of two items in the toolbar at the top (and in corresponding menus) and the fact that some of the widgets are not included in the widget palette. This second point is dealt with in the *Windows Compliance* section on page 262. The first set of differences is described here.

12.4.1 Windows Compliant Toggles

There are two Windows Compliant toggles - one on the toolbar and one in the Module menu. They both have the same function.

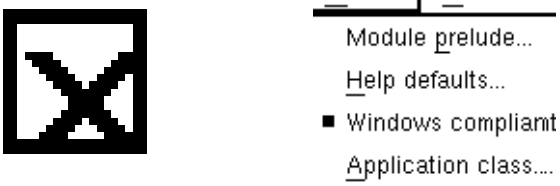


Figure 12-1 The Windows Compliant Buttons in the Toolbar (left) and the Module Menu (right)

These toggles indicate whether or not the current design is Windows compliant. If you read in a design created with a version of SPARCworks/Visual not in W6indows mode or you cut and paste areas of a compliant hierarchy, it is possible to create structures which are not Windows compliant.

In such a case a message is displayed informing you where the problem is and the two Windows Compliant toggles are unset. The toolbar toggle displays a red cross when it is unset. Having made the appropriate changes to your design, pressing either of these toggles will check the compliance again. If your design is now Windows compliant, the toolbar button will be set (the red cross will disappear). If not an error message will appear indicating the problem and the toggles will remain unset.

12.4.2 The Flavor Menu

The flavor menu in the toolbar and in the generate dialog are the same. They specify which flavor of code you wish to generate: plain Motif, MFC Motif or MFC Windows. This only applies to C++ code generation.

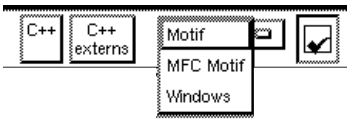


Figure 12-2 The Flavor Menu in the Toolbar

Having used the generate dialog once to specify the names of the files you wish to generate, you can use the toolbar generate buttons and the toolbar flavor button to regenerate files without having to display the generate dialog.

12.5 Windows Compliance

This section details the restrictions imposed by SPARCworks/Visual which ensure that code for the MFC flavors can be generated. When in Windows mode SPARCworks/Visual will not allow you to create a design which violates these restrictions. However, it is possible to read in a non-compliant design, or to make a design non-compliant by means of cut and paste.

12.5.1 Structure restrictions

Because of the differences between Motif and Windows in the way events are handled, some widgets cannot be made classes. In Windows, events concerning certain widgets are always sent to the enclosing class. Other widgets must be classes in order to handle Windows messages. Here is a list of these restrictions:

- Cannot be class
  - MenuBar
  - PopupMenu
  - CascadeButton
  - OptionMenu

- Any widget which is the child of a Shell

- Must be class
  - Shells
  - ScrolledWindow (unless child of Shell)
  - Frame
  - RadioBox, unless the child of a Frame
  - DrawingArea (unless child of MainWindow, ScrolledWindow or Shell)
  - Paned Window
  - Child of Paned Window

  The first error you are likely to encounter on reading in an old design is the fact that the Shell must be structured as a C++ class. This error is easy to fix and can be done automatically from the Compliance Failure dialog. See the *Compliance Failure* section on page 267 for more details.

4. C Structures

   SPARCworks/Visual does not support the Function or Data Structure options in a compliant design.

5. Classes and Callbacks

   Structural errors can be considerably more complicated if you have a design which is well-structured, making good use of C++ classes and with callback methods scattered among the child widgets. The following example demonstrates how this problem may occur and how to overcome it.

6. Example

   When a widget is given a callback method, the method is declared in the enclosing class. In the following example, while using a version of SPARCworks/Visual which was not in Windows mode, the MenuBar, MBar_class, was declared a class and the button given a callback method:

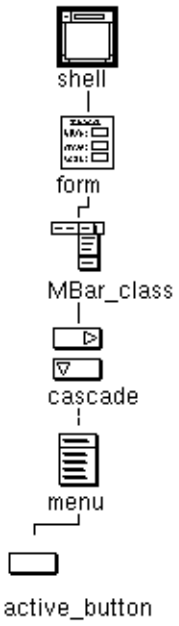


Figure 12-3 Non-Compliant Hierarchy

The method is declared in MBar_class. If you then read the design into a version of SPARCworks/Visual in Windows mode, you will be presented with the following error message because MenuBars cannot be classes:

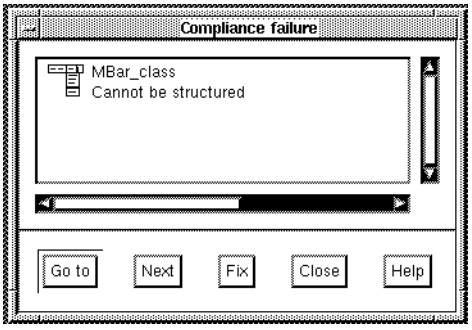


Figure 12-4 Error Message Showing Non-Compliance

If you remove the class definition of the MenuBar (using the Core Resources dialog), you will then see the following error message:

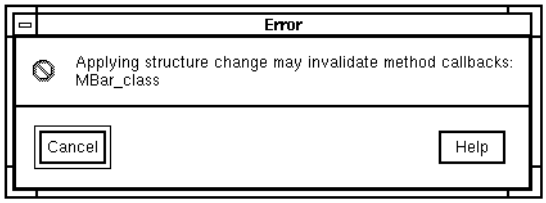


Figure 12-5 Message Informing of Method Callback Declaration Invalidation

Make sure that the method declarations are removed from the MenuBar and, in this case, added to the Shell. You cannot add them to the Form because the Form, like the MenuBar, does not map to an object on Windows.

Although SPARCworks/Visual assists you in the above way, changing whether a widget is a class or not could have a major impact on your application. You will need to reconsider the structure of your application very carefully.

12.5.2 Menubar restrictions

Menubars in Windows are created by setting an attribute of the Dialog. This leads to two compliance restrictions:

- Only one Menubar per shell supported. You cannot have a design which contains more than one Menubar in a Dialog
- Menubar parent must be child of Shell. A Menubar cannot be an immediate child of a Shell, nor can it be at a deeper level in the widget hierarchy than as a child of the Shell's child

12.5.3 FileSelectionBox

File Selection Box must be a child of a DialogShell or TopLevelShell. In Windows file selection is provided by a pre-defined FileSelection Dialog. This dialog can only contain a single work area child, it cannot support a Menubar, nor additional buttons (not managed by the work area).

12.5.4 Unsupported widgets

The following widgets have no comparable control in Windows and so cannot be used in a design that is to be portable:

- SelectionBox
- Command
- DrawnButton
- ArrowButton
- MessageBox
- SelectionPrompt

12.5.5 Scale

The Scale widget maps to a Windows ScrollBar control which cannot support child controls. Therefore a Scale widget with children violates the Windows compliance restrictions.

12.5.6 Frame and RadioBox

As both Frame and RadioBox (if not the child of a Frame) have to be classes, because of the way messages are passed to an enclosing control, and because the child of a Shell cannot be a class, because this would lead to the Dialog not handling any messages, Frame and RadioBox cannot be the immediate child of a Shell.

The title child of a Frame must be a Label widget. The Frame control in Windows (actually a CButton in disguise) simply has a title attribute. There is no way to use another control as the title.

12.5.7 MainWindow and ScrolledWindow

Windows does not support automatic scrolling and hence SPARCworks/Visual in Windows mode disables automatic scrolling options. The MainWindow widget may only include a work area and Menubar child. It does not support the command window or message window.

12.5.8 Paned Windows

A compliant design cannot contain a Paned Window which has Separator, MainWindow, OptionMenu, or MenuBar children. Neither may the children be definitions or instances.

12.5.9 Definitions

The XmNlabelType resource cannot be explicitly set for a widget which is a component of a definition when it is instantiated in another design. I.e. if a Button does not have XmNlabelType set in a definition then when that definition is used XmNlabelType cannot be set to PIXMAP. This is because Windows uses a different class (CBitmapButton instead of CButton) to implement a button with a bitmap on it. It is obviously not possible to change the class of a variable after it has been created, hence the restriction.

For the same reason it is not possible to have a CascadeButton in a definition which has no Pulldown menu and to then add the Pulldown in an instance.

Slightly more subtly, it is not possible to have a widget with methods added to an instance which is not being sub-classed. For example imagine you have an instance of RowColumn definition, and the root widget (i.e. the RowColumn) does not have its structure set to class. When not in Windows mode it is possible to add a button to the RowColumn instance and give it a method callback which is declared at an enclosing scope (say a Shell class). This is not possible in Windows; the event has to be handled by the enclosing control (in this case the CWnd which represents the RowColumn).

12.6 Compliance Failure

When you read in a design which was created by SPARCworks/Visual while not in Windows mode or you use cut and paste in such a way as to cause a design to become non-compliant, the Compliance Failure dialog appears showing you which widgets are causing the design to be non-compliant.

.

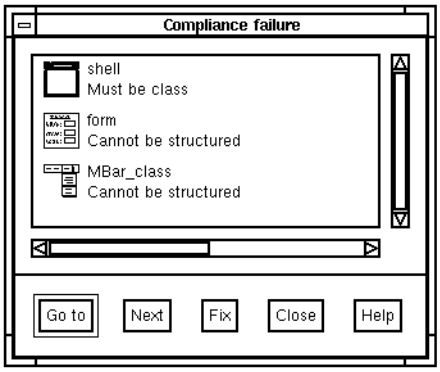


Figure 12-6 The Compliance Failure Dialog

You can select the widgets in the scrolled list of non-compliant widgets. You may then press one of the following buttons:

- *Go to* – This highlights the widget in your design. If the widget is part of a dialog which is not the current one, the relevant dialog is selected. If the widget is in a part of the design which is folded, the design is unfolded.
- *Double-clicking* on a widget is the same as pressing *Go to*. For structure problems, SPARCworks/Visual will also open the Core resources dialog on the Code Generation page.
- *Next* – This moves to the next widget in the list and selects the widget in the hierarchy.
- *Fix* – This fixes the problem, where possible. SPARCworks/Visual can only fix those errors connected with the structure of a widget. Other errors cannot be fixed automatically. See the *Windows Compliance* section on page 262 for a list of the possible compliance errors.

12.6.1 Example

This example shows how compliance failure is detected. In the hierarchy shown in Figure 12-7, the RowColumn widget rc_is_class has been made a class:

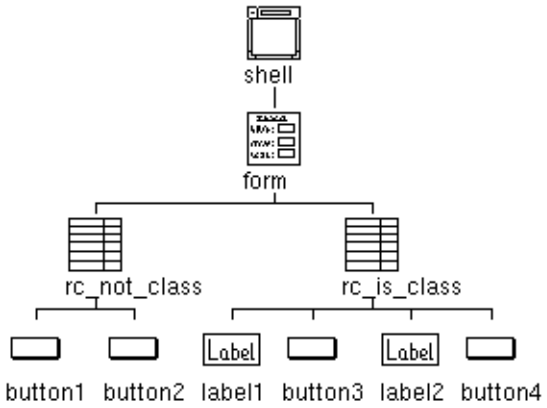


Figure 12-7 RowColumn Defined as Class

If you were to cut rc_is_class, clear form and then paste (i.e. pasting rc_is_class as a child of shell), SPARCworks/Visual would mark the design as non-compliant and display the Compliance Failure Dialog shown in Figure 12-8 indicating that the child of a shell may not be a class:

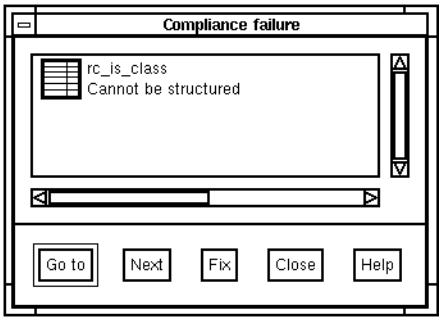


Figure 12-8 Error When Pasting a Class as Child of Shell

SPARCworks/Visual allows you to continue with the operation but until you change the structure of rc_is_class, you will have a design which is not Windows compliant. The Windows compliant toggle in the toolbar displays a red cross to remind you.

12

12.7 User-Defined Widgets

SPARCworks/Visual in Windows mode does not provide any explicit support for user defined widgets at present. However SPARCworks/Visual's C++ model does allow you sufficient flexibility to incorporate any user defined widgets for which you have developed an appropriate MFC based class. SPARCworks/Visual will treat a user defined widget as if it were an instance of the class from which it was derived. If, for example, your widget is derived from a widget not supported by the Motif MFC, you will not be able to add it to your Windows application. If, however, you have a new widget derived from XmPushButton, SPARCworks/Visual will treat it as though it is an XmPushButton.

By default, SPARCworks/Visual will generate Motif MFC code which is based on the CButton class but which creates an instance of your user defined widget. The Windows code will be exactly as if you had used a PushButton. If, however you have an MFC based implementation of your widget, you can specify its class name in the Instantiate as field on the Code Generation page of the Core resources dialog. You will then need to provide a similarly named class for the Motif MFC code.

SPARCworks/Visual will however, still generate the call to the Create method in the Windows code as if the control were a CButton. Although you cannot change this, it should not typically cause a problem.

See the section on page 571 of the *Motif MFC Reference* appendix to locate the necessary files for adding classes to the Motif MFC.

12.8 Links

In the Motif flavors links are pre-defined callbacks. In the Windows flavor they are implemented as simple global functions which are called by a button's message handler. There are, however, some restrictions on how links can be used on Windows. These restrictions only affect whether code is generated for a link, they do not affect the design's compliance.

12.8.1 Destination Widget Not an Object on Windows

If the widget selected as the destination is not mapped to an object on Windows, the Add button in the Edit Links dialog is pink. The link can still be added and will be effective on Motif but it will not be generated into the Windows code. To indicate this the widget in the list of links is pink. See the *Mapping Motif Widgets to Windows* section of the *Widget Reference* chapter on page 536 for more information on which widgets are mapped to Windows objects.

12.8.2 Buttons in Menus as Link Destinations

When adding a link where the destination widget is a PushButton in a Menu, the type of link is restricted to enable and disable. You cannot show or hide a Menu Button on Windows.

12.8.3 File Selection Dialog as Link Destination

FileSelectionBox is implemented on Windows as a CFileDialog, a modal dialog which is shown by calling the DoModal method. This method does not return until the FileSelection is complete or cancelled. On Windows, therefore, only the show link is supported. For both MFC flavors the code is structured so that the DoModal method does not return until the selection is complete or cancelled.

12.9 Manager Widgets and Layout

The Motif manager widgets have no equivalents on Windows. Widgets such as Forms and RowColumns do not exist on Windows and therefore are not generated at all if they are not made a class.

SPARCworks/Visual generates absolute values for sizes and positions as they are at the moment of generation. So, for example, if you have a PushButton that is 100 pixels wide, 30 pixels high and is located at 10, 200 then those explicit values will be used in the Windows code, even though you have not explicitly set the x, y, width and height resources but allowed them to be calculated by the Motif toolkit. In practice this gives very good results; generating Windows dialogs which look very similar to their Motif counterparts.

12.9.1 Fonts

Because SPARCworks/Visual is generating an absolute size for a Windows control, it is important that the size is appropriate for any font that will be used for text displayed in it. The best way to ensure this is to make SPARCworks/Visual use a similarly sized font to display the dialog whilst it is being designed. There are two ways to do this.

The first way is to force the control to use a particular sized font, perhaps by setting the XmNfontList resource for the control or by setting the appropriate font resource on the shell. Consequently the dialogs will be similarly sized.

Alternatively use a resource so that SPARCworks/Visual displays the design windows with a font that approximates to the default font used on Windows. This will cause SPARCworks/Visual to generate absolute sizes that are appropriate to the font. The Motif code will use a default font in the normal way. To make SPARCworks/Visual use a specific font for the design windows use settings similar to the following in your resource file:

```
visu*dialog.labelFontList:\
      -adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
visu*dialog.buttonFontList:\
      -adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
visu*dialog.textFontList:\
      -adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
```

These values may work well for you, but it will depend on the precise font used on your Windows system. It is the size and average width values which are important.

12.9.2 Resize Behavior

SPARCworks/Visual generates OnSize message handlers to provide some resize behavior when the user resizes a dialog. SPARCworks/Visual does not attempt to reproduce exactly the Motif geometry management, rather it generates a handler which will simulate the resize behavior of certain manager widgets. In particular, these are:

- ScrolledWindow
- Form
- Frame
- DialogTemplate

These managers do not need to be classes in order to produce the resize behavior: SPARCworks/Visual generates a handler for the enclosing class that handles all descendant widgets. You can suppress the generation of the resize handler, if you want to provide your own (through a sub class for example), by unsetting the MFC OnSize handler toggle on the Code generation page of the Core resources dialog. See Figure 12-9.

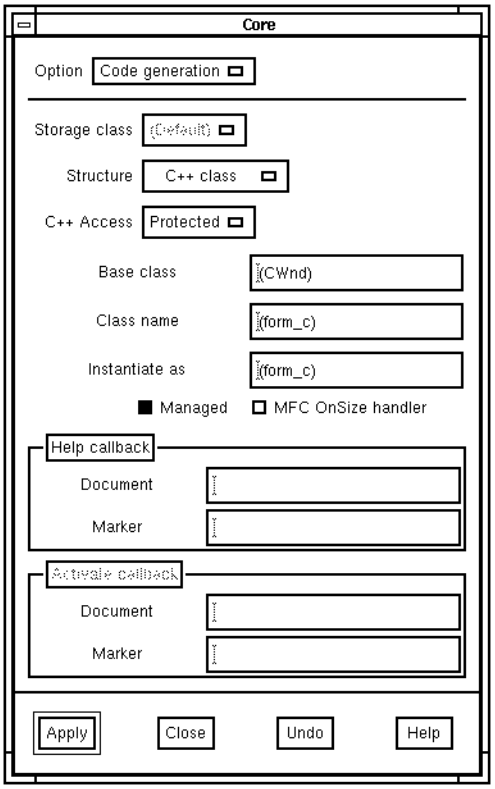


Figure 12-9 The Core Resources Dialog - Code Generation Page

12

12.10 Fonts

In order for a font to be generated into the Windows code *you must use font objects*. This is because fonts must be persistent on Windows. Only by using font objects can you guarantee this. SPARCworks/Visual provides a visual cue by making the Apply button pink if you select an item from the list of fonts in the font selection dialog. If, however, you select an item from the list of font objects, the Apply button is no longer pink.

12.10.1 Fontlists and Compound Strings

If you specify fontlists for your objects in SPARCworks/Visual, the first font in the list will be used for the object on Windows. Compound strings containing a mixture of tags will be translated to Windows using only the first font specified.

12.10.2 Font naming

Fonts in Microsoft Windows are named using a quite different mechanism from that used by X Windows. However, Windows has a quite sophisticated matching algorithm. So, although SPARCworks/Visual uses a quite crude mapping to translate the font specification, even if you specify a font which is not available in Windows, it will probably be substituted with something that looks reasonable.

12.11 Pixmaps, Bitmaps and Icons

Pixmap objects created in SPARCworks/Visual are converted into a Windows bitmap or icon (depending on whether the object is a button or label respectively). This is done for you automatically when you generate a Windows resource file. X monochrome bitmaps are not supported in the translation to Windows.

When you select “Windows Resources” from the generate dialog, SPARCworks/Visual informs you that it will create a resource file and a bitmap/icon file for each pixmap you have created. Bitmap files are generated with the suffix “.bmp” and icon files with the suffix “.ico”. You never need to save the pixmap to a file for Windows but you may wish to do so for the Motif version.

12.11.1 Buttons with Pixmaps

For a Motif Button with a pixmap type label, SPARCworks/Visual generates a CBitmapButton for Windows. One difference between buttons with pixmaps on Motif and CBitmapButtons on Windows is that CBitmapButtons have no border - in fact they do not look like buttons at all. You may, therefore, wish to incorporate a border in your pixmap design.

12.12 Colors

You can set the Background and Foreground color of a widget which will be mapped to an object on Windows. These colors will be generated into the Windows code in terms of their RGB (Red, Green, Blue) values. Windows does not normally have the richness of color that is commonly available on X/Motif. For this reason the colors may not look identical on the two platforms.

12.12.1 Color Objects

Specify the Background and Foreground colors in the usual way in SPARCworks/Visual. Background colors must be color objects - Foreground colors do not have to be.

12.13 Configuring SPARCworks/Visual

There are a number of application resources which apply to SPARCworks/Visual in Windows mode. One of these is the windows flag, indicating if SPARCworks/Visual should default to run in Windows mode. This is described above in the *Starting in Windows Mode* section on page 259.

12.13.1 Setting the Color of non-Windows Resource Fields

By default, SPARCworks/Visual indicates that a field in a resource panel, or a button or any other text field, is not applicable to Windows by coloring it pink. This color can be changed by altering the following line in the SPARCworks/Visual application resource file:

```
visu.mfcTextWarningBackground:#ecc9c9eacdda
```

The example above shows the default file entry - i.e. the color pink. You can change this large number to a more readable color name.

12.13.2 Setting the Filename Filter

In the generate dialog, SPARCworks/Visual provides a default filename filter in the Filter text field. You can change this in the application resource file. Search for the following:

```
visu.c++Filter:*.c
visu.c++StubsFilter:*.c
```

These are the default entries for Motif code generation - both vanilla Motif and MFC Motif. For Windows code generation, you should alter the following lines if necessary:

```
visu.visualC++Filter:*.cpp
visu.visualC++Filter:*.cpp
```

If you wish to share code between the two platforms, you might consider changing the filename filters for the two different flavors so that they are the same. See the following section on File names for more details about points to bear in mind when naming files intended for both platforms.

12.14 File names

Filenames on the PC are restricted to 12 characters in total (including the dot) which must be distributed as no more than eight before the dot and no more than three after. If you wish to share files between the two platforms, this restriction will have to be kept in mind when generating Windows MFC code. Remember, also, that MS-DOS and Windows are not case sensitive. Do not rely on upper and lower case letters to distinguish filenames.

12.14.1 Pixmaps

The above restrictions should also be remembered when naming pixmap objects. When you ask SPARCworks/Visual to generate a Windows resource file after you have created pixmaps, SPARCworks/Visual automatically generates Windows bitmaps and icons in separate files using the name you specified in the pixmap editor. If, therefore, you have specified a name with more than eight characters, you will encounter problems on Windows.

12.14.2 C++ Code

Different compilers have varied conventions acceptable on filename extensions. The suffix '.cxx' seems to be universally supported; most compilers should support '.cpp'; some compilation systems may accept '.C' and '.c++'. Visual C++ will complain if you specify '.c' for a file which contains C++ code.

See the *Setting the Filename Filter* section on page 276 for details of how to change the default filters in the Generate dialog.

12.15 The Callbacks Dialog

The Callbacks dialog has an extra feature when in Windows mode. There is a toggle button labelled Windows MFC.

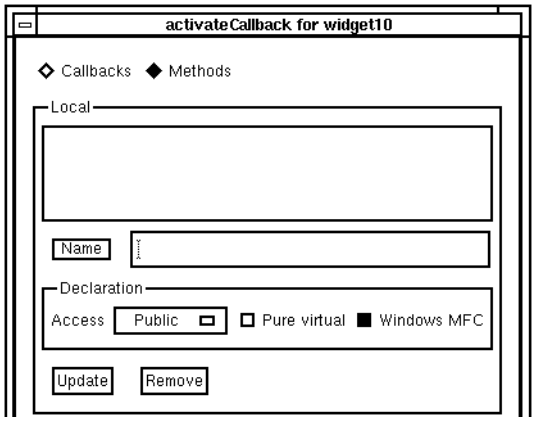


Figure 12-10 Top Area of Callbacks Page with Windows MFC Toggle

This toggle is used to denote whether the method is declared in the class structure of the enclosing class when generating Windows MFC code. The enclosing class is the nearest ancestor which has its structure set to class (either explicitly or automatically).You can see the effect of this toggle by finding the enclosing class and selecting "Method declarations" from the Widget menu. If the Windows MFC toggle was selected, the callback appears in the list of method declarations.

This toggle does not indicate whether or not the method appears in the Windows code stubs - the buttons labelled with the name of the callback indicate this. If they are pink, the callback is not generated for Windows. The Windows MFC toggle gives you control over the method declarations, although you would usually use the default that SPARCworks/Visual provides. Use the Method declarations dialog to declare methods in a class of your choosing or declare them in one of your own classes. They must be declared somewhere.

12.16 DrawingAreas

If a DrawingArea is not the child of a ScrolledWindow, MainWindow or Shell it is created as a basic CWnd class - otherwise it is ignored for Windows code generation. See the *Mapping Motif Widgets to Windows* section of the *Widget Reference* chapter on page 536 for more information.

12.16.1 Adding Drawing Callbacks for Windows

The Motif MFC class library does not include any drawing support; any you require will be platform specific. However, SPARCworks/Visual does allow you to add callback methods which by default are only declared for Motif flavors and Windows message handlers. When in Windows mode SPARCworks/Visual adds a set of additional toggle buttons to the DrawingArea resource panel which can be used to add a Windows message handler in the generated code.

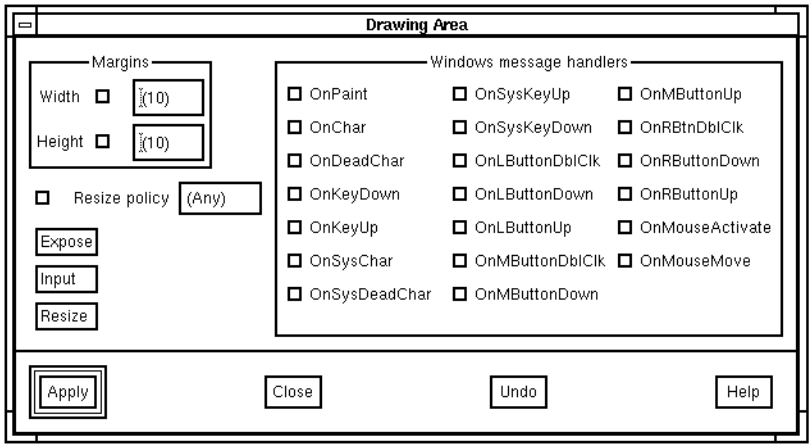


Figure 12-11 The DrawingArea Resource Panel

12.16.2 The Windows Message Handler for DrawingArea

If, for example, you were to select the OnRButtonDown toggle in the panel pictured above, the following stub is added to your callback stubs file:

```
afx_msg void scrolled_win_c::OnRButtonDown( UINT nFlags, CPoint
      point )
{
}
```

Note that `afx_msg` is a pseudo keyword on Windows. The following lines are added to your C++ externs file:

```
//{{AFX_MSG(scrolled_win_c)
afx_msg void OnRButtonDown( UINT nFlags, CPoint point ):
//}}AFX_MSG
DECLARE_MESSAGE_MAP ()
```

This registers the message handler with Windows.

12

12.17 Application Class

SPARCworks/Visual generates an instance of a CWinApp class to the MFC C++ flavors to represent the application. You can configure this class by means of the Application Class dialog which is displayed from the Module Menu. Figure 12-12 shows the dialog.

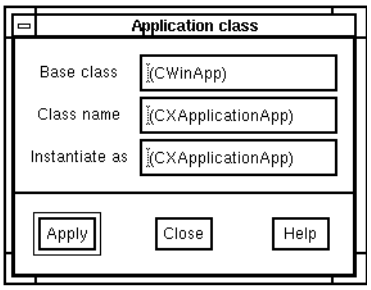


Figure 12-12 The Application Class Dialog

12.18 Code Generation

There are two points which should be taken into consideration when generating code.

12.18.1 Synchronizing Save and Code Files

SPARCworks/Visual has to store the widget id numbers in the save file for definitions so that code can be correctly generated for instances which indirectly modify the layout of an unnamed component. This can cause a problem if the design is changed in a way which affects the widget ids (such as resetting) before the code is generated. SPARCworks/Visual will detect such loss of synchronization and will prompt you to save the file.

12.18.2 Dialog Flashing

In order for SPARCworks/Visual to correctly generate layout information the dialogs need to be realized. SPARCworks/Visual will automatically show and then hide any unrealized dialogs when Windows code is generated. You may see the dialogs appear briefly on the display.

Cross Platform Tutorial 13

13.1 Introduction

This chapter shows you how to produce the following application which you will also build on Windows.

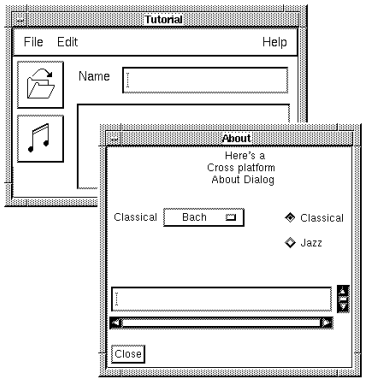


Figure 13-1 Final Application (Motif Version)

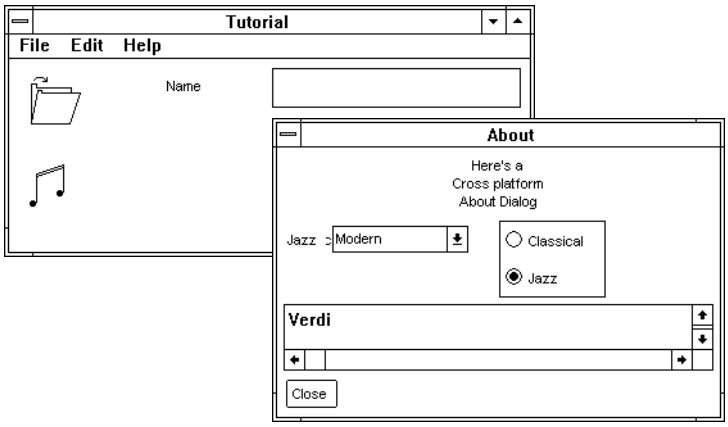


Figure 13-2 Final Application (Windows Version)

There is also a File Selection Dialog which is part of the application.

13.2 Starting Your Design

Make sure that you are running SPARCworks/Visual in Windows mode. See the *Starting in Windows Mode* section of the *Cross Platform Development* chapter on page 259.

1. Create the hierarchy shown in Figures 13-3 to 13-5.

The shell should be set to be an ApplicationShell.

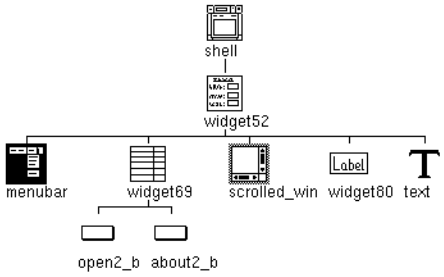


Figure 13-3 The Top Level Hierarchy

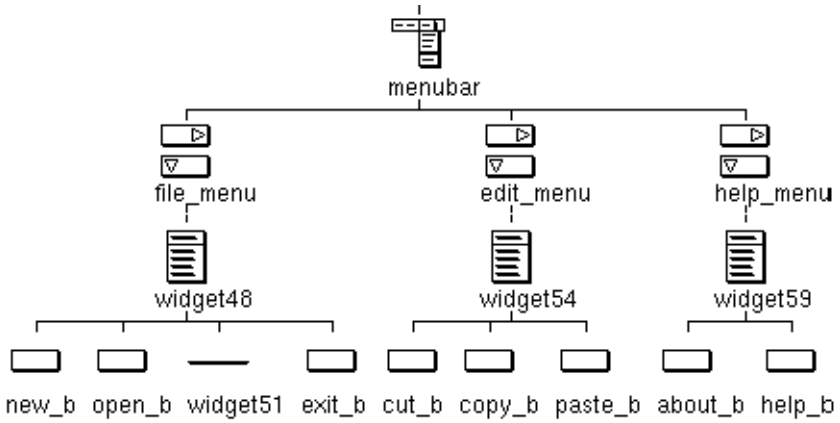


Figure 13-4 The Hierarchy Under the MenuBar

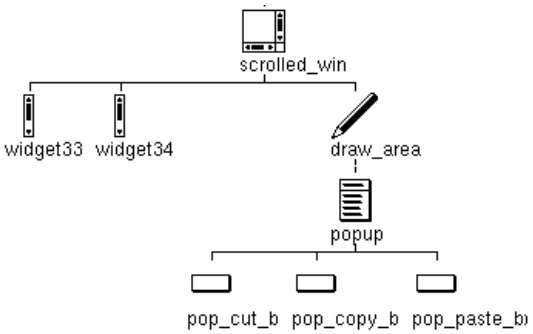


Figure 13-5 The Hierarchy Under the ScrolledWindow

2. Specify the variable names as shown.

3. Use the Form layout editor to adjust your dynamic display to that shown in Figure 13-6.

You may wish to specify attachments and relative positions. Although the Form widget will not be generated into the Windows code (we have not made it a C++ class), attachments and positions are calculated and handled by SPARCworks/Visual in the generated code. In this way the resize behavior will be preserved. See the *Manager Widgets and Layout* section of the *Cross Platform Development* chapter on page 271 for more details.

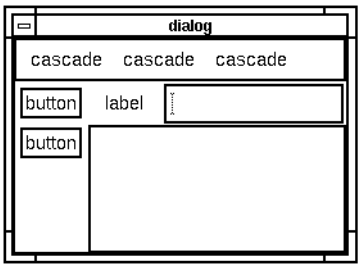


Figure 13-6 The Dynamic Display

4. Save your design into a file named main.xd.

13.3 Creating a Definition

For this example, you are going to create a widget definition in a separate design and add an instance of it in the main design. You will then create a subclass by adding widgets to the instance of the definition. The definition shows the use of inherited functionality and the control of multiple source files on Windows.

13.3.1 The Definition Design

Select New from the File menu.

1. Create the hierarchy shown in Figure 13-7.

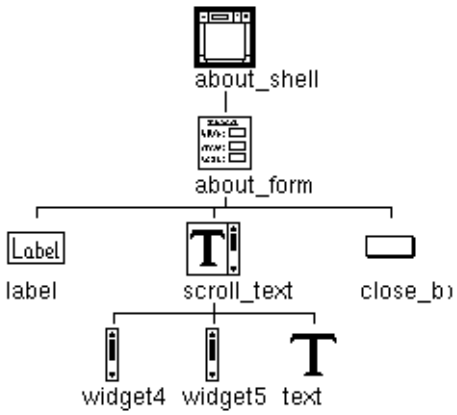


Figure 13-7 The About Dialog Hierarchy

2. Assign widget variable names as shown above.

It is important to name widgets when you are going to create a C++ class definition, otherwise the definition will not contain the correct widgets.

3. Use the Form Layout Editor to create a layout as shown in the following diagram of the resulting dynamic display.

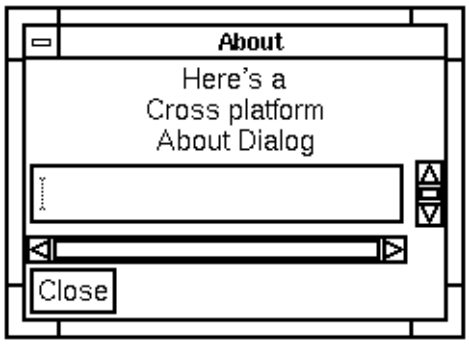


Figure 13-8 The About Dialog

In the example above, the label string resources have been set - buttons, labels etc. have been given meaningful labels. Also the text has been centered. You can do this now, although setting resources is covered in more detail in the *Setting Label Resources* section on page 293.

4. **Save your design into a file named about.xd.**

13.3.2 Adding a Callback

Here we shall add a callback method for the close button which will close the about dialog.

1. **Select the close button, close_b.**
2. **Invoke the resource panel, view the Callbacks page and select the Activate callback. Check that the Methods toggle at the top of the page is set.**
3. **Type `DoClose` into the text box and press the Update button to add the method.**
4. **Close the Activate callback panel and the resource panel.**

Note that for a Motif-only application you would not need to add a callback to close the application from a button - instead you could set the AutoUnmanage resource in the about_form resources to “Yes”. For Windows, however, you need to add a callback. This callback can be shared by the two platforms.

13.3.3 Setting Fonts

In order to make this design look the same on Windows as it does on Motif, we shall set the label, button and text fonts to a font similar to the Windows default font. Whether or not the font given here is suitable will depend on the configuration of Windows you are using. You may wish to try other fonts if this one does not look good. For Windows, font objects must be used.

See the *Fonts* section of the *Cross Platform Development* chapter on page 272 for more details.

1. **Select the main Shell widget, about_shell, and display the shell resource panel.**
2. **Press the Label Font button and make a font object using the font:**

```
-adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
```

If this font is not available on your machine, use a similar size font.

3. **Apply this font object to the Label font, Button font, and Text font.**

You are strongly recommended to read the *Fonts* section of the *Cross Platform Development* chapter on page 272 for a better understanding of the use of fonts in Windows designs.

13.3.4 Making the Design a Definition

Create a widget definition by following these steps:

1. **Select the Shell widget.**
2. **Select Define from the Palette menu. The hierarchy is shown within a colored box to indicate that it forms a definition.**
3. **Select Edit definitions from the Palette menu. Type `about.h` in the Include file field. Press Update and then cancel the dialog.**

13.4 Creating an Instance

You can now add an instance of the definition to your main design. By adding other widgets to it, thereby creating a subclass, you have the appearance and behavior of the original definition as well as the extra features of the additional elements you are going to add. You will then add links and callbacks to complete the design.

13.4.1 Subclassing the Definition

Open the saved file *main.xd*. Notice that there is an extra widget at the bottom of your widget palette. This is the definition from *about.xd*.

1. **Click on the definition palette button. This gives you a new dialog, then set the variable name for the dialog shell to “about_sh”.**
2. **Select the Form widget and add a RadioBox containing two ToggleButtons. Also add two OptionMenus, each with a Menu and three PushButtons under the Form widget. Set their variable names as shown in Figure 13-9.**

3. Change the C++ Access to Public for the two OptionMenus, the two ToggleButtons, and the two Labels.

We are changing the C++ access so that we can access all of these widgets in the callbacks we'll be writing later. C++ access is changed from the Code generation page of the Core resources panel.

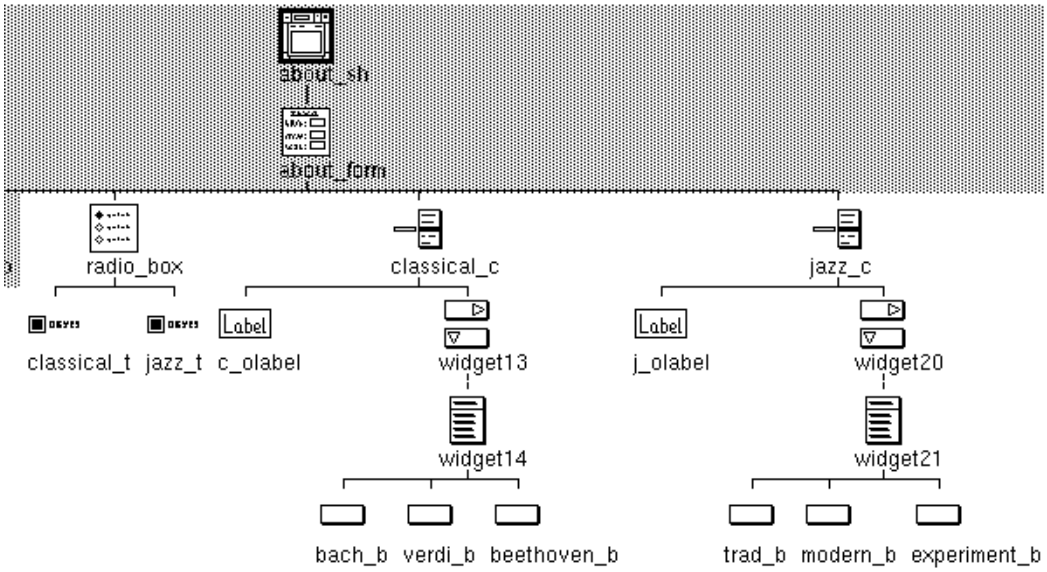


Figure 13-9 Hierarchy Showing New Widgets in Definition Subclass

4. Use the Form layout editor on about_form until your display resembles the one shown in Figure 13-10.

The two OptionMenus are directly on top of one another, each attached to the Form or widgets on all sides. Later you will write a callback which will show and hide the two OptionMenus.

To set the initial values of the OptionMenus, enter the widget name of the corresponding PushButton into the "Last selected" display resource of the OptionMenu, e.g. the classical_c "last selected" resource should be set to bach_b.

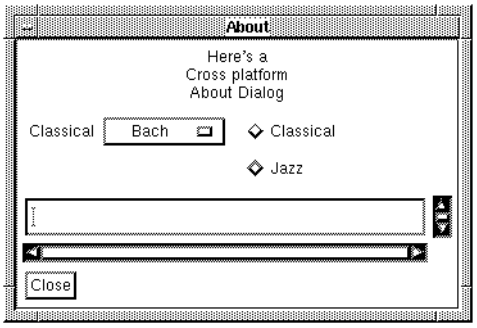


Figure 13-10 The Subclassed Definition

5. Save your design.

13.4.2 Links for the About Dialog

You are going to add a link to display the About dialog from the main window. SPARCworks/Visual is able to generate code to implement links on both Motif and Windows. See the *Links* section of the *Cross Platform Development* chapter on page 270 for restrictions on Windows which should be taken into account when creating links.

1. Select about_b in the Help menu of the Shell dialog and invoke the Edit Links dialog from the Widget menu.

Note that the Add button is pink when a menu button is selected as the destination widget. Show and Hide links are only effective for menu buttons on Motif; Enable and Disable links are effective on menu buttons in both Motif and Windows. However, the menu button currently selected is not the destination widget for this exercise.

2. Select about_sh in the tree.

3. Select the "Show" link in the Edit Links dialog and press the "Add" button to add the Show link to about_sh.

4. Add the same link from about2_b in the RowColumn of the Shell dialog.

To do this select about2_b and select “Edit Links” from the Widget menu. Then repeat step 2.

5. **Save the design.**

13.4.3 Adding Callbacks

You have already added a callback to the about dialog which closes the dialog. Now add callbacks for the two ToggleButtons, classical_t and jazz_t, for all the buttons in the two OptionMenus, classical_c and jazz_c and for the exit button in the file menu.

1. **Add the callback method `ClassicalChanged` to the classical_t ToggleButton of the about_sh dialog for its Value changed callback.**

See the *Callback Methods* section of the *C++ Code Tutorial* chapter on page 229 for further details.

2. **Add the `JazzChanged` callback method to the jazz_t ToggleButton for its Value changed callback.**
3. **Add a callback method `DoSetText` for the Activate callback of each of the Buttons in the two OptionMenus.**
4. **Select exit_b in the file_menu. Add an Activate callback entitled `DoExit`.**

The *Code Generation for Windows* section on page 298 shows you how to generate the stubs for these callback methods and how to fill them in.

13.5 File Selection Dialog

To complete the design, we shall add a file selection dialog. On Windows the file selection operation is supported by a modal dialog box CFileDialog. SPARCworks/Visual, therefore, only allows a file selection dialog as the child of a Shell. To popup the file selection dialog, we shall use links.

1. **Create a Shell and give it the widget variable name file_dialog.**
2. **Add a FileSelectionBox widget. Give the FileSelectionBox widget (the child of the Shell) the name file_select.**
3. **Add a Frame as the child of the FileSelectionBox widget and put a RadioBox with a couple of ToggleButtons inside the Frame.**

4. **Select the Frame again and add a Label.**

13.5.1 Links for the File Selection Dialog

The file selection dialog is opened by pressing one of the open buttons. We shall add a link from each of them to file_dialog.

1. **Select the button open_b in the menu file_menu of the Shell dialog. Invoke the Edit Links dialog.**
2. **Add a Show link to the file dialog shell, file_dialog.**
3. **Add the same link from open2_b in the RowColumn.**

13.6 Popup Menu

Another extra feature we shall add is the ability to popup a menu from the DrawingArea. SPARCworks/Visual cannot automatically generate code to popup menus. You will have to write a brief callback to do this. Motif and Windows have different ways of popping up a menu and so the callback will have to be different for each platform. In this example we shall popup a menu from the DrawingArea. You have already added the menu to the design, now you need to make it work.

1. **Select draw_area and display its resource panel.**

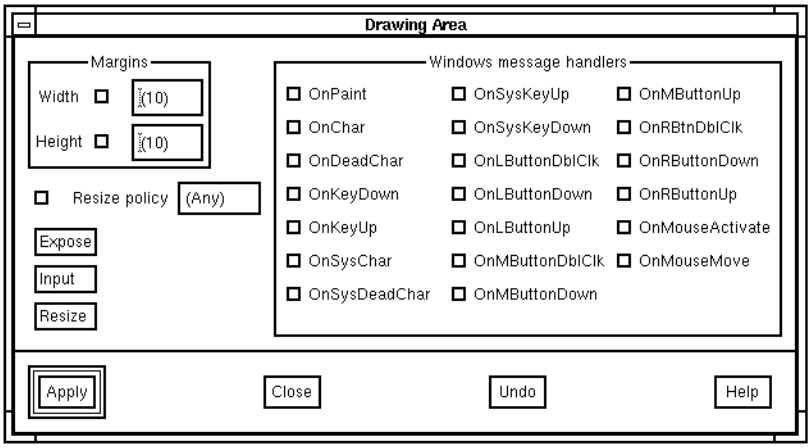


Figure 13-11 DrawingArea Resource Panel

The Motif MFC does not attempt to emulate the Windows drawing or input model, hence the list of Windows target message handlers.

2. Press the button labelled “Input” and add an Input callback method called DoInput.

The Input button is pink, indicating that this callback will have no effect on Windows but we are going to add a callback for the Motif version.

Note that the Windows MFC toggle on the Input callback method page is unset for this callback. This toggle is used to denote whether the method is declared in the class structure of the enclosing class when generating Windows MFC code. It does not indicate whether or not the method appears in the Windows code stubs - the button labelled with the name of the callback indicates this by appearing pink. As this callback is not relevant to Windows SPARCworks/Visual has unset the toggle by default. The toggle allows you control over the method declarations, although you would usually use the default that SPARCworks/Visual provides.

3. Set the OnRButtonDown toggle.

This will generate an appropriate Windows message handler, which will be called in response to the corresponding Windows message, and a matching callback stub.

The *Filling in the MFC Stubs* section on page 299 explains how to fill in the stubs to popup the menu.

4. Save the design.

13.7 Setting Resources

SPARCworks/Visual generates resource files for Motif and Windows. Motif, however, allows a far greater range of control over its widgets. Although there are resources in Windows, they are limited in comparison with Motif/X resources. Windows resources are compiled into the executable file so, unlike Motif, changing a resource on Windows means recompiling the application.

13.7.1 Setting Label Resources

In this example application, we have string, pixmap and keyboard resources. First of all, we shall set the strings of the labels, buttons and shells. To do this, first select the relevant widget then follow these instructions:

1. Set the label resource for the various widgets in the design as shown in Figures 13-12 to 13-16.

Note that as mentioned earlier, the two OptionMenus are directly on top of one another.

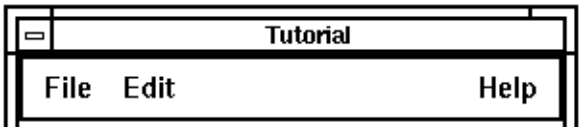


Figure 13-12 MenuBar Labels and Shell Title

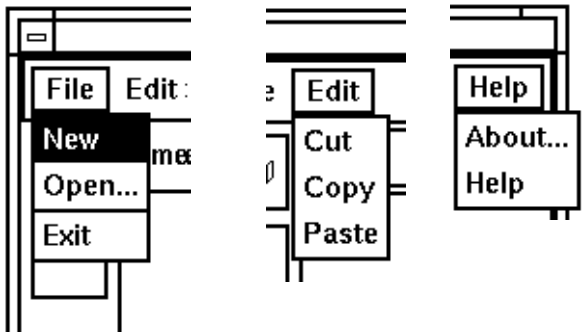


Figure 13-13 Menu Item Labels

Figure 13-14 OptionMenu Labels

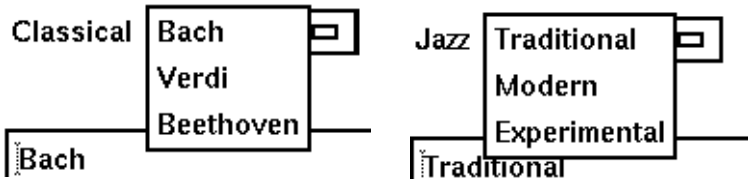


Figure 13-15 OptionMenus Items

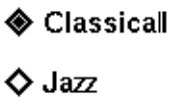


Figure 13-16 ToggleButtons Labels.

2. Set dialog title for the Form child of shell as shown in Figure 13-12

3. **Label the popup menu items pop_cut_b, pop_copy_b and pop_paste_b of the shell dialog, Cut, Copy and Paste respectively.**
4. **Label the ToggleButtons of the file_dialog dialog, New and Append.**
5. **In the file_dialog dialog, give the Label widget that is the child of the Frame the label "Type".**

13.7.2 Generating String Resources

All of these labels are string resources. When you generate the code for the Motif version of your design you can decide where to generate these resources: into the code or into the resource file. For the Windows version all string resources are automatically generated into the source code.

13.7.3 Creating Pixmaps

We shall use pixmaps for the buttons open2_b and about2_b of the shell dialog.

1. **Set the Type Resource of these buttons to Pixmap.**
2. **Create a pixmap object for each button.**

13.7.4 Naming the Pixmap

Remember that filenames on MS-DOS (and Windows) must be no longer than eight characters before the extension. SPARCworks/Visual uses the name to which you bind the pixmap in the basename of the file in Windows mode, so make sure that you have not specified a name longer than eight characters.

13.7.5 Setting Fonts

For the same reasons as discussed in the *Setting Fonts* section on page 286, we must set the fonts so that they will appear the same on Windows as on Motif.

1. **Select the main Shell widget, shell, and display the shell resource panel.**
2. **Press the Label Font button and make a font object using the font:**

```
-adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
```

Again, if this font is unavailable on your machine, use a similar size font.

3. **Apply this font object to the Label font, Button font, and Text font.**

13.8 Building the Application

Now that you have designed the application user interface in SPARCworks/Visual, you can generate the Motif and Windows code. You will need to generate C++ for the main program and the callback stubs along with header files and resource files. As well as code generation, this chapter also covers how to link the dialogs together by filling in the callback stubs.

13.8.1 Controlling the Sources

Some of the generated code will be Motif or Windows specific. By using the Motif MFC, however, some of the code is shared by both platforms. In particular, the callback stubs file can be shared where you are writing code which does not involve the Windows message handler and which is not system dependent. Our example includes both types. The callback from the about dialog is an example which shows how code can be shared by both platforms. The callbacks generated for the main design contain a mixture of platform dependent and independent code and will therefore need to be separated out in order to allow single sourcing of the callbacks which can be shared.

You are strongly advised to keep the rest of the code separate - in separate directories. In this way you are less likely to overwrite one version with another and will avoid including the wrong header file - the inclusion of a header file when a definition is used in a design is generated automatically. You do not control the name of that header file so you could pick up the wrong one if all your sources are in the same directory.

13.8.2 Code Generation for Motif

You will need to reopen about.xd and generate code for that file as well as for main.xd. Follow these steps, generating the code from main.xd first. Note that the order in which the files are generated is important.

1. **Display the Generate panel for C++.**
2. **Check that the Flavor option menu at the bottom is set to MFC Motif.**
3. **Select the Type declarations toggle and enter the name of the header file you are about to generate - main.h in the adjacent box.**

4. **Type the name of the main source file - main.cpp - into the selection field.**

   Remember that you should generate the Motif MFC and the Windows MFC code into separate directories to avoid confusion. Remember also that only a suffix of '.cxx' or '.cpp' is acceptable to both platforms.

5. **Make sure that the main program and links toggles are on for the main design.**

6. **Press Ok.**

7. **Generate the C++ externs using the name main.h. Make sure that the main program and links toggles are not on.**

8. **Generate the C++ stubs. Use the name motif_st.cpp. Make sure that the links toggles are not on.**

9. **Generate a Makefile with both the New and Template toggles set.**

10. **Generate another copy of the stubs file for the main design, giving it the name share_st.cpp. This is for the callback code which can be shared. For the moment it is identical to the file motif_st.cpp.**

11. **Generate the Makefile again, this time with the New toggle off and the template toggle on. This is to ensure that the second stubs file is included.**

12. **Select X Resources from the language menu and generate the resource file. Use the name main.res.**

    For this example, you only need to generate resources for the main design as we did not set any resources in the about dialog. Save the design.

13. **Now open the design about.xd and generate the code in the same way as above. Use the following names:**

    - *about.cpp* - for the source code. Make sure that the main program toggle and links toggles are off for this design.
    - *about.h* - for the C++ externs
    - *about_st.cpp* - for the stubs file

14. **Generate the Makefile again with the Template toggle on and the New toggle off.**

13.8.3 The Makefile

You may have to edit the Makefile in order to access the Motif MFC code, depending on how SPARCworks/Visual has been installed and configured. Check that the `MOTIFLIBS` and `CPPFLAGS` definitions access the Motif MFC library and include files. VISUROOT is the path to the root of the SPARCworks/Visual installation directory:

```
MOTIFLIBS=${VISUROOT}/motifmfc/lib/lib.a -lXpm -lXm -lXt -lX11
CPPFLAGS=-I${VISUROOT}/motifmfc/h -I. -I${XPMDIR}
```

13.8.4 Code Generation for Windows

The steps to generate the code for Windows are almost (but not quite) identical to those described above for Motif. Start by changing the Flavor menu to MFC Windows. SPARCworks/Visual remembers a different set of files for each flavor so that you can use the toolbar Flavor menus and code generation buttons once you have specified the name of the files for the different flavors. Remember that, on the PC, a filename must be eight characters or less before the extension. Visual C++ will complain if it encounters a source file containing C++ code which has been given only a '`.c`' extension. You can specify '`.cpp`' or '`.cxx`'.

Perform the following steps first for the main.xd and repeat for the about.xd design.

1. **Open the file.**
2. **Invoke the Generate panel for C++.**
3. **Check that the Flavor option menu at the bottom is set to MFC Windows.**
4. **Select the Type declarations toggle and enter the name of the relevant header file you are about to generate - main.h for the main design and about.h for the about dialog.**
5. **Type the name of the file - main.cpp or about.cpp - to be generated into the selection field.**

Remember that you should generate the Motif MFC and the Windows MFC code into separate directories to avoid confusion.

6. **Make sure that the main program and links toggles are on for the main design and off for the about dialog.**
7. **Press Ok.**
8. **Generate the C++ externs (using the name you previously specified as containing the Type declarations - main.h for the main design and about.h for the about dialog).**
9. **Generate the C++ stubs. You do not need to generate a stubs file for the about design as you can share the file already generated for MFC Motif. Give the stubs file for the main design the name wind_st.cpp and put it in your MFC Windows directory. Make sure that the Links toggles are off for the stubs files.**

You now have three separate stubs files for the main design. You will see why below. The following step should only be performed on main.xd after you have completed steps 1 to 9 on both main.xd and about.xd and saved the designs.

10. **Open main.xd.**
11. **Generate a Windows resource file giving it the name main.rc.**

A message is displayed informing you that SPARCworks/Visual will also generate a bitmap file for each of the pixmaps you have created, assigning as the basename of each file the name of the corresponding pixmap object.

13.9 Filling in the MFC Stubs

There are now four files containing callback stubs, three of them in your MFC Motif directory. Assuming that your MFC Motif directory is called MMFC and your MFC Windows directory is called WMFC, here is what you should have:

- MMFC/about_st.cpp (The about dialog stubs)
- MMFC/motif_st.cpp (The Motif stubs for the main design)
- MMFC/share_st.cpp (A copy of the Motif stubs for the main design)
- WMFC/wind_st.cpp (The Windows stubs for the main design)

The Motif and Windows stubs for the main design both contain the stubs which will be shared. The about dialog stubs file contains only one callback - DoClose. This file can be shared by both platforms. Fill it in as below.

13.9.1 about_st.cpp

```
/*
** Generated by SPARCworks/Visual
*/

/*
** SPARCworks/Visual generated prelude.
** Do not edit lines before “End of SPARCworks/Visual generated
    prelude”
** Lines beginning ** SPARCworks/Visual Stub indicate a stub
** which will not be output on re-generation
*/

/*
**LIBS: -lXm -lXt -lX11
*/

#include <afxwin.h>
#include <afxext.h>

#include <about.h>

/* End of SPARCworks/Visual generated prelude */

/*
** SPARCworks/Visual Stub about_shell_c::DoClose
*/

void
about_shell_c::DoClose ( )
{
    ShowWindow(SW_HIDE);
}
```

The code added here is Windows code, but with the MFC Motif library it will also work on Motif. This line hides the window whose shell is named about_sh.

You do not need two versions of this file - you just need to remember to copy it across to your PC with the other Windows files.

13.9.2 The Stubs Files for the Main Design

The Motif stubs files for the main design contain, amongst other callbacks, a callback to popup the menu from the DrawingArea and another for the Exit button in the File menu. These callbacks are system dependent because, on the Windows side, closing an application involves Windows message handling and popping up the menu from the DrawingArea uses the Windows drawing model. The other callbacks can be shared. Because of this, we shall need to arrange the stubs files so that we end up with the following:

1. *motif_st.cpp*. This will contain the routine to exit from the Motif application (DoExit) and the DoInput callback to popup a menu from the DrawingArea.
2. *wind_st.cpp*. This will contain the Windows versions of the above routines except that there will be no DoInput callback, instead there is OnRButtonDown, as explained in the *Popup Menu* section on page 291.
3. *share_st.cpp*. This will contain all the other callbacks. These can be shared by both platforms.

We shall now fill in the stubs files. Leave the header information as it is in all files. Alter the files so that they are exactly as in the following listings:

13.9.3 motif_st.cpp

```
/*
** Generated by SPARCworks/Visual
*/


/*
** SPARCworks/Visual generated prelude.
** Do not edit lines before “End of SPARCworks/Visual generated
      prelude”
```

```
** Lines beginning ** SPARCworks/Visual Stub indicate a stub
** which will not be output on re-generation
*/

/*
**LIBS: -lXm -lXt -lX11
*/

#include <afxwin.h>
#include <afxext.h>

#include <main.h>

#include <about.h>
#include <stdlib.h>

/* End of SPARCworks/Visual generated prelude */

/*
** SPARCworks/Visual Stub shell_c::DoExit
*/

void
shell_c::DoExit ( )
{
    exit(0);
}

/*
** SPARCworks/Visual Stub scrolled_win_c::DoInput
*/

void
scrolled_win_c::DoInput ( )
```

```
{
    popup->TrackPopupMenu(0, 0, 0, this, NULL);
}

/*
** SPARCworks/Visual Stub radio_box_c::JazzChanged
** SPARCworks/Visual Stub radio_box_c::ClassicalChanged
** SPARCworks/Visual Stub about_sh_c::DoSetText
*/
```

This file contains code that is specific to the Motif version for the following reasons:

- The method for exiting a Windows application involves the Windows message handler which is not implemented in the Motif version of the MFC
- The Input method of the DrawingArea widget does not exist on Windows and the means of popping up a menu is platform dependent

The comment lines for the shared stubs are left in to stop SPARCworks/Visual subsequently regenerating those stubs to the file.

13.9.4 wind_st.cpp

```
/*
** Generated by SPARCworks/Visual
*/

/*
** SPARCworks/Visual generated prelude.
** Do not edit lines before “End of SPARCworks/Visual generated
       prelude”
** Lines beginning ** SPARCworks/Visual Stub indicate a stub
** which will not be output on re-generation
*/
/*
**LIBS: -lXm -lXt -lX11
*/
```

```
#include <afxwin.h>
#include <afxext.h>

#include <main.h>
#include <about.h>

/* End of SPARCworks/Visual generated prelude */

/*
** SPARCworks/Visual Stub shell_c::DoExit
*/

void
shell_c::DoExit ( )
{
    SendMessage(WM_CLOSE);
}
/*
** SPARCworks/Visual Stub scrolled_win_c::OnRButtonDown
*/

afx_msg void
scrolled_win_c::OnRButtonDown( UINT nFlags, CPoint point )
{
    ClientToScreen(&point);
    popup->TrackPopupMenu( TPM_LEFTALIGN|TPM_RIGHTBUTTON, point.x,
       point.y, this, NULL);
}
/*
** SPARCworks/Visual Stub radio_box_c::JazzChanged
** SPARCworks/Visual Stub radio_box_c::ClassicalChanged
** SPARCworks/Visual Stub about_sh_c::DoSetText
*/
```

Here you can see the Windows version of the code to exit an application and to popup a menu.

Again the comment lines for the shared stubs are left in to stop SPARCworks/Visual subsequently regenerating those stubs to the file.

13.9.5 share_st.cpp

```
/*
** Generated by SPARCworks/Visual
*/

/*
** SPARCworks/Visual generated prelude.
** Do not edit lines before “End of SPARCworks/Visual generated
      prelude”
** Lines beginning ** SPARCworks/Visual Stub indicate a stub
** which will not be output on re-generation
*/
/*
**LIBS: -lXm -lXt -lX11
*/

#include <afxwin.h>
#include <afxext.h>

#include <main.h>
#include <about.h>

/* End of SPARCworks/Visual generated prelude */

/*
** SPARCworks/Visual Stub about_sh_c::DoSetText
*/
```

```
void
about_sh_c::DoSetText ( )
{
    int pos;
    char buf[255];
    if (radio_box->jazz_t->GetCheck()) {
        pos = jazz_c->GetCurSel();
        jazz_c->GetLBText(pos, buf);
    } else {
        pos = classical_c->GetCurSel();
        classical_c->GetLBText(pos, buf);
    }
    scroll_text->SetWindowText(buf);
}

/*
** SPARCworks/Visual Stub radio_box_c::JazzChanged
*/

void
radio_box_c::JazzChanged ( )
{
    if (jazz_t->GetCheck()) {
        about_sh->jazz_c->ShowWindow(SW_RESTORE);
        about_sh->classical_c->ShowWindow(SW_HIDE);
        about_sh->j_olabel->ShowWindow(SW_RESTORE);
        about_sh->c_olabel->ShowWindow(SW_HIDE);
    }
}

/*
** SPARCworks/Visual Stub radio_box_c::ClassicalChanged
*/
```

```
void
radio_box_c::ClassicalChanged ( )
{
    if (classical_t->GetCheck()) {
        about_sh->classical_c->ShowWindow(SW_RESTORE);
        about_sh->jazz_c->ShowWindow(SW_HIDE);
        about_sh->c_olabel->ShowWindow(SW_RESTORE);
        about_sh->j_olabel->ShowWindow(SW_HIDE);

    }
}

/*
** SPARCworks/Visual Stub shell_c::DoExit
** SPARCworks/Visual Stub scrolled_win_c::DoInput
*/
```

This code affects the subclassed about dialog and does the following:

- When the toggle labelled Jazz is selected, the OptionMenu jazz_c is displayed and the OptionMenu classical_c is hidden
- When the toggle labelled Classical is selected, the opposite occurs
- When an item is selected from one of the OptionMenus the text of the selected item is placed into the text field

In this file the comments for the non-shared Motif stubs are left in to stop SPARCworks/Visual from subsequently regenerating them into the file.

13.10 Compiling the Application

Having created the design and generated code you are now ready to compile the application on both Motif and Windows. This chapter shows you how to do this. After the brief section on building your application on Motif, there is an introduction to Visual C++. If you are already familiar with Visual C++, you may wish to skip ahead to the *Creating the Project* section on page 310.

For tutorial purposes, the stages involved in building the Windows version of the application are illustrated below using Visual C++ v1.0 running on Windows 3.1. However, the generated code can also be built using any C++ development environment capable of integrating the Microsoft Foundation Classes, e.g., Symantec C++. Refer to the *C++ Settings* section at the end of this chapter for general information about configuring Windows-based C++ compilers to build the application.

13.10.1 Motif

All you need to do now is make the application by typing `make` at the command prompt in your MFC Motif source directory. Your Makefile should pick up the following files:

1. main.cpp (The main program)
2. motif_st.cpp (The Motif stubs file of the main program)
3. about.cpp (The about dialog definition)
4. about_st.cpp (The about dialog stubs file)
5. share_st.cpp (The shared stubs file of the main program)
6. Including the following header files:
7. main.h
8. about.h

Before running the application, be sure to arrange for the resource file to be read, as explained in Section 8.5.

13.11 Windows

To illustrate compilation of the application on Windows, we will use Microsoft Visual C++ as an example build environment. Visual C++ is a tool for building and debugging Windows-based applications and libraries in an integrated Windows environment. Visual C++ encompasses a number of development tools including Visual Workbench, AppStudio, AppWizard, ClassWizard and several other utilities.

13.11.1 Projects

Visual C++ itself builds your application. You do not have to write a makefile. There is, however, a makefile which is created and maintained by Visual C++. This is always referred to as the *project file*.

In order to maintain the sources of your application, Visual C++ uses the concept of a *project*. A project references all the source files and libraries that make up a program, as well as the compiler and linker commands that build the program. A project is composed of a makefile (which has the same base name as the program with a '.mak' extension), a status file (which has a '.vcw' extension), a definitions file ('.def' extension), as well as the source and resource files. A project is identified by the makefile. This means that when asked for a project (if creating a new project or opening another one), you specify the makefile. You will normally have no need to edit either the status or the definitions file.

13.11.2 Using Visual C++

By using SPARCworks/Visual's cross-platform code generation capabilities much of the use of Visual C++ becomes unnecessary. The objects have been created by SPARCworks/Visual and the code has been generated for them. Without SPARCworks/Visual, you would normally start by designing the user interface objects (the dialogs, controls etc.) and then use various Visual C++ tools to create and manage the code to support them.

Having used SPARCworks/Visual, all that is left for you to do is to create the project, add code for any callback functions, compile and run. Following is a description of how to do this based on VisualC++ for Windows 3.1. Note that Visual C++ may appear slightly different on other Windows platforms although the SPARCworks/Visual generated code is suitable for all MFC targets. For more information on Visual C++, see the relevant Visual C++ documentation.

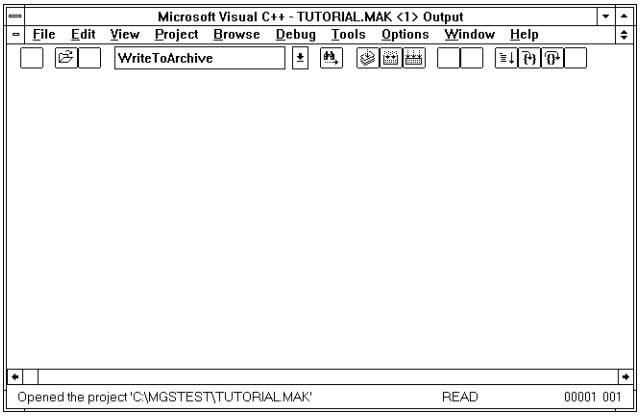


Figure 13-17 The Visual C++ Workbench Window

When you run Visual C++, the window you see is the Visual Workbench window. As its name suggests, this is the place where the work is done and from which all tools are invoked. There is a menubar along the top of the window which contains the tools and various other utilities. To actually *do* anything you will always have to go to this menubar - or use the toolbar buttons, select the function you require and go from there.

13.11.3 Creating the Project

Follow these steps to make your Visual C++ project:

1. Transfer the following files to your PC:

- `main.cpp` (The main program)
- `wind_st.cpp` (The Windows stubs file of the main program)
- `about.cpp` (The about dialog definition)
- `MMFC/about_st.cpp` (The about dialog stubs file)
- `MMFC/share_st.cpp` (The shared stubs file of the main program)
- `main.h`
- `about.h`
- `main.rc` (The Windows resource file)
- `*.bmp` (The bitmap files generated with the Windows resource file)

Note – PC-NFS, available from SunSoft, is a tool designed to make sharing files between your PC and your Solaris system easy.

2. Start Visual C++.

To create a new Project:

3. Choose New from the Project menu. The following dialog appears:

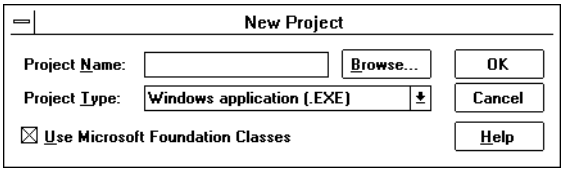


Figure 13-18 The New Project Dialog

4. Press the Browse button in order to locate the directory containing the SPARCworks/Visual generated source files.

5. At the prompt for the filename, type the name of your application with a '.mak' extension.

This is the makefile, the file which identifies the project. It will be created for you and updated as you add and delete source files. Visual C++ creates other files for you too. If you want to know more about the files Visual C++ wishes to create, look in the Visual C++ documentation.

Alternatively, if you wish to open an existing project, select Open... from the File menu and the following dialog appears:

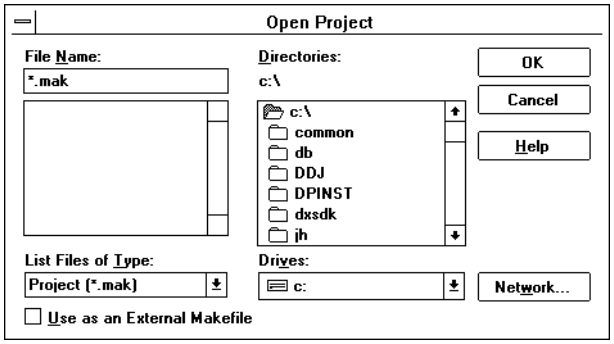


Figure 13-19 The Open Project Dialog

Use the directory browser on the right to find the directory containing the files generated by SPARCworks/Visual.

6. As you are creating a new project, Visual C++ will ask you to specify the source files. The Project Edit dialog appears, as shown below.

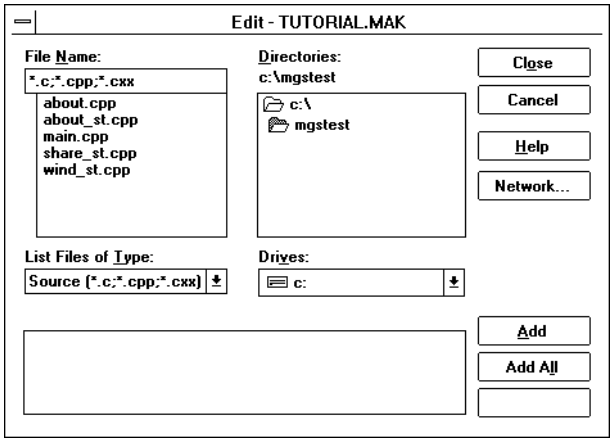


Figure 13-20 The Project Edit Dialog

7. Check that the File Types list is set to request files of type ‘.c’, ‘.cpp’ or ‘.cxx’.

8. **Press Add All to include all the sources in your project. Note that if you give a '.c' extension, Visual C++ will not be happy if the file contains c++ code.**

   You will have to select individual files and press Add if you have source files from another application in the same directory.

   Note that the header files, bitmap files and icon files (if you have them) are not added to the project explicitly. They are referenced from the source files and picked up automatically.

9. **Change the list of file types to '.rc' files. Add the resource file. Note that Visual C++ will only accept one resource file. If you have more than one, you should #include them in one file and add that file to the project.**

10. **Close the Edit dialog.**

11. **From the Options menu, choose Project. The following dialog appears:**

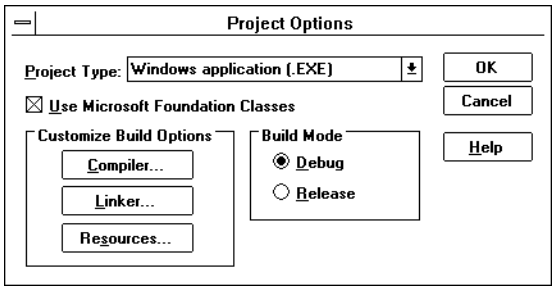


Figure 13-21 The Project Options Dialog

12. **Make sure that you have selected Use Microsoft Foundation Classes. Press the Compiler button. In the next dialog, shown below, select Memory Model from the category list and choose Large.**

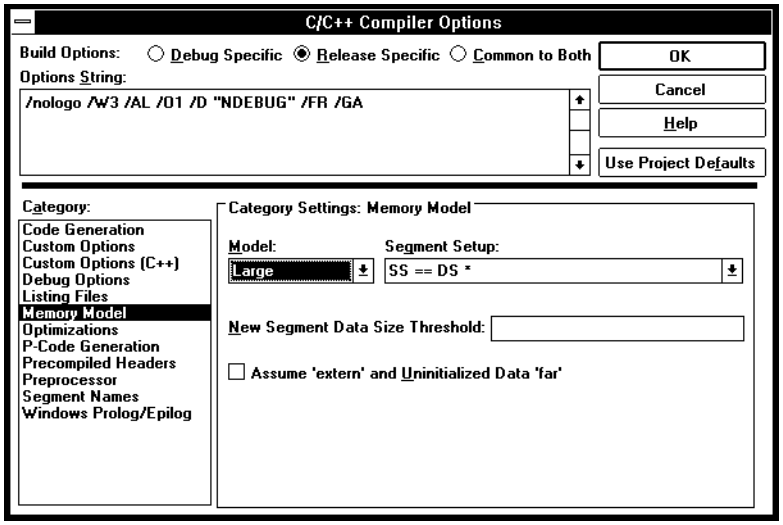


Figure 13-22 The Compiler Options Dialog

This will ensure that the code segment will be large enough for your application. If you experience problems consult your Visual C++ documentation.

13. **In the same dialog, select the Listing Files category, then from the subsequently displayed dialog, deselect Include Local Variables and Browser Information.**

Doing this will speed up the compilation process.

14. **Press OK for these two dialogs.**

Note that in the Project Options dialog shown in Figure 13.21, the Debug option is set. If you encounter problems with your code segment becoming too large, try deselecting this option.

15. **Finally, check that Visual C++ has been installed with the correct list of directories to search for include files. Choose Directories from the Options menu to display the dialog shown below.**

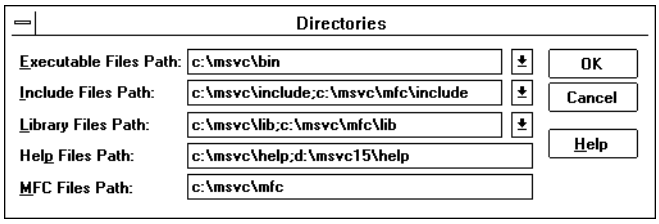


Figure 13-23 The Directories Options Dialog

Make sure that you are including files from the current directory (this is indicated by a '.'(period)). If the system files are not being included you should refer to your Visual C++ installation manual.

Having provided the details of your project, you may now build it.

16. Select "Build <projectname>.exe" in the Project menu or select the Build button on the toolbar at the top of the Workbench window.

Figure 13-24 Toolbar Buttons

Figure 13-24 shows three buttons on the Visual Workbench toolbar which you may wish to use to build your application. The left button will only build the currently active source file. The middle button builds all files which *need* to be built. The right button rebuilds *all* the files in your project.

17. If this is the first time the project has been built, you will be asked if you wish Visual C++ to create a definitions file for you:

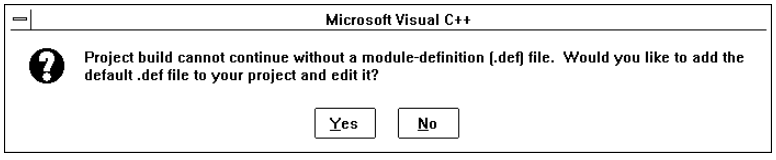


Figure 13-25 Definitions File Message

18. **Select Yes. Visual C++ will then produce a window for you to edit the definitions file. You will not normally need to alter this file, so simply close it.**

19. **Select Build again. Visual C++ will ask you if you wish to build all affected files. Select Yes. All the files you specified as being part of the project are then built. The output from the compiler is displayed in the output window.**

13.11.4 Handling Compilation Errors

If Visual C++ encounters any errors, these are displayed and the compiler stops compiling. You may now press F4 to ask Visual C++ to open the file containing the error and locate you at the relevant point in the file. Subsequent presses of F4 allow you to move through the list of errors, opening files as necessary. Double clicking on an error will also open the relevant file and move to the part of the file where the error was detected. The windows opened onto these files are full text editing windows. You can also view and edit a file in this way by selecting Open... from the File menu.

- **Once the application has built successfully, you can try it out. Select "Execute <projectname>.exe" from the Project menu.**

13.12 C++ Settings

Things to remember when building SPARCworks/Visual-generated applications on Windows platforms:

1. **You will need a C++ compiler and the Microsoft Foundation Class (MFC) include files and libraries installed on your system.**

2. **Configure the build tool to build a Windows .EXE file.**

3. **Make sure the compiler has a valid include path to the MFC header files.**

4. **Make sure the compiler has a valid include path to the subdirectory containing the SPARCworks/Visual-generated source files.**

5. **Compile the code using large memory model settings.**

6. **Make sure the linker links in the MFC libraries or DLLs.**

User-Defined Widgets 14

14.1 Introduction

SPARCworks/Visual is pre-configured with the Motif widget set and can be extended to support widgets from any other Xt toolkit in addition to the default set. Widgets added to SPARCworks/Visual are called *user-defined widgets*. They appear in the SPARCworks/Visual widget palette and users can create them, set their resources and generate code for designs that include them.

SPARCworks/Visual comes with a utility, *visu_config*, which helps you provide the information SPARCworks/Visual needs to support user-defined widgets. You specify which widgets you want to use and provide information about any nonstandard resource types defined by the widgets. *visu_config* generates two C files that serve as a bridge between SPARCworks/Visual and the user-defined widgets.

After generating the *visu_config* files, you build a new version of SPARCworks/Visual from the following components:

- The SPARCworks/Visual object file, *visu.o*
- Object files or archive libraries containing the added widgets
- The code file generated by *visu_config*
- The config file generated by *visu_config*
- Bitmap files for widget icons (optional)
- Pixmap files for widget icons (optional)

- Handwritten code files containing any auxiliary functions (optional)

Icons are recommended but not required. Handwritten code is only required if you want to provide customized popup dialogs, or if you have widgets with special problems described in the *Configuration Functions* section on page 361.

To help you get started, SPARCworks/Visual is distributed with an example configuration file for the Athena widgets, *Athena.xdc*. There are also pre-built configuration files for commercial widgets, such as those for the XRT family of widgets from KL Group. Look in the SPARCworks/Visual release directory for the latest set of supported widget integrations.

14.2 Requirements

User-defined widgets must build and run against the X11 Release 5 X Toolkit Intrinsics. For each widget class you need the public header file and the object file that implements the widget class. Both of these are provided by your widget supplier. The object file may be in an archive library. You may also need the private header file for the widget class.

You need access to the standard UNIX development tools, including the C compiler, linker and *make*.

visu_config requires standard widget information such as the widget class pointer. You should therefore have the widget documentation on hand. If your widget has non-standard resource types, or if you want to supply customized popup dialogs, you may also need to refer to the header files or widget source code to get the required information. See the *Getting Widget Information* section on page 322.

14.2.1 UIL Restriction

UIL code compiles correctly only for designs restricted to the default set of Motif widgets. C or C++ generated code must be used for designs containing user-defined widgets.

14.2.2 Caveats

Since SPARCworks/Visual's dynamic display works by creating actual instances of the widget, any widget you build into SPARCworks/Visual becomes part of the tool. If the widget doesn't function as expected, SPARCworks/Visual may fail when the user adds the widget to a design. Any memory leaks in the widget can affect SPARCworks/Visual and may cause a gradual degradation of performance or a core dump. Even widely-used widgets from standard vendors can have problems. You should therefore test widgets thoroughly before adding them to SPARCworks/Visual.

14.3 Prerequisites

To configure SPARCworks/Visual, you need some knowledge of C and an understanding of common UNIX development tools such as *make*. You don't have to be an expert on X but you need some knowledge of X and the X Toolkit Intrinsics. *visu_config* requires you to supply information about the widget, such as the widget class pointer and symbolic constants representing resource types. For suggestions on how to get the required information from the widget documentation or source code, see the *Getting Widget Information* section on page 322.

14.4 How SPARCworks/Visual Works

Before you start configuring a widget, it is helpful to understand how SPARCworks/Visual uses widgets in the dynamic display. This section describes some aspects of how SPARCworks/Visual works internally.

14.4.1 Creating Widgets

SPARCworks/Visual builds its dynamic display with real widgets, not simulations. It creates widgets by passing the widget class pointer you specify in *visu_config* to *XtCreateWidget()*. The widget class pointer also gives access to information about the widget's resources and their types so that SPARCworks/Visual can build a resource panel for the widget.

The order in which widgets are created and managed in the dynamic display is different from the typical order of operations in an application. When the user clicks on an icon to add a widget to the hierarchy, SPARCworks/Visual creates an instance of the widget, realizes it and manages it before any resources are set. When SPARCworks/Visual reads a hierarchy from a file, however, the hierarchy is created from the bottom up. Each widget is first created, its resources are set and finally it is managed. In either case, create-only resources can't be set ordinarily in the dynamic display since SPARCworks/Visual creates widgets before setting resources.

Some widgets, such as the Athena Form widget, require resources to be set at creation time or don't behave well when managed without children. In these cases, you can customize SPARCworks/Visual's procedure for adding the widget to the hierarchy by specifying a Realize function in *visu_config*. Note that neither of these problems occurs in the generated code. SPARCworks/Visual generates code that creates the hierarchy from the bottom up and sets all resources at creation time.

14.4.2 Highlighting Selected Widget

When the user selects a widget in the tree, SPARCworks/Visual highlights that widget's icon in the tree. It also highlights the widget itself in the dynamic display by swapping the widget's foreground and background colors. Highlighting the widget may cause problems if the widget has a create-only foreground resource. In this case, you can disable foreground swapping on the Widget Edit Dialog, as described in the *Widget Attributes* section on page 328.

14.4.3 Preventing Invalid Hierarchies

SPARCworks/Visual prevents invalid hierarchies by disabling all palette icons for widgets that are not valid children for the selected widget. Also, when a new widget is added to the hierarchy, SPARCworks/Visual doesn't automatically select it if it can't have children.

For user-defined widgets, SPARCworks/Visual looks at the widget's superclasses to determine valid hierarchies. You can also specify configuration functions to customize a widget's requirements for valid child and parent widgets. For more information, see the *Configuration Functions* section on page 361.

14.4.4 Building the Resource Panel

SPARCworks/Visual builds a resource panel for the widget based on resource names and resource types in the widget class record. Resources inherited from a known superclass, such as the Core widget or a Motif parent class, are left off the resource panel and can be set on the resource panel for the superclass.

SPARCworks/Visual automatically assigns resources of standard types to appropriate pages on the resource panel. They can however be explicitly assigned to other pages if required. Most widget resources fall into one of the standard categories; for a summary, see the *Resources* section on page 333.

When the resource panel is displayed, SPARCworks/Visual uses *XtGetValues()* to get the current values of all resources for the widget. All resources except enumerations are converted to text strings and displayed in text fields on the resource panel. The user can set the resource by editing a text string. In some cases, such as fonts, colors and callbacks, the user can supply text indirectly through a popup dialog. For resources that don't already have a SPARCworks/Visual popup, you can supply a popup; see the *Popups* section on page 345.

14.4.5 Setting Resources

When the user applies a new resource value, SPARCworks/Visual converts the text string to a value and applies it to the widget with *XtSetValues()*. It then immediately calls *XtGetValues()* to retrieve all resource values for the widget, converts the values back to text and displays them in the resource panel. This procedure shows immediately whether the toolkit accepted the new value and whether the new value caused other resources to change.

A function called a *resource converter* is used to translate a text string to a resource value and back. For standard resource types, the converter functions are built in and you don't have to do anything in *visu_config*. If your widget has a resource of a non-standard type, *visu_config* lets you specify information about the converter function. For details, see the *Converters* section on page 343. If you don't provide this information, the user can type a text string to set the resource in the generated code or resource file but SPARCworks/Visual can't set it in the dynamic display because it can't convert the text string to a value.

For enumeration resources, SPARCworks/Visual builds an option menu which, by default, is placed on the "Settings" page of the resource panel. SPARCworks/Visual can handle Boolean enumeration resources for user-defined widgets. For other enumerations, you must provide a list of valid values in *visu_config* so that SPARCworks/Visual can build an option menu. For details, see the *Enumerations* section on page 337.

14.4.6 Saving Designs and Code Generation

Saving designs and parsing save files is straightforward. For resource settings, SPARCworks/Visual writes the resource name and the text representation of its value to the *.xd* file. The resource name and value are saved as they appear on the resource panel.

When it generates code, SPARCworks/Visual uses two versions of every resource name. In the generated X resource file, resources are identified by a *name* such as *label*. SPARCworks/Visual gets resource names from the widget class record. In generated code, resources are identified by a *defined name* such as *XtNlabel*. SPARCworks/Visual constructs the defined name by adding an *XtN* prefix to the resource name. If your widget doesn't follow this naming convention, *visu_config* lets you specify a function to construct the defined name. For details, see the *Configuration Functions* section on page 361.

For enumerations, SPARCworks/Visual also uses two versions of each possible value: a *resource file symbol* such as *center* and a *code symbol* such as *XtJustifyCenter*. When you configure an enumeration, you must provide both versions of each value.

When SPARCworks/Visual generates code for any widget class, it generates an *#include* for that class's public header file. *visu_config* lets you specify an *#include* file for each user-defined widget class.

14.5 Getting Widget Information

To configure your widget, you may have to supply one or more variable names and symbolic constants from the widget code. This section summarizes the information you may have to provide. Most of the information you need should be available in the documentation for the widget. If not, you can get it from the widget's public and private header file, from the widget source code (if available), or from your widget supplier's technical support service.

In the following paragraphs we mention naming conventions that are observed by many widget suppliers. However, naming conventions are not invariable rules. Always check the names in the documentation or source code.

14.5.1 The Widget Class Pointer

When you add a widget class in *visu_config*, you need to supply the *widget class pointer*. The widget class pointer is the name of a pointer variable of type *WidgetClass*. This pointer gives access to a structure containing information about the widget class, including a list of resources and their types. SPARCworks/Visual uses this information to build a resource panel for the widget class. By convention, widget class pointers have names of the form *<classname>WidgetClass*.

If you cannot find the widget class pointer in the documentation, look in the public header file for a line such as:

```
externalref WidgetClass fredWidgetClass
```

or

```
extern WidgetClass fredWidgetClass
```

In either case, the widget class pointer is *fredWidgetClass*.

14.5.2 Resource Information

The resource *name* is a character string used to identify the resource in generated X resource files and on the resource panel. SPARCworks/Visual gets this name directly from the widget class record and so you don't need to supply it. This string is usually a straightforward name without a prefix, such as *label*.

The *defined name* is a symbolic constant used to identify the resource name in source code. By convention, the defined name has the form *<Prefix>N<name>*, where *<name>* is the resource name. To find defined names, look in the widget documentation, or look in the public header file for *#define* directives. For example, the following lines from the Athena Form header file identify the defined names *XtNtop* and *XtNbottom*:

```
#define XtNtop "top"
#define XtNbottom "bottom"
```

14.5.3 Non-Standard Resource Types

To configure resources of non-standard types, you need to know the *resource type*. This is not a type such as *unsigned char* but a symbolic constant defined as a string by which the widget class knows the resource type. By convention, resource types have the form <*Prefix*>*R*<*Type*>. For non-standard resource types, especially enumerations, <*Type*> may be the same as the resource name.

If the documentation for your widget gives a resource type as *foo*, look for a line like the following in the public header file:

```
#define XtRFoo "foo"
```

In this example, you would enter *XtRFoo* whenever *visu_config* asks for a resource type. The resource type can also be found in the source code for the widget. The following structure defines a resource whose resource type is *XtRFoo*. Note that you can also get the resource's defined name, *XtNfoo*, from this structure.

```
{
    XtNfoo, XtCFoo, XtRFoo,
    sizeof(foo), XtOffset( FooWidget, foo),
    XtRImmediate, (XtPointer) NULL,
}
```

14.5.4 Non-Standard Enumerations

If your widget has non-standard enumeration resources, you need to specify a list of possible values. In some cases you may have to read the source code to get the names you need. For instructions, see the *Enumerations* section on page 337.

14.6 The Main visu_config Dialog

Run *visu_config*:

```
visu_config
```

The main dialog shown in Figure 14-1 is displayed.

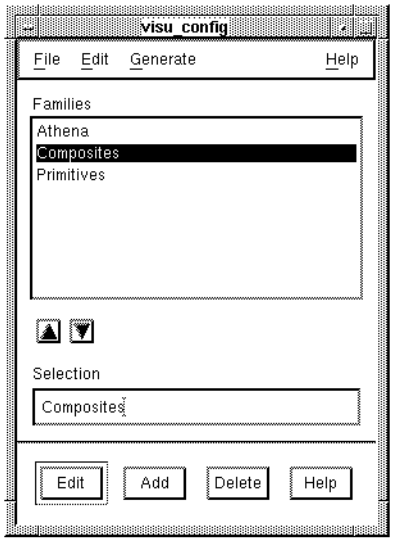


Figure 14-1 The Main visu_config Dialog

14.6.1 Menu Commands

The *visu_config* File Menu has options to save and read files containing your configuration data. By convention, these files have the suffix *.xdc*. Use “Open” to open an existing widget specification file; “Read” to merge another file with the one you are currently editing; “Save” and “Save As...” to save your file and “New” to clear the editing area.

The Edit Menu is used to display the Stop List dialog which lets you remove selected Motif widgets from the SPARCworks/Visual palette. This is discussed in the *Motif Widgets Stop List* section on page 355.

The Generate Menu contains options to generate the two code files needed to build SPARCworks/Visual with the added widgets. For information on generating code from *visu_config* and building SPARCworks/Visual, see the *Generating and Compiling Code* section on page 356.

14.6.2 Families

The main dialog displays a list of *families*. Families are groups of widgets that are displayed together in the widget palette. The list is empty when you start the program; the figure shows the dialog after loading the *Athena.xdc* file supplied with SPARCworks/Visual. We recommend that you open this file and inspect it as you read.

You can organize user-defined widgets into families in any way. Grouping widgets into families has two purposes. First, it keeps the SPARCworks/Visual widget palette to a reasonable size. At any given time, SPARCworks/Visual displays the icons for the default Motif widgets plus one user-defined family. An option menu lets the user switch from one family to another as with the pages of a resource panel. Second, grouping widgets into families also makes it easy to generate versions of SPARCworks/Visual with different sets of families. At code generation time, you can select any group of families from your list. This lets you customize SPARCworks/Visual to support users with different needs and skill levels.

14.6.3 Editing the Family List

To add a new family to the list, type a name for the family in the “Selection” field, then click on “Add”. The family can have any name you choose. It is used to identify the family in *visu_config* and in the option menu in the SPARCworks/Visual widget palette.

To delete a family, select it in the list and click on “Delete”. To reorder the list, select a family and use the arrow buttons to move it up and down. The order of the list determines the order of the option menu in the widget palette.

14.6.4 Suggestions for Organizing Families

You can include the same widget class in more than one family. For example, in *Athena.xdc*, the family named “Athena” contains all the Athena widgets and each of the two smaller families, “Composites” and “Primitives”, contains a subset of the Athena group. When you generate code from *visu_config*, you can decide how you want the widget palette to appear. You can either use the large family to display all the Athena widgets on the palette at the same time, or use the two smaller families to split the widgets between two pages, “Composites” and “Primitives”.

You might want to include a frequently-used widget in more than one family so that the user has access to it at all times regardless of what page of the palette is displayed. To do this, however, you have to enter and maintain two separate copies of the widget configuration information and you should test the icon separately on each page of the palette.

When you generate code from *visu_config*, you can select any group of families from the currently open file but you can't select families from other files. To configure SPARCworks/Visual with widget families from multiple *.xdc* files, use the "Read" option to merge the files before generating code.

14.6.5 Adding and Editing Widgets In a Family

To add or configure widgets in any family, select the family and then click on "Edit" to display the Family Edit dialog. The Family Edit Dialog lets you:

- Add or delete a widget class in the selected family
- Edit the specification for a widget class in the selected family
- Specify instructions for handling non-standard resource types
- Specify a popup dialog for any resource

For details about the Family Edit dialog, see the following sections.

14.7 Widget Classes

The Family Edit dialog has several pages, which you can select from the View Menu. The name of the currently selected family is displayed in the dialog's title bar. To display a list of widget classes in this family, select "Widgets" from the View Menu. Figure 14-2 shows the "Widgets" page for the Athena Composites family.

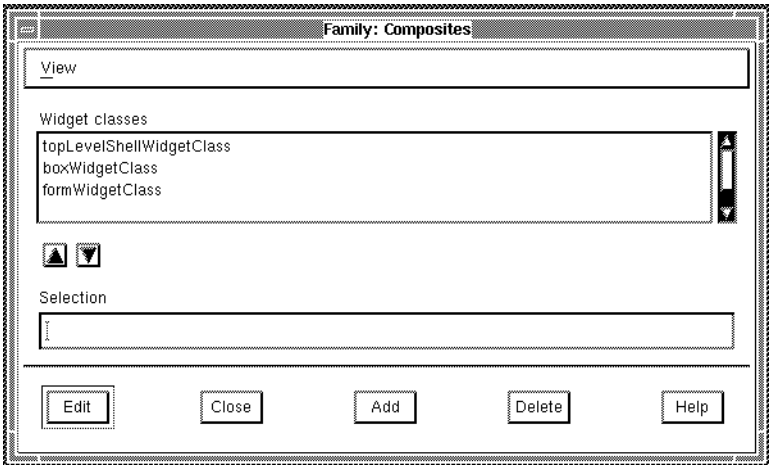


Figure 14-2 The “Widgets” Page of the Family Edit Dialog

14.7.1 Adding a Widget Class

To add a new widget class to the family, enter the widget class pointer in the “Selection” field and click on “Add”. To complete the process, specify attributes for the class as described in the following *Widget Attributes* section.

14.7.2 Editing the Widget Class List

To delete a widget class from the family, select it in the list and click on “Delete”. To reorder the list, select an item and use the arrow buttons to move it up or down. The order of the widget class list determines the order of widgets in the palette.

14.8 Widget Attributes

To specify attributes of a widget class, select the widget class in the Family Edit Dialog and click on “Edit”. This displays the Widget Edit Dialog, shown in Figure 14-3. The title bar displays the name of the widget class you are editing.

This section discusses the attributes set on the left side of the Widget Edit dialog. The right side is used to assign resources to existing or new pages, to specify custom popup dialogs for widget resources, and to override the default resource memory management. For details, see the *Resources* section on page 333.

14.8.1 Applying Changes

When you finish entering widget attributes, click on “Apply” to set the new values. “Undo” reverts to the last applied changes.

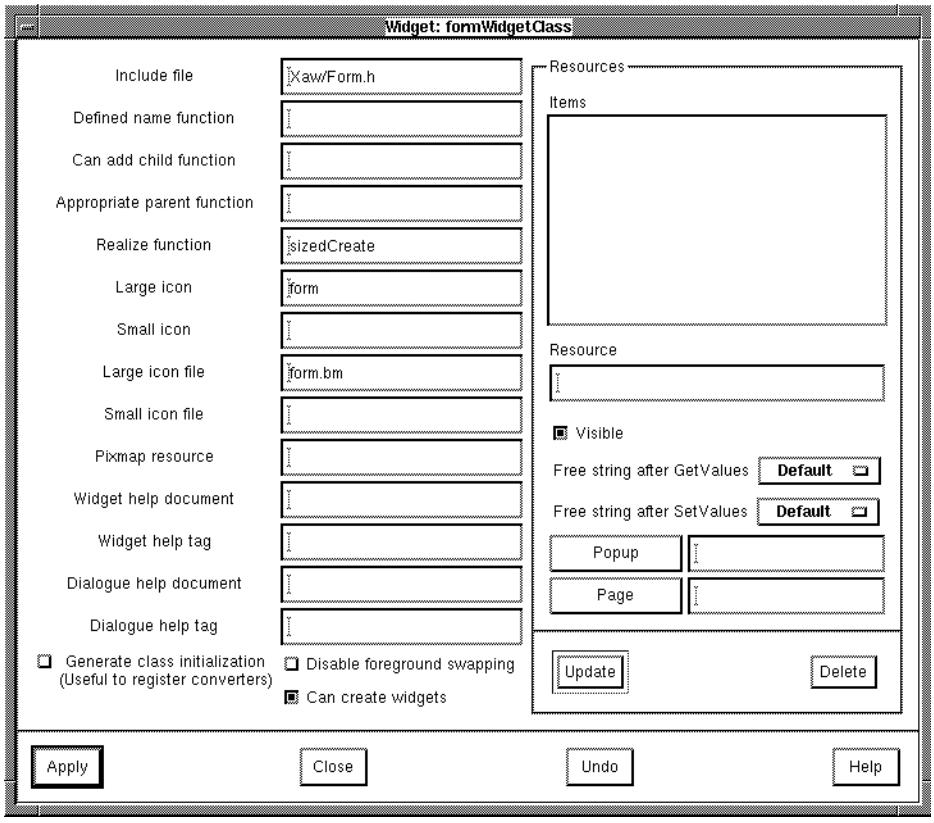


Figure 14-3 The Widget Edit Dialog with Attributes for Athena Form

14.8.2 Include File

In the “Include file” field specify the name of the public header file for the widget class. Enter the file name (usually relative to */usr/include*) without quotes or angle brackets. This file is included in two places: in the Code file generated by *visu_config* and in application code generated by SPARCworks/Visual.

14.8.3 Icons

An icon is an X pixmap or bitmap that represents a user-defined widget in the SPARCworks/Visual widget palette and design hierarchy. For each widget, you can specify both a pixmap and a bitmap. Icon pixmaps are stored in separate files and specified via a SPARCworks/Visual resource.

Icon bitmaps are built into SPARCworks/Visual and are used only if the pixmap resource is not set or the pixmap file cannot be found. If you don’t provide an icon in either form, or the specified icon cannot be found, SPARCworks/Visual displays a button with the widget class name in the widget palette and a crossed square icon in the hierarchy.

14.8.4 Pixmap Resource

Use the “Pixmap resource” field to specify a name for the pixmap resource. After building SPARCworks/Visual, you can set this resource to specify the pixmap file.

For example, if you specify *myWidgetPixmap* as the pixmap resource for a user-defined widget, you can specify a pixmap in the SPARCworks/Visual resource file using the following entry:

```
visu.myWidgetPixmap: /usr/local/newwidget.xpm
```

This specifies */usr/local/newwidget.xpm* as the location of the XPM file. You can use either an absolute or a relative pathname. For details on how to use the pixmap resource, see the *Palette Icons* section of the *Configuration* chapter on page 389.

14.8.5 Bitmap

To specify a built-in icon bitmap, you must provide two items of information: the name of the bitmap and the name of the corresponding bitmap file. To specify an icon for the large-screen (workstation) version of SPARCworks/Visual, use the “Large icon” and “Large icon file” fields. For the small-screen (VGA) version, use the “Small icon” and “Small icon file” fields. You can provide an icon for either version of SPARCworks/Visual, both, or neither.

The large-screen icon must be a 32 by 32 pixel X bitmap and the small-screen icon must be 20 by 20 pixels. You can create icon bitmaps using a tool such as the X bitmap utility. Pixmaps cannot be used for this purpose.

Enter the bitmap name in the “Large icon” or “Small icon” field. The bitmap name is defined in the first line of the bitmap file, as shown below:

```
#define <bitmap_name>_width 32
```

Enter the icon file name in the “Large icon file” or “Small icon file” field without quotes or angle brackets. Because this file is included when you build SPARCworks/Visual, your SPARCworks/Visual makefile must reference (-I) the directory where it is stored. This file does not have to be available to end users.

14.8.6 Help

These attributes let you specify on-line help that is displayed when the user invokes help for the widget on the palette or from the widget’s resource panel. For each help item, you specify a document and a tag. The help system looks in the *UserDocs* subdirectory of the help directory for the appropriate file. The file name is determined by appending the tag to the document. The tag is separated by the value of the *visu.userHelpCatString* resource. By default, this resource is set to ‘.’, yielding a default file name of *document.tag*.

14.8.7 Configuration Functions

There are four fields for specifying configuration functions: a Defined Name function, a Can Add Child function, an Appropriate Parent function and a Realize function. These functions can be used to fine-tune the way SPARCworks/Visual handles the widget in the dynamic display. If your widget uses a configuration function, type the name of the function in the corresponding text field.

The configuration functions perform the following tasks:

- *Defined Name function* – *T*ranslates resource names to appropriate defined names; required only if the widget doesn't follow naming conventions
- *Can Add Child function* – Determines whether you can add a widget of another class as a child of this widget
- *Appropriate Parent function* – Determines whether a widget of another class is an appropriate parent for this widget
- *Realize function* – Adds initialization steps when the widget is created in the dynamic display; required only if the widget cannot be created and managed satisfactorily without children, or requires resources to be set at creation time

The Can Add Child and Appropriate Parent functions are required only for widgets that have special requirements for valid hierarchies. Often these functions aren't needed. If you don't supply them, SPARCworks/Visual uses the rules for the first known ancestor of the widget class. For example, if a user-defined widget is derived from the Primitive class, SPARCworks/Visual uses the rules for the Primitive class and doesn't let the user add children to the widget.

For full definitions, synopses and examples of configuration functions, see the *Configuration Functions* section on page 361. Note that many widgets do not require configuration functions.

14.8.8 Disable Foreground Swapping

Normally, SPARCworks/Visual highlights the currently selected widget in the dynamic display. To disable highlighting for this widget class, turn on the "Disable Foreground Swapping" toggle. Do this only if the widget class has a foreground resource that cannot be set after widget creation time. This is required because SPARCworks/Visual's highlighting procedure can cause problems with such widgets. In all other cases, leave the toggle off.

14.8.9 Can Create Widgets

When the "Can Create Widgets" toggle is on, users can build the widget directly into hierarchies. Leave this toggle on if you want the widget's icon to appear in the palette.

Turn the toggle off to make the widget class an invisible superclass like the Core widget. If you turn the toggle off, the widget doesn't appear in the widget palette and users can't create instances of it directly. SPARCworks/Visual creates a separate resource panel for the class. The resource panel can be accessed by any derived widget classes.

14.9 Resources

By default, when SPARCworks/Visual creates the resource dialog for a widget, it gets the name and type of the resource directly from the widget code and builds a resource panel. It assigns each resource to a page of the resource panel (based on the resource type) and assigns a popup dialog for certain types. For example, resources of type *XtRPixel* are put on the Display page of the resource panel, with a button to pop up the SPARCworks/Visual color editor.

You do not have to do anything to configure a resource unless you want to change this default behavior or unless the resource is not of a type listed in the table of standard types.

14.9.1 Default Handling of Standard Resource Types

Resources of standard types are assigned to pages of the resource panel as shown in the following table:

Table 14-1 SPARCworks/Visual Standard Resource Types

Resource type symbol	Value	Page
XmRDimension	"Dimension"	Margins
XmRFontStruct	"FontStruct"	Display
XmRHorizontalDimension	"HorizontalDimension"	Margins
XmRInt	"Int"	Margins
XmRPixel	"Pixel"	Display
XmRPixmap	"Pixmap"	Display
XmRPosition	"Position"	Margins
XmRPrimForegroundPixmap	"PrimForegroundPixmap"	Display
XmRShort	"Short"	Margins
XmRString	"String"	Display
XmRVerticalDimension	"VerticalDimension"	Margins
XmRWidget	"Widget"	Display
XmRXmString	"XmString"	Display
XtRBoolean	"Boolean"	Settings
XtRCallback	"Callback"	Callbacks
XtRUnsignedChar	"UnsignedChar"	Settings
XmRFontList	"FontList"	Display
XtRFontStruct	"FontStruct"	Settings
XmRXmStringTable	"XmStringTable"	Display

14.9.2 Changing Widget Attributes

The right side of the Widget Edit dialog lets you do the following things for individual widget resources:

- Inhibit the resource from appearing on the resource panel
- Assign the resource to a selected page of the resource panel
- Specify a popup dialog for setting the resource

To customize the attributes of a resource, type the resource name in the Resource field. This should be the defined name, such as *XtNcursorName*.

To remove the resource from the resource panel, turn off the “Visible” toggle.

To specify a different page of the resource panel, click on the “Page” button to display the current list of pages. The list is preset with the default SPARCworks/Visual pages. To add a page, type the name of the new page in the “Page name” field and then click on “Update”. To assign the current resource to a page, select the page in the list and then click on “Apply”.

visu_config automatically invokes the SPARCworks/Visual predefined popups for some types of resources, as shown in the following table. You don’t have to specify a popup explicitly to use these popups for resources of these types:

Table 14-2 Standard Resource Popups

Resource Type	**Popup**
XmRFontList	Font selector
XtRFontStruct	Font selector
XmRPixel	Color selector
XmRPixmap	Pixmap editor
XmRPrimForegroundPixmap	Pixmap editor
XmRXmString	Compound String editor
XtRCallback	Callback dialog

You can specify a popup dialog for any resource. You can select one of the predefined popups or create your own. For more information, see the *Popups* section on page 345.

14.9.3 Nonstandard Resource Types

By default, resources of types not in the table of SPARCworks/Visual standard resource types are placed on the “Miscellaneous” page of the resource panel. The user can set them by typing strings into text fields. The strings are generated into the code exactly as the user types them and resource settings take effect when the generated code is compiled.

This default behavior has some disadvantages. SPARCworks/Visual doesn't recognize the resource type, so it cannot set the resource in the dynamic display. Enumeration values must be spelled and capitalized correctly (including any prefix) or the generated code will not compile.

visu_config offers several ways to refine the way SPARCworks/Visual handles nonstandard resource types:

- You can specify an *alias* for a resource whose type behaves like one of the standard types. For example, if your resource is of a non-standard type that works the same as *XmRInt*, you can instruct SPARCworks/Visual to treat it as *XmRInt*
- You can configure SPARCworks/Visual to handle *enumerations* of types other than *XtRBoolean*. After you do this, the enumeration resource is placed on the "Settings" page of the resource panel from where the user can set it with an option menu. Misspellings are prevented and the resource is active in the dynamic display
- You can specify *converters* for resources of other types. The converter lets SPARCworks/Visual set the resource in the dynamic display
- You can specify a *popup dialog* for any resource. For resources with popup dialogs, SPARCworks/Visual creates a PushButton on the resource panel to invoke the dialog

The following sections give detailed instructions for these procedures.

14.10 Aliases

If your widget uses a resource type that is not in the standard list but has the same semantics as a standard type, you can tell *visu_config* that your type is an alias for the standard type. For example, if you define a type *XtRDegrees* as an integer from 0 to 359, you can set up an alias to specify that *XtRDegrees* is equivalent to *XmRShort*. The user can then set any *XtRDegrees* resource on the "Margins" page of the resource panel and see the results immediately in the dynamic display.

14.10.1 Requirements

The new resource type must have the same semantics as the standard type. Specifically, the process by which a text string is converted to a resource value and back again must be the same for both types.

14.10.2 Specifying an Alias

On the Family Edit Dialog, pull down the View Menu and select “Aliases” to display the dialog shown in Figure 14-4.

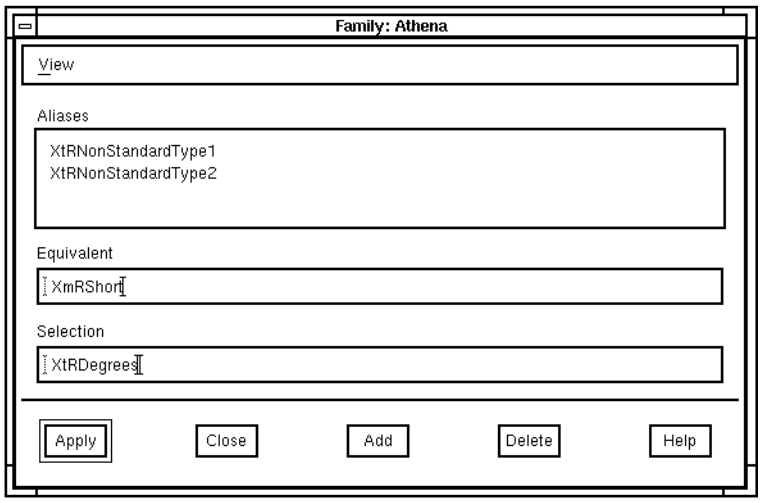


Figure 14-4 The “Aliases” Page of the Family Edit Dialog

To specify an alias, enter the non-standard resource type in the “Selection” field and the name of the corresponding standard in the “Equivalent” field. Click on “Apply” to register the alias.

14.11 Enumerations

Enumeration resources are resources with a fixed set of possible values. You can determine if a widget has enumeration resources by inspecting the documentation. If the documentation lists all possible values for a resource, it is an enumeration. Note that enumerations of type *XtRBoolean* are handled automatically and don’t require the configuration procedure described in this section.

By default, SPARCworks/Visual treats all other enumerations as it treats any unknown resource type. It places them on the “Miscellaneous” page of the resource panel and the user can set them by typing a new value in a text field. The setting is passed to the generated code but has no effect in the dynamic display.

Use the instructions in this section to give SPARCworks/Visual the information it needs to build an option menu for the resource on the “Settings” page of the resource panel and set the resource in the dynamic display.

14.11.1 Configuring an Enumeration

Select “Enumerations” from the View Menu in the Family Edit Dialog to display the “Enumerations” page. Figure 14-5 shows the list of enumerations for the Athena Primitives family.

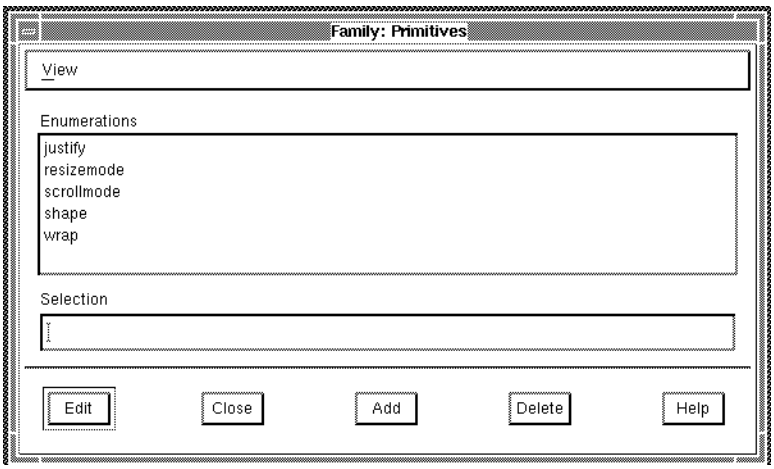


Figure 14-5 The “Enumerations” Page of the Family Edit Dialog

To add a new enumeration, enter a name in the “Selection” field and click on “Add”. This name is only for your convenience in *visu_config*. You can use the resource name from the widget documentation or any other string that helps you identify the resource. Since all enumerations for the family are kept together in one list, it may be useful to include the widget name as well as the resource name.

14.11.2 Configuring Enumeration Values

To complete the process, you need to provide the following information about the enumeration resource:

- The resource type
- A list of possible values
- The default value
- The code symbol for each value
- The resource file symbol for each value

Suggestions for getting the code symbol and resource file symbol are given at the end of this section. To configure an enumeration, select it in the list of enumerations and click on “Edit”. This displays the Enumerations Entry Dialog, shown in Figure 14-6. The title bar displays the name of the enumeration.

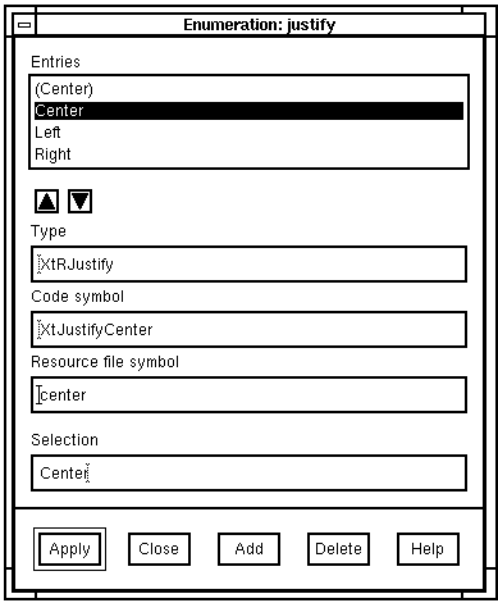


Figure 14-6 The Enumerations Entry Dialog

Specify the resource type and configure each value as described below. When you finish, click on “Apply” to set the new values.

14.11.3 Specifying the Type

Enter the resource type in the “Type” field.

14.11.4 Specifying Values

To get a list of values for an enumeration, look in the widget documentation or the public header file. Make an entry in the “Entries” list for each possible value. Entries in the list are only used in the option menu on the resource panel and can be any names you want. SPARCworks/Visual uses names that are meaningful to the user, such as *Horizontal* rather than *XmHORIZONTAL*.

For each value, type the name in the “Selection” field and click on “Add”. Note that omitting a value isn’t fatal but the user won’t be able to set that value.

14.11.5 Configuring Values

For each value, you must specify the *code symbol* and the *resource file symbol*. The code symbol is a symbolic constant that denotes the value in generated code. The resource file symbol is a string that denotes the value in resource files.

To find the code symbol, look at the list of values in the widget documentation or the widget source code. Type the code symbol in the “Code symbol” field.

Type the resource file symbol for the value in the “Resource file symbol” field. Often, the resource file symbol is the code symbol, converted to lower case and stripped of any prefix, but this is not an invariable rule. For example, the resource file symbol for *XtJustifyCenter* is *center*.

The resource file symbol is not always listed in widget documentation. If it isn’t, you may be able to get it from an example resource file or from the widget supplier’s technical support service. If you have source code for the resource converter, you can get the resource file symbol from the code, as described at the end of this section.

14.11.6 Specifying the Default Value

The first entry in the list of values is reserved for the default value. This entry is used to indicate a resource that is not set explicitly. The option menu should also contain an entry for the same value set explicitly. By convention, the default value is distinguished by putting its name in parentheses, as shown in the following list:

```
(Center)
Center
Left
Right
```

Specify the same code and resource symbols as for the corresponding explicit value.

Note that SPARCworks/Visual builds one option menu for the enumeration type. If your widget has multiple resources of the same enumeration type, they share an option menu. If the resources have different default values, a suggested approach is to enter a generic default value, *(Default)*, on the option menu.

Enter code and resource symbols for any one of the possible default values. This does not result in errors in the generated code or when widgets are initially created in the dynamic display. The dynamic display may be incorrect if the user explicitly sets this resource and then explicitly requests the default value. However, any such problem disappears after the widget is reset.

14.11.7 Specifying Order of Entries

The order of entries in the list controls the order in which they appear in the option menu on the resource panel. Entries can be listed in any order as long as the default value is listed first. To move an entry to a different position, select it in the list and use the arrow buttons to move it up or down.

14.11.8 Getting the Resource File Symbol

This section explains how to get the resource file symbol you need to enter on the Enumerations Entry Dialog from the resource converter code for the widget. The resource converter is a function used to convert a string read from the resource file to the corresponding value. A simple converter may contain fragments like this:

```
if (StringsAreEqual (in_str, "vertical"))
    i = XmVERTICAL;
else if (StringsAreEqual (in_str, "horizontal"))
    i = XmHORIZONTAL;
```

In this example, the string "*horizontal*" is converted to the value *XmHORIZONTAL* and "*vertical*" is converted to *XmVERTICAL*. *horizontal* and *vertical* are the resource file symbols; *XmHORIZONTAL* and *XmVERTICAL* are the code symbols.

You may find it done more indirectly, as shown in the following code:

```
if (!haveQuarks) {
    XtQEhorizontal = XrmStringToQuark(XtEhorizontal);
    XtQEvertical = XrmStringToQuark(XtEvertical);
    haveQuarks = 1;
}
XmuCopyISOLatin1Lowered(lowerName, (char *)fromVal->addr);
q = XrmStringToQuark(lowerName);
if (q == XtQEhorizontal) {
    orient = XtorientHorizontal;
    done(&orient, XtOrientation);
    return;
}
if (q == XtQEvertical) {
    orient = XtorientVertical;
    done(&orient, XtOrientation);
    return;
}
```

In this code, the resource file symbols are represented by the symbolic constants *XtEhorizontal* and *XtEvertical*. To get the resource file symbols, *horizontal* and *vertical*, examine the related header files for lines such as:

```
#define XtEhorizontal "horizontal"
#define XtEvertical "vertical"
```

The code file symbols, *XtorientHorizontal* and *XtorientVertical*, are the end result of the conversion. Note that in this example there is an intermediate step: the strings are converted to quarks and the quarks are used for the comparison. The mechanics of the conversion procedure don't affect *visu_config*.

14.12 Converters

If your user-defined widget has resources of non-standard types other than enumerations, SPARCworks/Visual places them on the "Miscellaneous" page of the resource panel by default. The resources can be allocated to other pages from the Widget Edit Dialog. The user can set these resources by typing a string into a text field. By default, the string is generated into the code or resource file but SPARCworks/Visual doesn't set it in the dynamic display. To make the resource work in the dynamic display, you can configure SPARCworks/Visual with a *converter function* for this resource. The converter function is a function that converts a text string to a resource value.

To configure the converter, select "Converters" from the View Menu. This displays the page shown in Figure 14-7. The "Entries" list contains a list of resource types in this family for which converters have been specified.

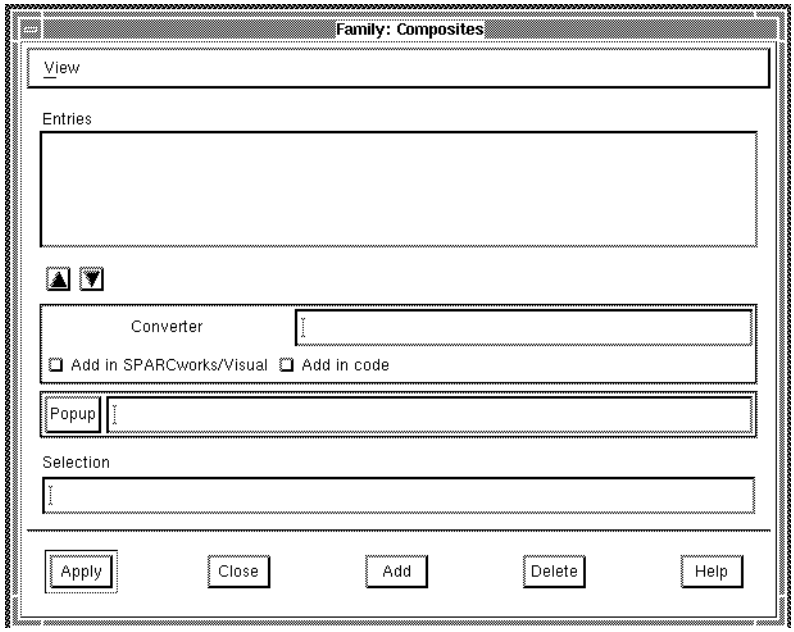


Figure 14-7 The “Converters” Page of the Family Edit Dialog

14.12.1 Resource Type

Enter the resource type in the “Selection” field and click on “Add”.

14.12.2 Converters Added Internally

Many widgets add their own converters internally when the class is initialized. If this is the case, all you have to do is add the resource type to the list. Adding the resource type to the list informs SPARCworks/Visual that the converter is available; otherwise SPARCworks/Visual doesn’t attempt to set the resource in the dynamic display.

To find out whether the widget class adds its own converter, look in the widget documentation or look for a call to *XtSetTypeConverter()* in the code for the widget’s Class Initialize method. If the converter is not added internally or if you are in doubt, instruct SPARCworks/Visual to add the converter explicitly, as described below.

14.12.3 Converters Added Explicitly

If the widget doesn't add converters internally, you can instruct SPARCworks/Visual to add them explicitly. To do this, specify the name of the converter function in the "Converter" box. The converter must be a function of type *XtTypeConverter*. Toggles are provided to add the converter explicitly either in SPARCworks/Visual (for use in the dynamic display), in the generated code, or in both. In general, if you have to add the converter explicitly, both toggles should be on.

14.12.4 Popup Dialog

The "Popup" button lets you specify a popup dialog to be used for setting all resources of this type. For details, see the *Popups* section below.

14.13 Popups

Resource popups are dialogs used to set a resource, such as SPARCworks/Visual's color and font selectors. For resources that have popups, SPARCworks/Visual creates a button on the resource panel to invoke the popup, in addition to the usual text field. Figure 14-8 shows popup buttons on the SPARCworks/Visual Core resource panel.

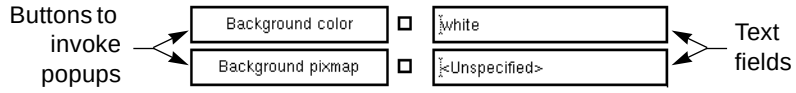


Figure 14-8 Popup Buttons on Core Resource Panel

14.13.1 Popups for Individual Resources

You can specify a popup dialog for any individual resource. To do this, use the right side of the Widget Edit Dialog, shown in Figure 14-9. Select the name of the resource in the list. If you haven't yet added the resource to the list, enter the defined name of the resource, such as *XtNresourceName*, in the "Resource" field and click on "Update".

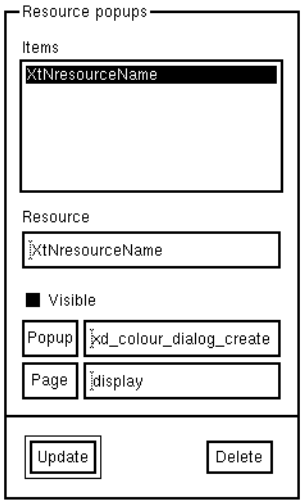


Figure 14-9 Popup Portion of Widget Edit Dialog

This dialog uses resource names and not resource types. Therefore, you can specify different popups for different resources of the same type. For example, if you specify a popup dialog for a specific resource of type *XmRInt*, that popup is not displayed for other resources of that type.

After you enter the resource name, click on the “Popup” button to display the Popups Dialog. Select a popup using the instructions in “The Popups Dialog section below.

14.13.2 Popups for Resource Types

If you specify a converter for a resource type, you can specify a popup dialog for that type. To do this, click on “Popups” on the “Converters” page of the Family Edit dialog. This displays the Popups Dialog. Select a popup using the instructions in the following section.

14.13.3 The Popups Dialog

The Popups Dialog is shown in Figure 14-10.

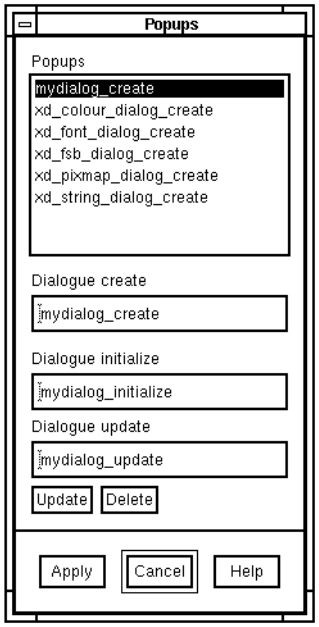


Figure 14-10 The Popups Dialog

The Popups Dialog displays the current list of popups. The list is preset with the built-in SPARCworks/Visual popups:

Table 14-3 Built-In Popups

Name	Popup
xd_colour_dialog	Color selector
xd_font_dialog	Font selector
xd_fsb_dialog	File selection dialog
xd_pixmap_dialog	Pixmap editor
xd_string_dialog_create	Compound String editor

To apply a popup to the currently selected individual resource or converter type, select the popup in the list and click on “Apply”. Note that all popup dialogs return the resource value in the form of a string such as *"red"* or *"<big_font>"*. Therefore they work for resources that are declared as strings but are used to specify fonts, colors, or filenames.

14.13.4 Custom Popup Dialogs

You can create your own popup dialog, add it to the list and apply it to any resource or converter type. Three functions are required for each popup dialog: a create function, an initialize function and an update function. Enter the name of each function in the appropriate text field and then click on “Update” to add the popup to the list. Popups in the list are identified by the name of the create function.

14.13.5 Code Requirements

The popup functions are invoked at different points in SPARCworks/Visual, as shown below:

Function	When called
create function	When the dialog is first popped up.
initialize function	Each time the user clicks on the resource button.
update function	Each time a new widget is selected.

Add callbacks on widgets in your popup dialog to furnish hooks for additional functionality such as setting the new value, editing the value, accessing help and popping down the dialog. Your dialog should have at least the following standard callbacks:

Callback	Functionality
Apply callback	Sets new resource value in source text widget.
Close callback	Pops down the dialog by unmanaging it.

The Text or TextField widget on the resource panel (called the *source text widget*) is used to pass information from the dialog to the resource panel. When the user sets a value on the popup dialog, the value should be converted to text and set into the source text widget. The user can then click on “Apply” on the resource panel to set the value in the dynamic display, just as if the text had been typed by hand.

You can build your dialog in SPARCworks/Visual or code it by hand. The example at the end of this section shows a popup dialog that was built in SPARCworks/Visual.

14.13.6 Create Function

The create function is called the first time the user clicks on the resource button to pop up the dialog. This function should create the widget hierarchy for the dialog.

```
void popup_create ( Widget parent )
```

The *parent* parameter is the Application Shell widget for SPARCworks/Visual, to be used as the parent widget for the dialog. This parameter is required to call functions such as *XmCreateDialogShell()*. You can build popup dialogs in SPARCworks/Visual and use SPARCworks/Visual's generated creation procedure as the create function, or write a create function that calls it. In this case, pass *parent* on to the function generated by SPARCworks/Visual.

Note that the initialize function is called immediately after the create function and so you don't have to manage the widgets if you do it in the initialize function.

14.13.7 Initialize Function

The initialize function is called every time the user clicks on the resource button to pop up the dialog. The first time the dialog is invoked, the initialize function is called after the create function. The function should make the dialog visible by managing it and initialize any fields in it. The initialize function is passed the source text widget, the currently selected widget and the resource name.

```
void popup_initialize( Widget source_text, Widget current)
```

The *source_text* parameter is the source text widget on the resource panel. This widget can be used to obtain the resource value currently displayed on the resource panel. Since the source text widget is used to pass back the new value from the dialog, the initialize function should also save the source text widget in a static variable so that it is available later.

current represents the currently selected widget. The initialize function should check whether the currently selected widget has the expected resource because the user can invoke the popup dialog from the resource panel after selecting a widget of a different type. Note that *current* is *NULL* if the user has deleted all widgets in the hierarchy.

resource_name contains the resource name (a string such as “label”), not the defined name. This parameter is especially useful when you use the same popup to set more than one resource.

14.13.8 Update Function

The update function is called for every popup dialog each time the selection changes in the widget hierarchy.

```
void popup_update( Widget current )
```

current represents the newly selected widget. The update function should make the “Apply” button insensitive if the newly selected widget is of a different class, or if *current* is *NULL*. If you use the same dialog to set multiple resources, the safest approach is to make the “Apply” button insensitive in all cases. The user then has to click on the resource button again in order to use the popup. The extra step invokes the initialize function and ensures that the intended resource is set.

14.13.9 Popup Example

This example shows a simple resource popup consisting of a slider that is used to select an integer value. When the user clicks on “Apply” in the popup, the slider’s value is converted to a text string and placed into the text widget in the resource panel.

The dialog itself was built in SPARCworks/Visual. The Shell for the dialog was designated a Data Structure resulting in the structure shown below. Named widgets in the dialog are easy to access through the structure pointer *foo_dialog*. For example, *foo_dialog->scale* accesses the Scale widget.

```
typedef struct foo_dialog_s {
    Widget foo_dialog;
    Widget form;
    Widget scale;
    Widget apply;
    Widget close;
} foo_dialog_t, *foo_dialog_p;
static foo_dialog_p foo_dialog = (foo_dialog_p) NULL;
```

The following function is generated to create the Shell and all its children. The body of the generated function is omitted.

```
foo_dialog_p create_foo_dialog (Widget parent)
{
/* SPARCworks/Visual generated code to create the dialog omitted here.*/
}
```

In the module prelude a static variable is created to hold the source text widget. The initialize function provides the source text widget.

```
static Widget source_text;
```

The dialog has an “Apply” button and a “Close” button, each with an Activate callback. The “Apply” button invokes the callback function shown below. This callback function gets the current value of the Scale, converts it to a text string and sets it into the source text widget. Note that this doesn’t set the resource; the user must still click on “Apply” on the resource panel.

```
static void
foo_do_apply (Widget w, XtPointer client_data, XtPointer call_data )
{
    int i;
    char buf[52];
    XmScaleGetValue ( foo_dialog->scale, &i );
    sprintf ( buf, “%d”, i );
    XmTextSetString ( source_text, buf );
}
```

The dialog also has a “Close” button. The Activate callback function on this button simply unmanages the child of the dialog’s Shell widget.

```
static void
foo_do_close (Widget w, XtPointer client_data, XtPointer call_data )
{
    XmAnyCallbackStruct *call_data = (XmAnyCallbackStruct *)
        call_data;
    XtUnmanageChild ( foo_dialog->form );
}
```

The create function calls the SPARCworks/Visual generated creation function and saves the widget structure.

```
foo_create( Widget parent )
{
    foo_dialog = create_foo_dialog( parent );
}
```

The initialize function extracts the text from the source text field, converts it to an integer and sets the Scale to reflect the current value. It saves the source text field and so the new value set using the Scale can be applied to the source text field. It enables or disables the “Apply” button depending on the class of the current widget and makes the dialog visible.

```
foo_initialize( Widget text, Widget current)
{
    char *source_value;
    int i;
    source_text = text;
    source_value = XmTextGetString( source_text );
    i = atoi( source_value );
    XtFree( source_value );
    XmScaleSetValue( foo_dialog->scale, i );
    XtSetSensitive( foo_dialog->apply,
        current && XtIsSubclass (current, fooWidgetClass ) );
    XtManageChild( foo_dialog->form );
}
```

The update function enables or disables the “Apply” button, depending on the class of the current widget.

```
foo_update( Widget current )
{
    XtSetSensitive( foo_dialog->apply,
        current && XtIsSubclass( current, fooWidgetClass ) );
}
```

14.14 Resource Memory Management

SPARCworks/Visual assumes a default memory management model for XmString and String (char *) type resources. For XmString type resources this model assumes that the widget will copy the XmString both on SetValues and on GetValues (i.e. the application can free the XmString after a GetValues or SetValues).

For String resources it is assumed that the widget copies the String on SetValues but not on GetValues (i.e. the application only frees a String after a SetValues). If you have a resource that does not conform to this model (typically an XmString resource that is not copied by the widget on GetValues), then you can override SPARCworks/Visual's default behavior using the two Option Menus in the resource section. Where an XmString is not copied on GetValues, you should set the GetValues Option Menu to “Don't Free”. Similarly, if you have a String resource that is copied by the widget on GetValues (for example XmNmnemonicCharset in a Label Widget) then you should set the GetValues Option Menu to “Free”.

It is important to make sure that SPARCworks/Visual does not free memory that it should not free as this will cause SPARCworks/Visual to crash. It is less important if SPARCworks/Visual is not freeing memory that it should free, as this will simply accumulate as a memory leak.

14.15 XmStringTable Resources

For SPARCworks/Visual to handle XmStringTable resources correctly, you must also specify the integer type resource which is used as a count for the number of entries in the table. Add a resource specification for the XmStringTable.

14.16 Headers

visu_config generates two code modules, the Config file and the Code file. *visu_config* lets you specify a list of headers for each of these files. To specify headers, use the Family Edit Dialog. Select “Headers” from the View Menu to display the “Headers” page, shown in Figure 14-11.

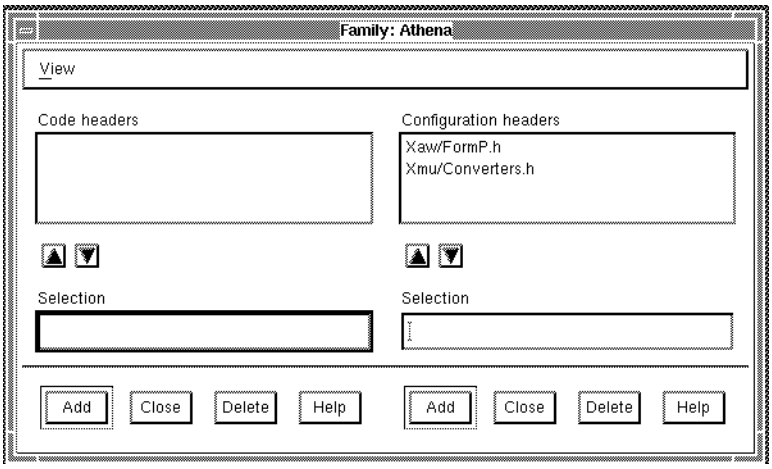


Figure 14-11 The “Headers” Page of the Family Edit Dialog

There is a separate list of headers for each file. To add a header, type the filename, without quotes or angle brackets and click on “Add”. To delete a header, select it and click on “Delete”. To reorder a list, select any entry and use the arrow buttons to move it up or down.

The Code file defines the widget class records for user-defined widget classes. *visu_config* automatically generates an *#include* for the widget class header file which it takes from the Widget Edit dialog. Often no additional Code headers are needed.

The Config file contains a list of user-defined widgets, enumerations, and aliases. Widget headers for user-defined classes aren’t automatically generated to this file. If you configure *visu_config* with non-standard enumerations or resource types, include the header file in which they are defined.

The easiest way to find out what headers are needed is to generate and compile the code. If the compiler returns an undefined reference, find out which header contains the necessary definition and add it to the header list for that file. Then regenerate the code and try again.

14.17 Motif Widgets Stop List

You can use *visu_config* to *stop* selected Motif widgets from appearing in SPARCworks/Visual. Stopped widgets do not appear in the widget palette. They work correctly if read in from an existing design file but cannot be selected in the hierarchy or created interactively. For example, you can use this feature to prevent users from using the PanedWindow widget if it isn't in your company's style guide.

To stop a widget, pull down the Edit Menu in the main *visu_config* dialog and select "Stop list" to display the dialog shown in Figure 14-12:

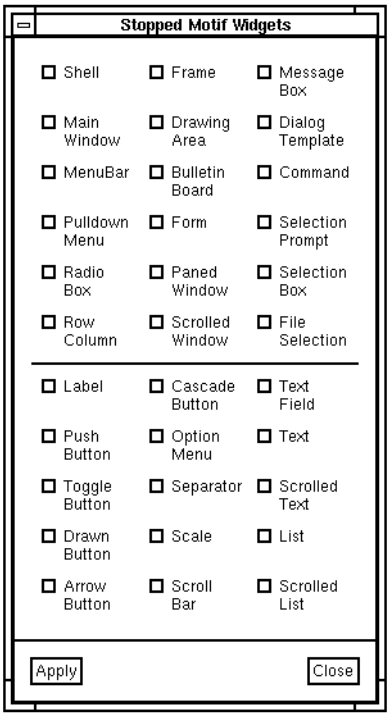


Figure 14-12 The Stopped Motif Widgets Dialogs

To remove a Motif widget from the widget palette, set the appropriate toggle and click on "Apply".

Widgets can also be stopped by setting the *visu.stopList* resource, as described in the *Configuration* chapter. Note that widgets stopped using the resource can be reactivated easily using the resource file, while widgets stopped in *visu_config* can only be reactivated by rebuilding SPARCworks/Visual.

User-defined widgets cannot be stopped using this dialog. You can select a set of user-defined families on the Generate Dialog, as discussed in the following section.

14.18 Generating and Compiling Code

The Generate Menu has two options, Config and Code which are used to generate the two configuration files. The pages displayed for each option are similar, as shown in Figure 14-13. Use the toggle buttons to select the families you want to include. Generate both files, using the same set of families for each.

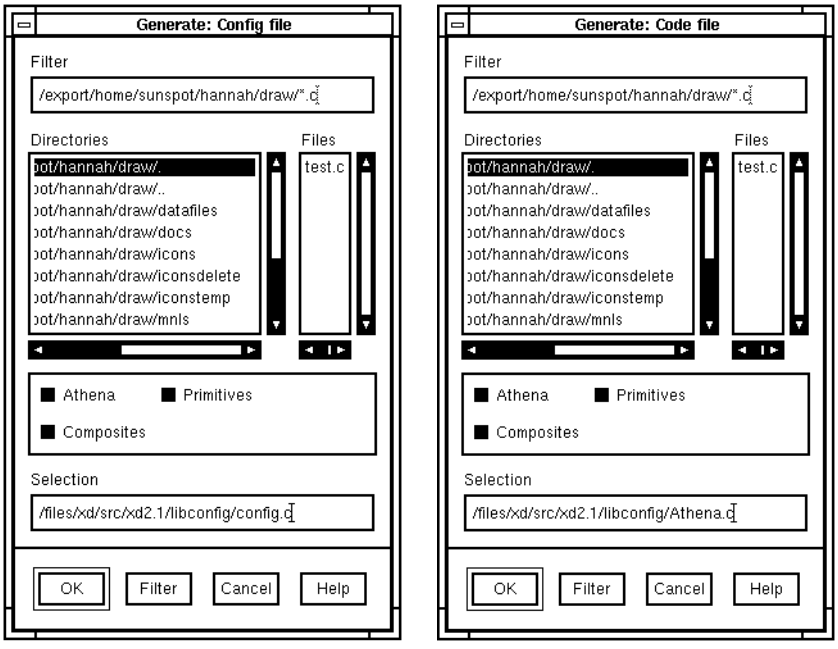


Figure 14-13 The Config and Code Generate Dialogs

14.18.1 Compiling

The example makefile in *$*VISUROOT*/user_widgets/Athena*[1] compiles a Config file named *config.c*, a Code file named *Athena.c* and a file containing configuration functions, *Athenaextras.c*. These files and all the Athena icon bitmaps, are located in *$*VISUROOT*/user_widgets/Athena*. You can use this makefile to compile the Athena example. The result is an executable file called *visu.bin*.

When you configure with other widgets, use the example makefile as a starting point. Make sure that the makefile references all directories containing icon bitmap files and required header files.

If the compilation fails, inspect the generated code to find the problem. The cause may be a missing header. You can supply additional headers on the Headers page of the Family Edit dialog. The compiler also catches any misspellings on any of the *visu_config* dialogs. Fix the problem, regenerate the Code and Config files, then recompile.

Note that the linker detects any misspelled function names. If the misspelling occurred in *visu_config*, correct the problem, regenerate the Code and Config files, then recompile.

14.18.2 Invoking SPARCworks/Visual

The executable file is invoked with the correct environment by setting the environment variable USER_WIDGETS to the name of the user widgets directory (in this example “Athena”) and invoking the standard visu command in $VISUROOT/bin. This causes the local bitmaps or color_icons directories to be added to $XBMLANGPATH and the local app-defaults directory to replace $VISUROOT/app-defaults in $XFILESEARCHPATH.

1. $VISUROOT is the path to the SPARCworks/Visual installation root directory.

A site-wide default can be set up by making a symbolic link called “local” in $VISUROOT/user_widgets to the user widgets directory of choice. In this case all users will get that version of visu by default unless they explicitly override it with a setting of USER_WIDGETS. When USER_WIDGETS contains a value that is not recognized, or the visu.bin in that user widgets directory has not been built, the site default, if configured, or original “vanilla” visu is invoked instead.

14.19 Testing the Configuration

This section describes a recommended testing procedure for the SPARCworks/Visual interface for a user-defined widget. Use *visu_config* as suggested to fix any problems. Note that these tests are designed to detect problems with the way in which the widget was configured into SPARCworks/Visual; they don’t test the widget itself.

14.19.1 Creating a Widget

This test is not appropriate if you turned off the “Can create widgets” toggle in the widget attributes panel.

Run SPARCworks/Visual and verify that the icon for the user-defined widget is correct. If you use the small screen SPARCworks/Visual, invoke SPARCworks/Visual with the name *small_visu* and verify that the right icon displays in this case, too.

Create a hierarchy that contains an instance of your widget. If SPARCworks/Visual fails when the widget is added, you may need a Realize function. For details, see the *Configuration Functions* section on page 361. If the failure is accompanied by an X error message about zero height/width windows, try using *sizedCreate()* (found in *Athenaextras.c*) as the Realize function for the widget.

If this does not work, try setting the “Disable foreground swapping” toggle in *visu_config*.

14.19.2 Foreground Swapping

If you have not disabled foreground swapping, create a hierarchy containing the user-defined widget. Select the Shell in the hierarchy and then select the user-defined widget. Verify that the widget in the dynamic display highlights correctly when selected. If this causes a problem, set the “Disable foreground swapping” toggle in *visu_config*.

14.19.3 Defined Name

Create a dialog containing an instance of the user-defined widget with every resource set. Generate C from SPARCworks/Visual and compile it. If it fails to compile, you may get a message like this:

```
XtNfoo undefined
```

If you get such a message, first verify that the generated code includes the right public header for the widget class. If it doesn’t, correct the header on the “Include file” field of *visu_config*’s Widget Edit dialog.

If the header is being generated correctly, you may need a Defined Name function. Look in the public header for a line such as:

```
#define <something> "foo"
```

If <*something*> is not *XtNfoo*, you need a Defined Name function. For details, see the *Configuration Functions* section on page 361.

14.19.4 Pages

If you have specified that resources should appear on specific pages, verify that they do and that all the required pages are present.

14.19.5 Converters

Display the “Miscellaneous” page of the resource panel for the user-defined widget. Verify that you can type valid resource values into the text widgets and that they are correctly applied to the widget. If you get a message indicating that there is no resource converter, you need to use the “Add in SPARCworks/Visual” setting in *visu_config*.

14.19.6 Enumerations

Display the “Settings” page of the resource panel for the user-defined widget. Make sure the option menus all have the default value in parentheses at the top of their menus.

Display the “Miscellaneous” page if there is one. Enumeration resources appear on this page if they weren’t configured in *visu_config*. If enumeration resources do appear on this page, go back and add them using *visu_config*.

Set each value for the enumeration in turn, including the default. Verify that each value works as expected in the dynamic display and that the generated code compiles correctly.

14.19.7 Popup Dialogs

If you specified custom popup dialogs for any resources, display the page of the resource panel on which each resource appears. Verify that the resource panel displays a button for each resource with a popup dialog. Click on the button. Verify that the dialog appears and is correctly initialized with the current value for that resource.

Set the resource in your dialog. Verify that the text widget on the resource panel updates correctly. Apply the setting from the resource panel and verify the result in the dynamic display.

If your dialog has a “Close” button, verify that it works as expected and that the dialog reappears when you click on the button on the resource panel.

14.19.8 Code Inspection

Finally, verify that the generated code is correct. To check the generated code, set each resource in turn, generate a C code file and an X resource file and inspect them to see that you get what you expect.

Code inspection for all the Motif widgets forms part of the SPARCworks/Visual release process. Therefore if you have a user-defined widget that is derived from a Motif widget, you can concentrate on testing resources that are specific to the user-defined widget. This is also true if you have a user-defined widget that is derived from another user-defined widget that you have already tested.

14.20 Configuration Functions

visu_config lets you provide configuration functions to customize SPARCworks/Visual's handling of user-defined widgets. This section provides definitions and examples of the configuration functions.

To add a configuration function, specify the name of the function on *visu_config*'s Widget Edit dialog for the widget class, then regenerate the Code and Config files from *visu_config*. Edit your makefile to compile and link the file containing the code for your configuration functions.

For examples of the configuration functions that were used to integrate the Athena widgets into SPARCworks/Visual, see *$*VISUROOT*/user_widgets/Athenaextras.c*[1].

14.20.1 Realize Function

By default, SPARCworks/Visual creates widgets in the dynamic display by calling *XtCreateWidget()*. You can supply a Realize function to substitute for this. A Realize function is only needed for widgets that cause problems when created in SPARCworks/Visual.

14.20.2 Realize Function Prototype

The Realize function has the following form. Note that it takes the same parameters and returns the same result as *XtCreateWidget()*.

```
Widget realize( char *name, WidgetClass class, Widget parent,
      ArgList args, Cardinal arg_count )
```

The *ArgList* passed to a Realize function is always empty.

1. $VISUROOT is the path to the SPARCworks/Visual installation root directory.

14.20.3 Realize Function Example

Some composite widgets, such as the Athena Form widget, cannot be realized without children unless their dimensions are explicitly set at creation time. Otherwise the widget is created at zero size, causing an X error. To solve this problem in the dynamic display, you can supply a Realize function like the one shown below, found in *Athenaextras.c*. This function initializes the widget's width and height resources to non-zero values, then calls *XtCreateWidget()* and returns the result.

```
WidgetsizedCreate( char *name, WidgetClass class, Widget parent,
       ArgList args, Cardinal arg_count )
{
    Arg al[2];
    int ac=0;
    XtSetArg(al[ac], XtNheight, 20); ac++;
    XtSetArg(al[ac], XtNwidth, 20); ac++;
    return XtCreateWidget ( name, class, parent, al, ac);
}
```

The Realize function is only used when SPARCworks/Visual creates the widget in the dynamic display. It has no effect in the generated code.

14.20.4 Defined Name Function

In order to generate both code files and X resource files, SPARCworks/Visual uses both the resource name, such as *label*, and the corresponding defined name, such as *XtNlabel*. SPARCworks/Visual gets the name directly from the widget class record. By default, SPARCworks/Visual derives the symbolic constant from the name by adding an *XtN* prefix.

If your widget doesn't follow this convention, you can configure SPARCworks/Visual with a Defined Name function. The Defined Name function is a custom procedure that converts a resource name to its corresponding symbolic constant. To find out whether a widget needs a Defined Name function, look in the public header for the widget class. The header file contains lines like the following that define the symbolic constant and its value:

```
#define XtNlabel "label"
#define XtNfont "font"
```

```
#define XtNinternalWidth "internalWidth"
```

You need a Defined Name function if any of the defined names don't follow the naming convention. For example, many widget toolkits, including Motif, use a different prefix:

```
#define XmNbuttons "buttons"
#define XmNbuttonSet "buttonSet"
#define XmNbuttonType "buttonType"
```

14.20.5 Defined Name Function Prototype

The Defined Name function has the following form:

```
char *defined_name ( char *name )
```

The Defined Name function is passed a character string containing a resource name and should return a character string containing the corresponding defined name. Your Defined Name function can refer to SPARCworks/Visual's internal Defined Name function, *def_defined_name()*. This function simply adds the default *XtN* prefix to the resource name.

14.20.6 Defined Name Function Example

The defined name for the Athena Clock widget resource *hands* is not *XtNhands* but *XtNhand*. Therefore, the Clock widget needs the following Defined Name function:

```
char *clock_defined_name( char *name )
{
    /*
     * XtNhand is defined as hands, so can't just put
       * XtN on the front
     */
    if ( strcmp ( name, "hands" ) == 0 )
        return "XtNhand";
    return def_defined_name ( name );
}
```

All Clock resources except *hands* follow the naming convention and so *def_defined_name()* is used to convert them.

14.20.7 Can Add Child and Appropriate Parent Functions

You can supply Appropriate Parent and Can Add Child functions to define the rules for valid parent-child relationships involving user-defined widgets. These rules control SPARCworks/Visual features such as graying-out of palette icons, automatic selection of newly created widgets and dragging icons in the construction area.

Often these functions aren't needed. If you don't supply them, SPARCworks/Visual uses the rules for the first known ancestor of the widget class. For example, if a user-defined widget is derived from the Primitive class, SPARCworks/Visual uses the rules for the Primitive class and doesn't let the user add children to the widget.

The Appropriate Parent and Can Add Child functions are combined with rules for other widgets. For example, SPARCworks/Visual already has a rule that MenuBar widgets can only have CascadeButtons as children, so you don't need an Appropriate Parent function to prevent the user from making your widget a child of a MenuBar. You only need to supply functions if your widget class has additional rules.

If you configure SPARCworks/Visual with a non-Motif composite widget, the widget class should have a Can Add Child function to prevent it from having Motif children. Motif widgets assume that their parents are Motif widgets and SPARCworks/Visual may core dump if a Motif widget is made a child of a non-Motif widget.

14.20.8 Appropriate Parent Function Prototype

The Appropriate Parent function is called when the user tries to add a widget of the user-defined class to the hierarchy, or tries to drag or copy the user-defined widget to another parent. This function determines whether the user-defined widget can be added as a child of the selected widget.

The Appropriate Parent function has the following form:

```
Boolean is_appropriate_parent ( Widget parent, WidgetClass
      childclass )
```

The first parameter is an instance of a widget in the hierarchy that is a proposed parent widget; the second is a pointer to your new widget class. The function should return *TRUE* if it is valid to add a child of your new class to the proposed parent widget and *FALSE* otherwise.

Because the Appropriate Parent function is passed the instance of the proposed parent widget, you can make rules based on either the class or the state of the parent widget. For example, you can check the parent widget's dimensions, ancestor widgets, or other children before letting the user add a new child of your user-defined class.

14.20.9 Appropriate Parent Function Example

By default, SPARCworks/Visual lets Motif Manager widgets, such as the Form and Row Column, have children of any type. Appropriate Parent functions let you restrict the widget to being a child of only certain classes. For example, if your widget can only be a child of a DrawingArea, supply an Appropriate Parent function that returns *TRUE* if the proposed parent widget is a DrawingArea and *FALSE* if not. The code for this case is very simple:

```
Boolean drawing_area_parent ( Widget w, WidgetClass class )
{
    if ( XtClass ( w ) == xmDrawingAreaWidgetClass )
        return True;
    return False;
}
```

14.20.10 Can Add Child Function Prototype

The Can Add Child function has the following form:

```
Boolean can_add_child (XWidget_p parent, WidgetClass childclass)
{
...
}
```

This function is used for two purposes. SPARCworks/Visual calls the Can Add Child function to determine whether a widget of a specific class is a valid child of the user-defined widget and calls the Can Add Child function to determine whether it should automatically select a newly created instance of the user-defined widget.

The first parameter is a pointer to an existing instance of the user-defined widget class. The parent widget instance is passed as an *XWidget_s* structure, an internal SPARCworks/Visual data type that represents a widget instance. One field of this structure is a pointer to the widget instance. For documentation on the *XWidget_s* structure, see *$*VISUROOT*/user_widgets/hdrs/xwidget.h*.

The second parameter may be a pointer to a proposed child widget class, or may be *NULL*. If the second parameter is non-*NULL*, SPARCworks/Visual is inquiring about a proposed child class. The function should return *TRUE* if the child can be added and *FALSE* if not. Note that you have a pointer to the parent *instance* but not to the child *class* because the child widget hasn't been instantiated. Your function may make rules based on the current state of the instance of the user-defined widget. For example, you can write a function that lets your widget accept only a limited number of children. If the second parameter is *NULL*, the user has just created a widget of this class.

If the Can Add Child function returns *TRUE*, SPARCworks/Visual selects the newly created widget in the hierarchy; otherwise, the parent widget remains selected. In this case, the Can Add Child function should usually return *TRUE* if the widget can have children of any type and *FALSE* otherwise.

14.20.11 Can Add Child Example

The following example shows a Can Add Child function.

```
Boolean paned_can_add_child ( XWidget_p xw, WidgetClass class ) {

/* For newly created instance of this widget class, make the newly created
widget the currently selected widget in the hierarchy. */
if (class == NULL)
    return TRUE;

/* Allow all children except Drawing Area and ScrollBar. */
if  ( class == xmDrawingAreaWidgetClass ||
      class == xmScrollBarWidgetClass )
    return FALSE;
else
    return TRUE;
}
```

Command Line Operations 15

15.1 Introduction

This chapter describes command line code generation and the commands provided for conversion of UIL and GIL code into SPARCworks/Visual save files.

15.2 Generating Code from the Command Line

This feature lets you include SPARCworks/Visual code generation as a step in an application Makefile. The command line synopsis is:

```
visu [-csepAKCSEulbarmRMFWX [code_file]] [-windows] -f filename
```

Table 15-1 visu command line options

Switch	Meaning
c	C
s	C stubs
e	C externs
p	C pixmaps
A	Force ANSI C (use with **-c**, **-s** and **-e**)
K	Force K&R C (use with **-c**, **-s** and **-e**)
C	C++
S	C++ stubs
E	C++ externs

Table 15-1 visu command line options

Switch	Meaning
u	UIL
l	C for UIL
b	C externs for UIL
a	UIL pixmaps
r	X resource file
m	Makefile
X	X resource file (synonym for r)
M	Generate Motif flavor C++
F	Generate Motif MFC flavor C++
W	Generate Windows MFC flavor C++
R	Windows resource file
windows	Start SPARCworks/Visual in Windows mode
f file	Specify input file

code_file represents the file to be generated. If you do not specify a *code_file*, SPARCworks/Visual generates code to the last target file specified in your source file for the given language.

filename represents the design file (*.xd*) to be used as a source for the code generation. You must always specify a *filename*. If you do not also specify a *code_file*, use the *-f* separator to indicate that you are providing only one filename.

The *-windows* switch specifies Windows mode. For further information on starting SPARCworks/Visual in Windows mode see the *Starting in Windows Mode* section of the *Cross Platform Development* chapter on page 259.

The *F*, *W* and *R* switches must be used in conjunction with the *-windows* switch.

15.2.1 Examples

The command:

```
visu -c foo.c foo.xd
```

generates C code from the design in *foo.xd* into the file *foo.c*.

The command:

```
visu -c -f foo.xd
```

generates C code from *foo.xd* into the target file that was specified the last time C code was generated from *foo.xd* via the Generate Dialog.

You can use a single command to generate multiple files using one of the following forms:

```
visu -c -e -s -f foo.xd
```

or

```
visu -c <c_file> -e <extern_file> -s <stub_file> foo.xd
```

SPARCworks/Visual exits with status zero if successful and non-zero status if it fails to generate the code for any reason.

15.2.2 Trouble-Shooting

SPARCworks/Visual must be connected to an X server to generate code from the command line. Usually command line code generation does not create any visible windows but windows do appear momentarily on the server screen for designs containing certain types of widgets, such as ScrolledList and ScrolledText and when generating Windows code.

If you don't specify a *code_file*, SPARCworks/Visual relies on the filename saved in the design file for the specified type of code. The filename is only saved when you specify it on the Generate Dialog and then save the file. If you have never used the Generate Dialog to generate this type of code from the design file, SPARCworks/Visual produces only an error message.

In all cases, the generate toggles are set as they were last saved in the design file. If you have never generated this type of code from the design file, default toggle settings are used.

15.3 Converting UIL Source to SPARCworks/Visual Save Files

The *uil2xd* filter converts UIL source code to SPARCworks/Visual save files. It reads UIL source from standard input and writes a SPARCworks/Visual save file on standard output.

By default, *uil2xd* generates a SPARCworks/Visual save file for the latest release. The command line synopsis is:

```
uil2xd - tlxywhpsaX -I include_dir
```

The command line options are listed in the following table:

Table 15-2 uil2xd command line options

Switch	Meaning
t	Don't convert ScrolledWindow containing Text to Scrolled Text. By default **uil2xd** converts a ScrolledWindow widget containing a Text widget into a ScrolledText widget. Use the -t flag to preserve the structure.
l	Don't convert ScrolledWindow containing List to Scrolled List. By default **uil2xd** converts a ScrolledWindow widget containing a List widget into a ScrolledList widget. Use the -l flag to preserve the structure.
x	Pass through XmNx resources. By default **uil2xd** does not output absolute positions in the save file. Use the -x flag to pass XmNx resources into the output file.
y	Pass through XmNy resources. By default **uil2xd** does not output absolute positions in the save file. Use the -y flag to pass XmNy resources into the output file.
w	Pass through XmNwidth resources. By default **uil2xd** does not output absolute sizes in the save file. Use the -w flag to pass XmNwidth resources into the output file.
h	Pass through XmNheight resources. By default **uil2xd** does not output absolute sizes in the save file. Use the -h flag to pass XmNheight resources into the output file.
p	Preserve position resources in output file. Same as -x -y
s	Preserve size resources in output file. Same as -w -h

Table 15-2 uil2xd command line options

Switch	Meaning
a	Preserve position and size resources in output file. Same as -p -s
X	Print list of switches.
I include_dir	Adds include_dir to the list of directories that will be searched for include files.

uil2xd does not handle the following constructs:

- String tables containing compound strings
- Color_table
- Icon
- Ascii tables in argument definition
- Integer tables in argument definition
- Imported keyword – this is a fatal error
- Exported keyword
- Private keyword
- Creation procedure
- Default character set clause
- Identifier section

Except for the imported keyword, *uil2xd* simply ignores these constructs.

15.4 Converting GIL Source to SPARCworks/Visual Save Files

The *gil2xd* filter converts SunSoft's DevGuide save files into SPARCworks/Visual save files. The converter works by mapping the OPEN LOOK objects into Motif objects. It reads GIL source from standard input and writes a SPARCworks/Visual save file on standard output.

It generates a save file for the latest version of SPARCworks/Visual. The command line synopsis is:

```
gil2xd -xywhpsaX
```

The command line options are listed in the following table:

Table 15-3 gil2xd command line options

Switch	Meaning
x	Pass through XmNx resources. By default **gil2xd** does not output absolute positions in the save file. Use the -x flag to pass XmNx resources into the output file.
y	Pass through XmNy resources. By default **gil2xd** does not output absolute positions in the save file. Use the -y flag to pass XmNy resources into the output file.
w	Pass through XmNwidth resources. By default **gil2xd** does not output absolute sizes in the save file. Use the -w flag to pass XmNwidth resources into the output file.
h	Pass through XmNheight resources. By default **gil2xd** does not output absolute sizes in the save file. Use the -h flag to pass XmNheight resources into the output file.
p	Preserve position resources in output file. Same as -x -y
s	Preserve position and size resources in output file. Same as -w -h
a	Preserve position and size resources in output file. Same as -p -s
X	Print list of switches

gil2xd does not handle connections other than function calls and the simple notify actions for buttons which can be mapped to links. Other connections are reported as warnings. *gil2xd* simply ignores these constructs.

15.4.1 Mappings

Few of the mappings from OPEN LOOK objects to Motif widgets are straightforward as they depend somewhat on their context. The fundamentals of the mappings are outlined below.

base-window

Maps to a DialogShell with a MainWindow child with a Form work area.

popup-window

Maps to a DialogShell with a Form child.

canvas-pane

Maps to a DrawingArea which will be a child of a ScrolledWindow if horizontal-scrollbar or vertical-scrollbar is true. An associated PopupMenu is created as a child of the DrawingArea.

control-area

Maps to a Form.

menu

Maps to a Menu. If the menu has a menu-title attribute, the first child widget will be a Label which shows the title, followed by a Separator. The menu items are mapped to additional children of the Menu. If the menu-type attribute is command, the widgets will be ToggleButtons; if they have an associated menu they will be CascadeButtons, otherwise they will be PushButtons. As SPARCworks/Visual has no concept of shared menus, menus which are referenced from more than one place will map to copies of the Menu.

message

Maps to a Label.

button

Maps to a PushButton if it does not have a menu, otherwise it maps to a CascadeButton. This CascadeButton will be created in a MenuBar. CascadeButtons which have the same y co-ordinate will be created in the same MenuBar. The MenuBar will be created in an enclosing MainWindow if possible, otherwise it will be created at the appropriate location.

slider and gauge

Both map to a Scale. Separators will be added as children for tick marks and Labels may be added to show the min-value-string and max-value-string. The min-value and max-value map to the Scale's minimum and maximum fields respectively.

setting

Maps to an OptionMenu if setting-type is stack, otherwise maps to a RowColumn. The choices are mapped to PushButtons in an OptionMenu and ToggleButtons in a RowColumn. For exclusive and non-exclusive settings the ToggleButton is adjusted so that the indicator is not used (shadowThickness = 2, marginLeft = 0, indicatorOn = false).

text field

Maps to RowColumn with Label and Text widgets. Text will be ScrolledText if text-type is set to multiline.

list

Maps to a ScrolledList. If the list has a label attribute set, the ScrolledList is created as a child of a RowColumn with a Label child which displays the label. If the list has a title attribute, the ScrolledList is created as a child of a Frame with a Label to display this title.

drop-target

Maps to a Label.

stack

Maps to a Form which has each of the stack members as children. The children are attached to both sides of the Form.

group

Maps to a RowColumn which has each of the member widgets as children.

term-pane and text-pane

Both map to ScrolledText.

15.4.2 Attributes

Once the gil objects have been mapped to widgets the attributes must be mapped to appropriate widget resources. The following resources are always mapped:

Table 15-4 Resources that are Always Mapped

gil	xd	Notes
x	XmNx	
y	XmNy	
width	XmNwidth	
height	XmNheight	
foreground-color	XmNforeground	
background-color	XmNbackground	
initial-state	XmNsensitive	inactive - sensitive = false invisible - managed = false

The width and height resources are only used if the -w or -h flags are set when gil2xd is run. The x and y resources will be output if the -x or -y flags are set. However, for widgets which are children of Forms the x and y co-ordinates will be used to calculate default Form attachments to preserve the approximate layout.

Note that many of the Motif manager widgets will ignore explicit x, y, width and height resources anyway. The gil2xd filter can be used without any of the runtime flags to produce an adequate layout which can be easily modified using SPARCworks/Visual.

Other resources are mapped to the nearest possible resource.

Table 15-5 Resources that are Mapped to the Nearest Possible Resource

gil	xd	Notes
columns	XmNcolumns	
constant-width	XmNrecomputeSize	
group-type	XmNorientation	Sets XmNnumColumns, XmNorientation and XmNpacking to reproduce a similar layout of group

Table 15-5 Resources that are Mapped to the Nearest Possible Resource

gil	xd	Notes
icon-file	XmNiconPixmap	
icon-label	XmNiconName	
icon-mask	XmNiconMask	
initial-state	XmNinitialState	DialogShell only
initial-value	XmNvalue	
label	XmNlabelString	If matching label-type attribute is glyph then label is mapped to labelPixmap
label	XmNtitle	For shells
label	XmNtitleString	For gauge
label-type	XmNlabelType	
layout-type	XmNorientation	
max-value	XmNmaximum	
menu-type	XmNradioBehavior	If exclusive, XmNradioBehavior = true
min-value	XmNminimum	
multiple-selections	XmNselectionPolicy	If set, XmNselectionPolicy = MULTIPLE_SELECT
orientation	XmNorientation	
pinnable	XmNtearOffModel	If pinnable, XmNtearOffModel = TEAR_OFF_ENABLED
read-only	XmNeditable	
resizeable	XmNallowResize	
rows	XmNnumColumns	Sets XmNnumColumns, XmNorientation and XmNpacking to reproduce a similar layout of settings
rows	XmNrows	For text widget
rows	XmNvisibleItemCount	For list widget
selection-required	XmNradioAlwaysOne	

Table 15-5 Resources that are Mapped to the Nearest Possible Resource

gil	xd	Notes
show-border	XmNshadowThickness	If set sets XmNshadowThickness to 1 for forms which are not children of a Shell
show-value	XmNshowValue	
slider-width	XmNscaleWidth	Sets XmNscaleWidth or XmNscaleHeight depending on orientation
stored-length	XmNmaxLength	
text-initial-value	XmNvalue	
text-type	XmNeditMode	If multiline, XmNscrollVertical = false, rows maps to XmNrows
title	XmNlabelString	
value-length	XmNcolumns	

Actions which have a CallFunction function_type are mapped to callbacks where appropriate.

Table 15-6 Actions that are Mapped to Callbacks

Action	Callback	Widget
Create	XmNcreateCallback	Any
Destroy	XmNdestroyCallback	Any
Notify	XmNactivateCallback	PushButton
select	XmNinputCallback	DrawingArea
adjust	XmNinputCallback	DrawingArea
DoubleClick	XmNinputCallback	DrawingArea
Repaint	XmNexposeCallback	DrawingArea
Resize	XmNresizeCallback	DrawingArea
Select	XmNvalueChangedCallback	Gauge
Adjust	XmNdragCallback	Gauge
Notify	XmNvalueChangedCallback	Gauge
Popup	XmNmapCallback	Menu
Popdown	XmNunmapCallback	Menu

Table 15-6 Actions that are Mapped to Callbacks

Action	Callback	Widget
Notify	XmNentryCallback	Menu
Notify	XmNvalueChangedCallback	ToggleButton
Unselect	XmNvalueChangedCallback	ToggleButton
Popup	XmNpopupCallback	Shell
Popdown	XmNpopdownCallback	Shell
Notify	XmNactivateCallback	Text
KeyPress	XmNvalueChangedCallback	Text
Notify	XmNentryCallback	RowColumn
Done	XmNunmapCallback	Form
Notify	XmNbrowseSelectionCallback	List

There are a number of other gil actions which are not detailed in this list. These are not supported by the filter as there is no appropriate Motif callback.

Notify actions for PushButtons which have a Show, Hide, Enable or Disable connection are mapped to the appropriate SPARCworks/Visual link.

Makefile Generation 16

16.1 Introduction

This section describes SPARCworks/Visual's Makefile generation facilities. SPARCworks/Visual can create two types of Makefile: simple or with templates. The simple Makefile only contains the build rules for the local *.xd* file and so is only useful for applications that are contained in a single file. Makefiles with templates contain the build rules plus structured comments that serve as templates for updating. A Makefile with templates can be updated to add files without overwriting your previous work. Unlike most generated files, Makefiles with templates can be edited and regenerated without losing your work.

This section gives step-by-step instructions for creating a Makefile with templates and then updating the Makefile when a second design file is added to the application.

16.2 Creating the Initial Makefile

The first step is to create a design, generate the C code for it and then generate an initial Makefile.

1. **Create a new directory *myapp*. Make this your current directory and start SPARCworks/Visual.**

2. **Build the widget hierarchy shown in Figure 16-1:**

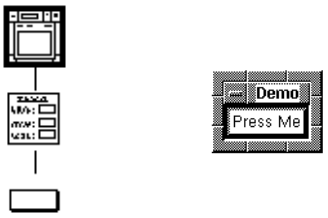


Figure 16-1 Main Program Widget Hierarchy

3. **Give the PushButton an Activate callback, *button_pressed*.**

4. **Use the Shell resource panel to designate the Shell widget as an Application Shell.**

Before you generate a Makefile, you must generate the code files that you want to include in it. Generating the code files sets the names for these files in the design file. Until you do this, the Makefile generation feature doesn't know the names of these files and can't add them to the Makefile.

5. **Generate C to file *myapp.c*. Set toggles to generate includes and main, but not links.**

6. **Generate C stubs to *app_stubs.c*. Set toggles to generate includes, but not links.**

Now you can generate a Makefile that compiles and links *myapp.c* and *app_stubs.c*:

7. **Generate the Makefile to *Makefile*. Set both "New makefile" and "Makefile template" toggles on, as shown in Figure 16-2.**

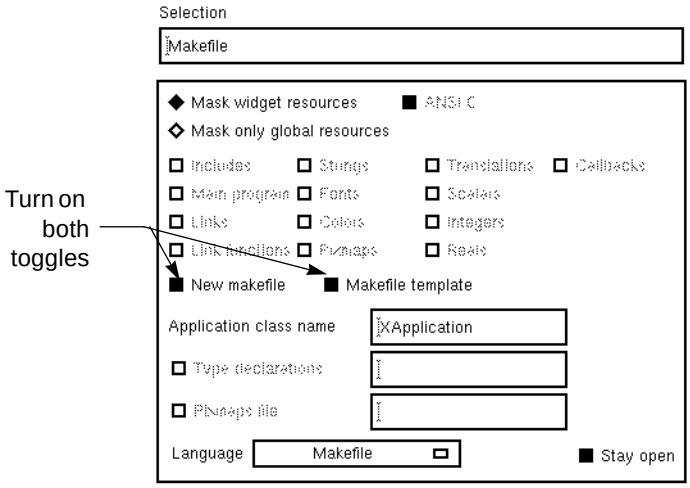


Figure 16-2 Initial Makefile Generation.

The generated Makefile contains the required make rules and template lines for further amendment. Ignoring the template lines, the generated Makefile contains the following rules:

```
XD_C_PROGRAMS=\
        myapp
XD_C_PROGRAM_OBJECTS=\
        myapp.o
XD_C_PROGRAM_SOURCES=\
        myapp.c
XD_C_STUB_OBJECTS=\
        app_stubs.o
XD_C_STUB_SOURCES=\
        app_stubs.c
myapp: myapp.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS)
    $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp myapp.o
       $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)
myapp.o: myapp.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c myapp.c
```

```
app_stubs.o: app_stubs.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c app_stubs.c
```

8. Save the current design to *myapp.xd*.

16.3 Updating the Initial Makefile

Once you have generated the initial Makefile, you can update it to reflect additional work. To demonstrate this, use the following instructions to build a popup dialog in another file, generate code for it and then update the Makefile to reflect the new modules.

1. Select "New" from the File menu to start a new design.

2. Build the hierarchy shown in Figure 16-3.

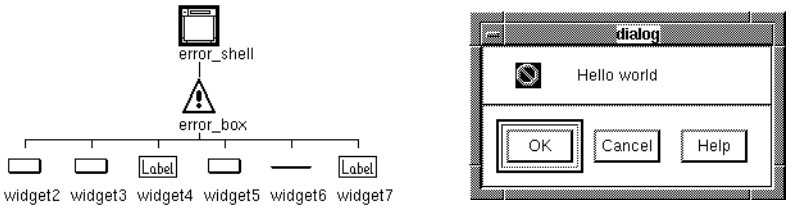


Figure 16-3 Secondary Popup Dialog.

3. Name the Shell and MessageBox *error_shell* and *error_box* respectively.

4. Give the MessageBox a Cancel callback, *cancel_error*.

5. Generate C externs to *error.h*.

6. Generate C to file *error.c*. Select the Type declarations toggle to include *error.h*. Set toggles to generate includes but not main or links.

7. Generate C stubs to *error_stubs.c*. Set toggles to generate includes, but not links.

8. Generate Makefile to *Makefile*. Turn off the "New makefile" toggle, leaving the "Makefile template" toggle set, as shown in Figure 16-4.

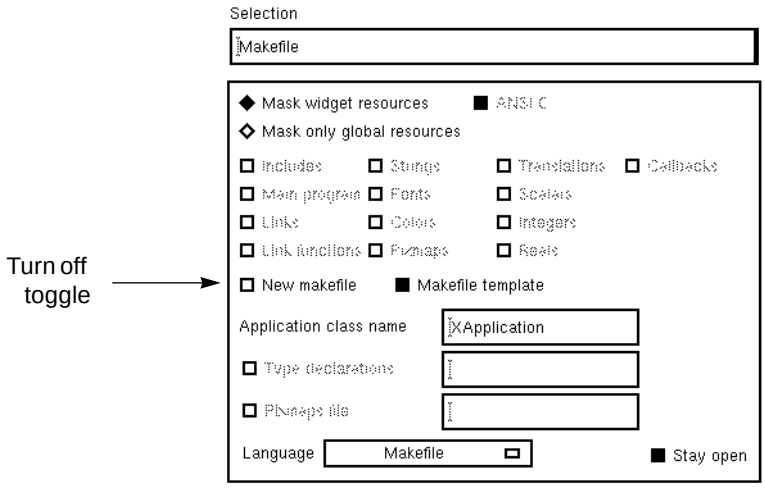


Figure 16-4 Updating Makefile.

The generated Makefile is updated with the new modules.

```
XD_C_PROGRAMS=\
       myapp
XD_C_PROGRAM_OBJECTS=\
       myapp.o
XD_C_PROGRAM_SOURCES=\
       myapp.c
XD_C_OBJECTS=\
       error.o
XD_C_SOURCES=\
       error.c
XD_C_STUB_OBJECTS=\
       error_stubs.o\
       app_stubs.o
XD_C_STUB_SOURCES=\
       error_stubs.c\
       app_stubs.c
```

```
myapp: myapp.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS)
       $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp\
       myapp.o $(XD_C_OBJECTS)\
       $(XD_C_STUB_OBJECTS) $(MOTIFLIBS)\
       $(LDLIBS)

myapp.o: myapp.c
       $(CC) $(CFLAGS) $(CPPFLAGS) -c myapp.c

error.o: error.c
       $(CC) $(CFLAGS) $(CPPFLAGS) -c error.c

app_stubs.o: app_stubs.c
       $(CC) $(CFLAGS) $(CPPFLAGS) -c app_stubs.c

error_stubs.o: error_stubs.c
       $(CC) $(CFLAGS) $(CPPFLAGS) -c error_stubs.c
```

16.3.1 Building the Application

SPARCworks/Visual has generated the code files for two dialogs, two stub files and a Makefile. All that remains is to fill in the two stubs and build the application.

1. Edit *app_stubs.c* and make the following changes.

At the end of the generated list of includes, add the following line:

```
#include <error.h>
```

Add functionality to the *button_pressed* callback stub. This callback pops up the error dialog when the button is pressed.

```
void
button_pressed (Widget w, XtPointer client_data, XtPointer call_data
       )
{
    if ( error_shell == NULL )
        create_error_shell (XtParent (XtParent (w) ) );
    XtManageChild ( error_box );
}
```

2. Edit *error_stubs.c* and make the following changes.

Add functionality to the *cancel_error* callback. This function pops down the error dialog when the user presses the “Cancel” button.

```
cancel_error (Widget w, XtPointer client_data, XtPointer call_data )
{
    XtUnmanageChild ( error_box );
}
```

3. **Save the current design to *error.xd*.**
4. **Read the file *myapp.xd* into the current design.**
5. **Generate resources into the file *myapp.res*.**
6. **To access these resources do the following:**
7. **If you are using a C shell enter on the command line:**

```
setenv XENVIRONMENT myapp.res
```

8. **If you are using a Bourne shell or ksh, enter on the command line:**

```
XENVIRONMENT=myapp.res export XENVIRONMENT
```

9. **Set VISUROOT to the path of the SPARCworks/Visual installation directory.**

   This must be done as the Makefile includes files and libraries relative to the SPARCworks/Visual installation directory.

10. **Build the application. On the command line, type:**

```
make
```

    The following messages are displayed as the application builds:

```
cc -I. -I$VISUROOT/xpm -c myapp.c
cc -I. -I$VISUROOT/xpm -c error.c
cc -I. -I$VISUROOT/xpm -c error_stubs.c
cc -I. -I$VISUROOT/xpm -c app_stubs.c
cc -I. -I$VISUROOT/xpm -L$VISUROOT/xpm -o myapp myapp.o error.o
       error_stubs.o app_stubs.o -lXpm -lXm -lXt -lX11
```

    where VISUROOT is set to the path of the SPARCworks/Visual installation directory.

11. **To run the application, type:**

```
myapp
```

16

16.4 Editing the Generated Makefile

You can edit and regenerate the Makefile without losing information. You can make the most commonly needed changes by editing the Makefile flags at the beginning of the file. For example, to make the compiler search the *../hdrs* directory for header files, append the entry:

```
-I../hdrs
```

to the end of the CFLAGS line in the Makefile.

The new setting of the *CFLAGS* variable applies to all application build commands:

```
cc -I. -I$VISUROOT/xpm -I../hdrs -c my_app.c
cc -I. -I$VISUROOT/xpm -I../hdrs -c error.c
cc -I. -I$VISUROOT/xpm -I../hdrs -c error_stubs.c
cc -I. -I$VISUROOT/xpm -I../hdrs -c app_stubs.c
cc -I. -I$VISUROOT/xpm -I../hdrs -L$VISUROOT/xpm -o my_app my_app.o
       error.o error_stubs.o app_stubs.o -lXpm -lXm -lXt -lX11
```

where VISUROOT is set to the path of the SPARCworks/Visual installation directory.

The change to *CFLAGS* is retained when you regenerate the Makefile with the "New makefile" toggle off. It is only lost if you generate a new Makefile.

16.4.1 Editing Template Lines

A large part of the generated Makefile consists of *template lines*. Template lines are comments that control the form of new template instances that are generated. Template lines have a #SPARCworks/Visual: prefix. For example, the following template lines tell SPARCworks/Visual how to generate the Makefile lines that produce a C object file from a C source file (*XDG_C_SOURCE*):

```
#SPARCworks/Visual:XDG_C_OBJECT: XDG_C_SOURCE
#SPARCworks/Visual:    $(CC) $(CFLAGS) $(CPPFLAGS) -c XDG_C_SOURCE
```

Each time you update the Makefile to add a file to your application, SPARCworks/Visual generates a *template instance* for each relevant template. These instances contain the actual build commands for your application.

Template instances are marked with a “DO NOT EDIT” comment at the beginning and at the end. A typical instance of the template shown above looks like:

```
#DO NOT EDIT >>>
error.o: error.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c error.c
#<<< DO NOT EDIT
```

Template instances should not be edited because your edits may be lost the next time you generate the Makefile. Instead, to change the build commands, edit the corresponding template lines. After you edit a template line, delete any instances of that template line that already exist in the Makefile. The instances are found just after the template line.

For example, to build all C files for debugging, you would:

1. **Change the template line:**

```
#SPARCworks/Visual: $(CC) $(CFLAGS) $(CPPFLAGS) -c XDG_C_SOURCE
```

to

```
#SPARCworks/Visual: $(CC) $(CFLAGS) $(CPPFLAGS) -g -c XDG_C_SOURCE
```

2. **Remove the instances following the template line.**
3. **Regenerate the Makefile, with the “New” toggle off, for each design in the application.**

This procedure generates new instances using the modified template.

16.4.2 Template Configuration

The original templates are specified by the file pointed to by the following resources:

```
visu.motifMakeTemplateFile: $VISUROOT/make_templates/motif
visu.mmfcMakeTemplateFile: $VISUROOT/make_templates/mfc
```

There are two resources so that you can have different templates customized to pick up the appropriate class libraries. The value for the resource can contain environment variables which will be expanded by /bin/sh.

If SPARCworks/Visual cannot find the file specified, it will fallback to the template specified in the SPARCworks/Visual resource file, using the *makefileTemplate* application resource. To make a change to the template apply globally to all new Makefiles, edit the resource file. For details, see the *Configuration* chapter.

SPARCworks/Visual refers to the template only when you generate a new Makefile. To change templates in an existing Makefile, edit the file by hand as described in the previous section, or delete the file and start over.

16.4.3 Dependency Information

SPARCworks/Visual does not generate dependency information into the Makefile. The default template includes a "depend" target that can be invoked using:

```
make depend
```

This operation invokes the *makedepend* utility, which scans the existing makefile and appends a standard dependency list. Use *man makedepend* for more information.

16.5 Using your own Makefiles

The $VISUROOT/*make_templates* directory contains the template(s) used to generate makefiles. The include directories and libraries you will need to include in your makefile are listed in this file. Those not required by your system are commented out. A typical example is shown below:

```
solaris 2.x
#XINCLUDES=-I/opt/SUNWmotif/include -I/usr/dt/include
#XLIBS=-L/opt/SUNWmotif/lib -L/usr/dt/lib -L/usr/openwin/lib
...
```

XINCLUDES and *XLIBS* specify the extensions to the included directory and library path respectively.

The *motif* template is used for makefiles generated in non-Windows mode and the *mfc* template is used for generating makefiles for Motif MFC in Windows mode.

Configuration 17

17.1 Introduction

There are several ways to customize the SPARCworks/Visual interface, using either the resource file or *visu_config*. This section explains the main features that can be customized via the resource file. These features are:

- Palette icons
- Palette contents and layout
- Toolbar
- Makefile templates

For further information on SPARCworks/Visual resources, see the *Application Defaults* Appendix. For information on using *visu_config*, see Chapter 14.

17.2 Palette Icons

SPARCworks/Visual has an icon for each widget class. The icon is drawn on the palette buttons and *displayed* in the tree hierarchy. The icon can be either a full color XPM format pixmap or a monochrome bitmap. On start-up, SPARCworks/Visual *searches* through the application resources to find a pixmap or bitmap file for each icon. If one cannot be found, a built-in bitmap is used.

17.2.1 Specifying the Icon File

Each Motif widget has an application resource that specifies its icon file. *$*VISUROOT*/app-defaults/*visu contains a complete list of these resource names. For example, the resource for the ArrowButton icon is:

```
visu.arrowButtonPixmap: arrow.xpm
```

SPARCworks/Visual searches in the same way as *XmGetPixmap()* to find the file. This search path is complex; for details, refer to the Motif documentation. In practice, SPARCworks/Visual places the default pixmap files in *$*VISUROOT*/bitmaps* and adds *$*VISUROOT*/bitmaps/%B* to the XBMLANGPATH environment variable. Individual users can provide their own local pixmaps by creating a file of the correct name, such as *arrow.xpm*, in a directory and adding that directory with the file matching string "*/%B*" to the SW_VISU_XBMLANGPATH environment variable. For example,

```
setenv SW_VISU_XBMLANGPATH /home/me/pix/%B
```

17.2.2 Default Pixmaps

SPARCworks/Visual comes with two sets of icon pixmaps. The default set is located in *$*VISUROOT*/bitmaps*. These icons are drawn using minimal color and will work on either color or monochrome screens.

A set of color pixmaps is located in *$*VISUROOT*/color_icons*. To change to the color pixmaps, either set the environment variable SW_VISU_ICONS to *color_icons*, or add *$*VISUROOT*/color_icons/%B* to the SW_VISU_XBMLANGPATH environment variable. To revert to the default icons either unset SW_VISU_ICONS or set it to *bitmaps*.

Alternatively, an individual user can specify a different file name by setting the resource in a local copy of the SPARCworks/Visual resource file:

```
visu.arrowButtonPixmap: my_arrow.xpm
```

or:

```
visu.arrowButtonPixmap: /home/me/visu_bitmaps/my_arrow.xpm
```

17.2.3 Pixmap Requirements

You can create your own color pixmaps for icons using XPM3 format. This format can be created using the SPARCworks/Visual *pixmap* editor. Icon pixmaps can be any size; the palette and tree are displayed with a vertical spacing to accommodate the tallest icon. The SPARCworks/Visual display looks best when all icon pixmaps are the same size. Default sizes are 32 by 32 for the large-screen icon pixmap and 20 by 20 for the small-screen version.

When printed from SPARCworks/Visual, the icons are dithered to grey scales.

17.2.4 Transparent Area

The icon should contain an area of transparency. SPARCworks/Visual uses this area to display highlighting and structure colors in the tree and the background color on the palette button. XPM supports transparency by means of the color name “none” (not the color object “none”).

To create a transparent area in the SPARCworks/Visual pixmap editor, type “none” in the color selector and apply it to a color button in the pixmap editor. Use this button to draw your transparent areas. The pixmap editor displays transparent areas as white.

Transparency does not work for other Motif pixmaps without additional application coding.

17.2.5 Icons for User-Defined Widgets

visu_config’s Widget Edit dialog lets you specify icons for user-defined widgets. For more information, see the *Widget Attributes* section of the *User-Defined Widgets* chapter on page 328.

17.2.6 Icons for Palette Definitions

The palette definition dialog lets you specify an icon for each palette definition and a file name to be used as a fallback if the resource is not set. The pixmap file is searched for in the same way as the default pixmaps. If SPARCworks/Visual cannot find the pixmap it will use the default pixmap for the widget that is at the root of the definition.

17.3 Palette Contents

You can stop certain widget classes from appearing on the SPARCworks/Visual palette using either the *stopList* resource or *visu_config*. Stopped widgets do not appear on the palette and so they cannot be created interactively. They can be read in from a SPARCworks/Visual save file but are not selectable.

To stop a widget class, specify the class name in the *stopList* resource. For example, to remove the Motif PanedWindow and ArrowButton from your widget palette, set the resource:

```
visu.stopList: XmPanedWindow,XmArrowButton
```

To stop a user-defined widget, specify the class name:

```
visu.stopList: boxWidgetClass,formWidgetClass
```

visu_config has a Stopped Motif Widgets dialog which contains a toggle for each of the Motif widgets. For more information, see the *User-Defined Widgets* chapter. Widgets that are removed from the palette in *visu_config* cannot be added back in using SPARCworks/Visual resources.

17.4 Palette Layout

The default settings display a vertical palette containing three columns of widget icons and attached to the main window. You can change the default layout using the resource file. You can also change the palette layout at run time using the Palette Menu.

17.4.1 Separate Palette

You can display the palette in a separate window. To separate the palette at run time, use the “Separate Palette” option in the Palette Menu. To have a separate palette by default, set the following resource:

```
visu*pm_separate.set:true
```

When you separate the palette in the resource file, you must also explicitly set the default height of the SPARCworks/Visual main window:

```
visu.main_window.height: 600
```

The resource file contains several examples of alternative layouts, such as:

```
! Two columns is good if you do not have user defined widgets
visu*icon_panel.composite_icons.numColumns: 2
visu*icon_panel.basic_icons.numColumns: 2

! Set both labels and icons on
visu*pm_both.set: true

! You might also want to make the tool a bit wider
visu.main_window.width: 750

! Four columns, with labels underneath
visu*icon_panel.composite_icons.XmRowColumn.\
       orientation: VERTICAL
visu*icon_panel.composite_icons.numColumns: 4
visu*icon_panel.basic_icons.XmRowColumn.\
       orientation: VERTICAL
visu*icon_panel.basic_icons.numColumns: 4
visu*icon_panel*xwidget_icons.XmRowColumn.\
       orientation: VERTICAL
visu*icon_panel*xwidget_icons.numColumns: 4
visu*icon_panel*defn_icons.XmRowColumn.\
       orientation: VERTICAL
visu*icon_panel*defn_icons.numColumns: 4
```

17.5 Toolbar

The SPARCworks/Visual interface includes a toolbar. Buttons in the toolbar correspond to buttons in the menus. Generally, when you select a toolbar button, it does exactly the same thing as the corresponding menu button. The code generation buttons are an exception. When you select code generation from the menu bar, SPARCworks/Visual displays the Generate Dialog and generates the file only after you click on “Apply”. However, when you select a code generation button in the toolbar, SPARCworks/Visual generates the file immediately, using your last specifications, without first showing the dialog. The dialog appears only if you have never generated that type of file from the current design.

17.5.1 Configuring the Toolbar

To configure buttons into the toolbar, use the *toolbarDescription* resource. This should contain a comma-separated list of the widget names of the buttons. It can also contain the word *separator* to add extra space between items and the word *flavor* to insert the Windows flavor option menu.

To obtain the widget name of a button, search the visu.*defaults* file for the entry that sets the *labelString* resource on the button in the menu bar. For example, visu.*defaults* contains this line:

```
visu*em_cut.labelString: Cut
```

em_cut is the widget name of the “Cut” button in the Edit Menu.

The following line produces a toolbar with “Cut”, “Copy”, “Paste”, “Core resources...”, “Layout...” (layout editor), and “C” (code generation) buttons:

```
visu.toolbarDescription:separator,em_cut,\
       em_copy,em_paste,separator,wm_prim,\
       wm_layout,separator,gm_c
```

17.5.2 Labels for Toolbar Buttons

The toolbar buttons have the same widget name as the counterpart menu button and so, by default (assuming that no pixmaps are configured), they appear with the same words. For example, the resource file contains the line:

```
visu*gm_c.labelString: C...
```

By default, this applies to both the menu and the toolbar. You might want to remove the ellipsis for the toolbar version since code generation buttons in the toolbar do not display a dialog. To do this, add the following line:

```
visu*toolbar.gm_c.labelString: C
```

17.5.3 Pixmaps for Toolbar Buttons

The toolbar buttons also have a string resource associated with them that specifies an XPM pixmap or X11 bitmap file in exactly the same way as for the palette buttons.

```
visu*toolbar.em_copy_file.toolbarPixmap:\ em_copy_file
```

These pixmaps can be overridden by changing the resource or by providing a file earlier in the search path.

17.6 Makefile Features

This section describes resources that control Makefile generation. For an introduction to this feature, see the *Makefile Generation* chapter.

17.6.1 File Suffixes

Object and executable file names are derived from source file names by suffix substitution. The suffixes are specified by the following application resources:

```
visu.objectFileSuffix: .o
visu.executableFileSuffix:
visu.uidFileSuffix: .uid
```

17.6.2 Makefile Template

The template used for generating new Makefiles is defined in two ways: either by filename or directly in the resource file. The second way is used as a fallback if the file cannot be found.

The template file is specified by one of two resources:

```
visu.motifMakeTemplateFile: $VISUROOT/make_templates/motif[1]
visu.mmfcMakeTemplateFile: $VISUROOT/make_templates/mfc
```

There are two resources so that you can have different templates customized to pick up the appropriate class libraries. The value for the resource can contain environment variables which will be expanded by /bin/sh.

The fallback template is specified by the *makefileTemplate* resource:

```
visu.makefileTemplate:\
# System configuration\n\
# ---------------------\n\
\n\
# everything is in /usr/include or /usr/lib\n\
XINCLUDES=\n\
XLIBS=\n\
```

1. $VISUROOT is the path to the root of the SPARCworks/Visual installation directory.

```
LDLIBS=\n\
.
.
.
```

You can edit the Makefile variables set at the beginning of the template to reflect your system configuration. For example, you can set the *XINCLUDES* variable to the path for your X include files.

17.6.3 Template Protocol

This section explains the symbols used in the Makefile template. In general we recommend that you not edit the default template except for the variables at the beginning. If you want to edit the actual template lines, first read the *Makefile Generation* chapter and try to get the results you want by setting a variable.

Lines in the Makefile template that begin with #SPARCworks/Visual: are template lines. When SPARCworks/Visual generates or updates a Makefile, it creates instances of the appropriate template lines for each design file based on the files you have generated and converts the special symbols beginning with an *XDG_* prefix (SPARCworks/Visual generated) to file names. *XDG_* symbols convert to a single file, or a list of files if the symbol name ends with the *_LIST* suffix.

List symbols are used singly and not mixed with ordinary symbols in lines such as the following:

```
#SPARCworks/Visual:XD_C_PROGRAMS=XDG_C_PROGRAM_LIST
```

The *XDG_C_PROGRAM_LIST* symbol translates to a list of all executables that the Makefile can build. A typical instance of this template line is:

```
#DO NOT EDIT >>>
XD_C_PROGRAMS=\
       myapp1\
       myapp2
#<<< DO NOT EDIT
```

Ordinary template symbols, without the *_LIST* suffix, represent single files. You can combine any number of ordinary template symbols in lines such as:

```
#SPARCworks/Visual:XDG_C_PROGRAM: XDG_C_PROGRAM_OBJECT
      $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS)
#SPARCworks/Visual: $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o
      XDG_C_PROGRAM XDG_C_PROGRAM_OBJECT $(XD_C_OBJECTS)
      $(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)
```

When SPARCworks/Visual generates the Makefile, it adds a separate instance of these lines for each design file for which you have generated code with the "Main program" toggle set. Other files in the application are linked in as *XD_C_OBJECTS* and *XD_C_STUB_OBJECTS*.

```
#DO NOT EDIT >>>
myapp1: myapp1.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(CC)
      $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp1 myapp1.o
$(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)
#<<< DO NOT EDIT
#DO NOT EDIT >>>
myapp2: myapp2.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(CC)
      $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp2 myapp2.o
$(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)
#<<< DO NOT EDIT
```

The following table shows the ordinary template symbols. You can add a *_LIST* suffix to any of these symbols to generate the corresponding list symbol.

Table 17-1 Makefile Template Symbols

Name	Use
XDG_C_PROGRAM_SOURCE	C source with main program (*main.c*)
XDG_C_PROGRAM_OBJECT	Corresponding object (*main.o*)
XDG_C_PROGRAM	Corresponding executable (*main*)
XDG_C_SOURCE	C source (*foo.c*)
XDG_C_OBJECT	Corresponding object (*foo.o*)
XDG_C_STUB_SOURCE	C stubs (*stubs.c*)
XDG_C_STUB_OBJECT	Corresponding object (*stubs.o*)
XDG_C_EXTERN	C header (*externs.h*)
XDG_C_PIXMAP	C pixmaps (*pixmaps.h*)
XDG_CC_PROGRAM_SOURCE	C++ source with main program (*main.C*)
XDG_CC_PROGRAM_OBJECT	Corresponding object (*main.o*)

Table 17-1 Makefile Template Symbols

Name	Use
XDG_CC_PROGRAM	Corresponding executable (*main*)
XDG_CC_SOURCE	C++ source (*foo.C*)
XDG_CC_OBJECT	Corresponding object (*foo.o*)
XDG_CC_STUB_SOURCE	C++ stubs (*stubs.C*)
XDG_CC_STUB_OBJECT	Corresponding object (*stubs.o*)
XDG_CC_EXTERN	C++ header (*externs.h*)
XDG_CC_PIXMAP	C++ pixmaps (*pixmaps.h*)
XDG_UIL_SOURCE	UIL source (*foo.uil*)
XDG_UIL_OBJECT	Corresponding compiled UIL (*foo.uid*)
XDG_C_FOR_UIL_PROGRAM_SOURCE	C for UIL source with main program (*main.c*)
XDG_C_FOR_UIL_PROGRAM_OBJECT	Corresponding object (*main.o*)
XDG_C_FOR_UIL_PROGRAM	Corresponding executable (*main*)
XDG_C_FOR_UIL_SOURCE	C for UIL source (*foo.c*)
XDG_C_FOR_UIL_OBJECT	Corresponding object (*foo.o*)
XDG_C_FOR_UIL_EXTERN	C for UIL header (*externs.h*)
XDG_UIL_PIXMAP	UIL pixmaps (*pixmaps.uil*)
XDG_X_RESOURCE_FILE	X resource file (*foo.res*)

The Compound String Editor 18

18.1 Introduction

The labels in the tutorial example all use simple text strings. This chapter describes the SPARCworks/Visual compound string editor, which uses an internal structure to let you create more complex strings. In conjunction with complex font objects, these strings let you display labels that use more than one font, or labels that are written entirely or partly from right to left.

18.2 Compound Strings and Font Objects

So far, you have learned how to select a font and apply it to a widget. You have also created a simple font object that corresponds to a single font. *Complex font objects* let you produce more complex visual effects. A complex font object corresponds to a list of fonts and you can arrange for different parts of a string to be displayed using different fonts from the list.

18.2.1 Compound Strings and Character Strings

To most people a string is just an ordered list of ASCII characters - a character string. Most Motif resources that have string values use a different string representation: the *compound string*. Motif compound strings are used for all string resource values except for the strings in Text and TextField, which are ordinary character strings. (It is important not to confuse the Motif compound string with the compound text format of X. The Motif compound string is often called an *XmString* because this is the naming convention for the Motif toolkit functions used to manipulate it.)

A compound string is a way of encoding text so that it can be displayed in multiple languages and fonts without changing the code. In this section, you will learn how to create a string to be displayed in multiple fonts. For information about using multiple languages, see the *Internationalization* chapter and the Motif documentation.

Conceptually, a compound string consists of four types of component:

- A text string (a string of bytes)
- A fontlist tag. Fontlist tags were previously called “charsets” and this term is still used in many Motif documents
- A direction indicator: right-to-left or left-to-right
- A newline separator

Although these types of component can be in any order, it is common for each text string component to be preceded by a fontlist tag component.

A compound string can be used for the label of a widget by specifying a *fontlist* for the widget’s font resource. A fontlist is a set of (font, tag) pairs. The fontlist tags indicate which font in the fontlist to use for each text string component.

To familiarize yourself with the features of SPARCworks/Visual’s compound string editor, use the following step-by-step example while running SPARCworks/Visual at your work station. This example is separate from the main SPARCworks/Visual tutorial.

♦ **Select “New” from the File Menu.**

The object of the exercise is to reproduce the masthead of the London Guardian on a Label widget, shown in Figure 18-1.

Figure 18-1 Final Appearance of the Text String

18.2.2 Creating a Fontlist Font Object

Create a widget that uses a label and give it a font object with more than one font in its list.

1. **Create a widget hierarchy with a Label as the child of a Form.**
2. **Click twice on the Label widget to display the resource panel.**
3. **Click on the "Font" resource button to display the font selector.**
4. **Use the pulldown menus to select a 24-point medium italic Times font.**
5. **In the "Font object" field, type:** `masthead`
6. **In the "Fontlist tag" field to the right of the "Font object" field, type:** `italic`
7. **Click on "Bind".**

This creates a font object called *masthead* with one font in its list. The font is tagged *italic*.

Note – Tag names are arbitrary. However, several pre-defined tags are listed by the "Default" button and the "UIL" pulldown menu. The "Default" and "UIL" buttons are above the "Fontlist tag" field. Selecting the "Default" option produces the tag *XmFONTLIST_DEFAULT_TAG*. The UIL menu lists tags that can be used in UIL (Motif's User Interface Language). If you are a UIL user, note that most UIL compilers produce a "severe internal error" if you use a tag that is not on this list. If you are not a UIL user, ignore the UIL options.

Next, add another font to the font object. The second font has a different tag.

8. **Select a 24-point bold regular Helvetica font.**
9. **In the "Fontlist tag" field, type:** `bold`

Do not change the *masthead* font object name.

10. Click on “Bind”.

This adds the bold font to the font object list and tags it *bold*. You can see samples of the two fonts in the “Sample” field by selecting the different tags in the “Fontlist tag” list.

You now have a font object that can be used to display a string using two different fonts. Apply it to the Label widget:

11. Click on “Apply” in the font selector.

12. Click on “Apply” in the Label resource panel.

The text in the Label is displayed in the italic Times font because it is the first in the list. To display a text string that uses both fonts, you must create a compound string.

18.2.3 Creating A Compound String

When you type text into the “Label” text field of the resource panel, SPARCworks/Visual creates a compound string which only contains text string components and separator components, which correspond to the newlines. To create a compound string that includes fontlist tags, you must use the compound string editor.

1. Click on the “Label” button in the resource panel.

This displays the compound string editor, shown in Figure 18-2.

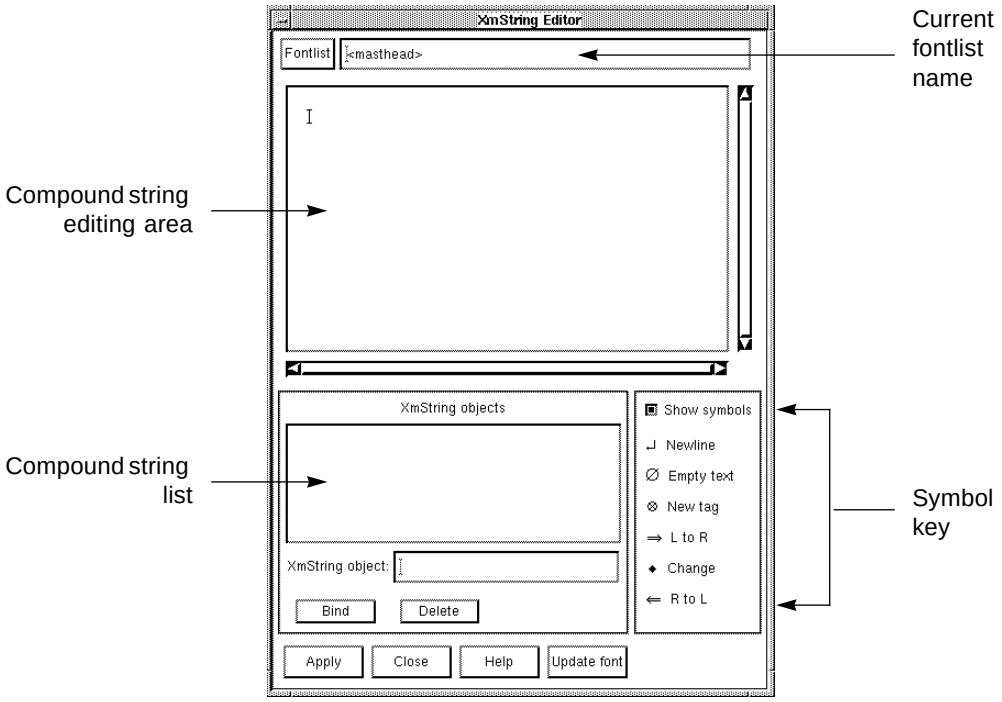


Figure 18-2 Compound String Editor

The compound string editor includes the current fontlist name, an editing area and a list of existing compound string objects. The fontlist name is currently *masthead*, to correspond with the name of the font object used in the widget.

As you enter text in the compound string, it appears in the editing area. The fontlist named at the top of the screen is used to display the text. Other components, such as empty text components and directional indicators, appear as symbols. You can turn off the symbol display using the “Show symbols” toggle to see how the text will look in the widget.

If the “Show symbols” toggle is not on:

2. Click on the “Show symbols” toggle.

To create the compound string, start by entering the text:

3. Click on the I-beam cursor in the editing area.

This makes the cursor blink, indicating that the editor is ready for text entry.

4. **Type the following string with no space between the words:** `TheGuardian`

The text is displayed using the first font in the fontlist, which is the italic Times font, as shown in Figure 18-3.

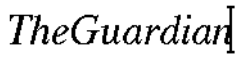


Figure 18-3 Initial Text for the Compound String

To make different parts of the text display in different fonts, you must insert fontlist tags into the compound string.

5. **Position the pointer between *The* and *Guardian*, then press and hold mouse button 3.**

A popup menu appears. Selecting an item from this menu inserts a component of the corresponding type into the string. Selecting “Delete” deletes the current component. To display the word *Guardian* in the bold Helvetica font, you must insert the appropriate fontlist tag (*bold*) at the pointer.

6. **Pull right for the Fontlist tags Menu and select “bold”.**

This inserts the fontlist tag symbol and changes the display as shown in Figure 18-4.

Figure 18-4 Compound String with Fontlist Tag

If you accidentally insert the tag in the wrong position, you can pick it up and move it using mouse button 1.

While the word *The* is now correctly displayed in the italic font, this is only because Motif uses the first font in the fontlist by default. To make the compound string complete you should insert the *italic* tag at the beginning of the string.

7. **Position the pointer before *The*, then press and hold mouse button 3.**

8. Pull right and select “italic”.

This inserts a second fontlist tag symbol, as shown in Figure 18-5.

Figure 18-5 Compound String with Fontlist Tags

The compound string is now complete. To see it as it will appear on the label:

9. Click on the “Show symbols” toggle to turn off the symbols.

This redraws the string without the fontlist tag symbols, as shown in Figure 18-6.

Figure 18-6 Compound String Without Symbols

18.2.4 Creating a Compound String Object

Now you can create a compound string object and bind it to the string.

1. In the “XmString object” field, type: `guardian`

2. Click on “Bind”.

The name of the object appears in the *XmString* object list. Finally, you can apply the compound string object to the Label.

3. Click on “Apply” in the compound string editor.

This applies the compound string object to the Label resource panel.

4. Click on “Apply” in the Label resource panel.

The text in the Label now displays the compound string in the selected fonts.

5. Direction Indicators

So far, this exercise has demonstrated two of the components in a compound string: the text and fontlist tags. The other two components are the newlines used as delimiters and direction indicators.

A newline causes a line break in the text. You can either press <*Return*> while entering text, or insert a newline using the pull-down menu. Before experimenting with inserting, moving and deleting newlines, make sure the "Show symbols" toggle is on so you can see what you are doing.

The direction indicators let you create text in either a left-to-right or a right-to-left direction. Although the default direction is from left to right, you can specify any portion of the string to be drawn from right to left.

If the "Show symbols" toggle is not on:

1. **Click on the "Show symbols" toggle to display the symbols.**
2. **Position the pointer between** *Guar* **and** *dian***, then press and hold mouse button 3.**
3. **Pull down and select "Right to left".**

The compound string changes as shown in Figure 18-7.

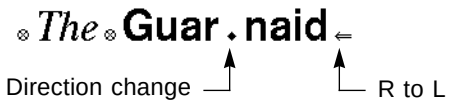


Figure 18-7 Compound String with Direction Change

The small diamond represents a change of direction from left-to-right to right-to-left. The arrow is the direction indicator, which appears at the beginning of the affected text. Since this is a right-to-left segment, the beginning of the text is on the right, not on the left. A change to left-to-right is represented by a direction indicator arrow pointing to the right at the beginning (the left end) of the new text.

Inserting newlines and fontlist tags into a right-to-left segment may seem to produce the opposite effects from what you expect if your normal reading direction is from left to right.

Remember that symbols affect the text that follows them, which means the text to the left in a right-to-left segment.

Finally, bind the compound string object *guardian* to the new string.

4. Click on “Bind”.

The Label text changes as shown in Figure 18-8.

Figure 18-8 Final Appearance of the Text String

18.2.5 Updating Changes to the Font

If you create a new font object, or decide to use a different one while using the compound string editor, you can update the “Font” field in the widget’s resource dialog by pressing the “Update font” button in the Compound String Editor. This ensures that the widget is using the same font object as the editor. This is not necessary if you have changed the contents of the font object - only if you want to use a different one.

18

Advanced Layout 19

19.1 Introduction

This chapter describes some techniques that can be used to achieve more complicated widget layouts.

19.2 Column Layout Using the RowColumn

Dialog elements are frequently arranged in a column or row. This is easy when only a single column is needed but requires more work to create multiple columns.

The easiest way to create a single column layout is to use a RowColumn widget instead of a Form. The RowColumn can take almost any widget as a child and different widget types can be children of the same RowColumn. If the orientation of the RowColumn is vertical, it produces a single column; if horizontal, it produces a single row. These results are shown in Figure 19-1.

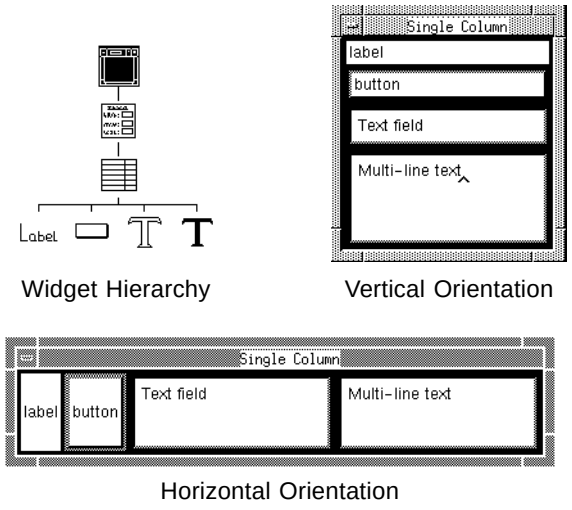


Figure 19-1 Single-Column Layouts

Figure 19-1 shows the default behavior of the RowColumn widget when the Packing resource is set to “Tight”. “Tight” packing produces exactly one column or row. In the column layout, all widgets are constrained to the same width but they can have different heights; in the row layout, all widgets are constrained to the same height but they can have different widths.

If you set Packing to “Column”, both the width and height of all widgets are the same, as shown in Figure 19-2.

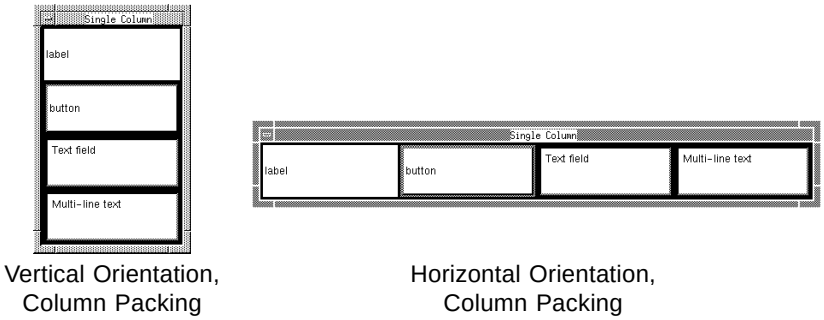


Figure 19-2 Column Packing

19.2.1 Resize Behavior of RowColumns

Since any dialog can be resized, the behavior of its components upon resizing is always important. There is no general documentation of resize behavior; you have to discover it by experimentation. A vertical RowColumn with “Tight” packing behaves as shown in Figure 19-3.

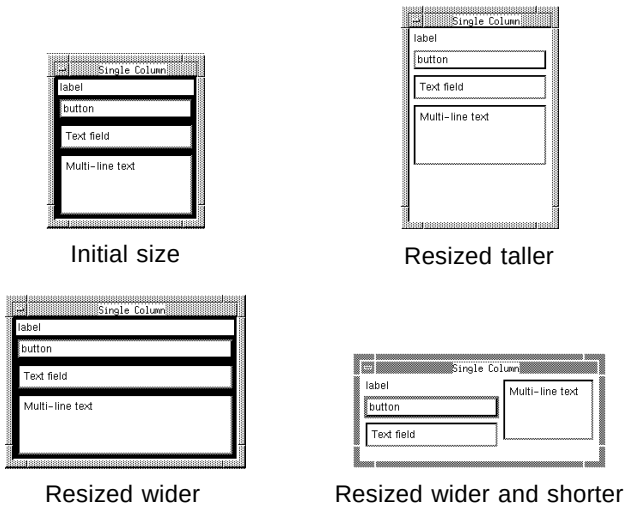


Figure 19-3 RowColumn Resizing

All children in a RowColumn widget are displayed if at all possible. If the RowColumn is not large enough to display all of its children, the ones that don’t fit are not displayed at all. It is rare to see only part of a child widget.

19.2.2 Multiple Columns

You can use the RowColumn widget to lay out widgets in multiple columns, as shown in Figure 19-4.

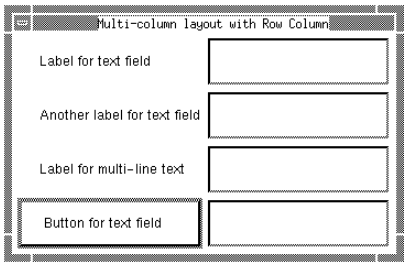


Figure 19-4 Multiple Column layout with RowColumn Widget

This layout has limitations, however. In order to get more than one column, you must set Packing to “Column”, which forces all of the children to be the same size. In this case, because one of the text boxes is 2 lines high, all of the text boxes must be that size. The result wastes space and may be confusing to the user. The layout shown in Figure 19-5 is an improvement:

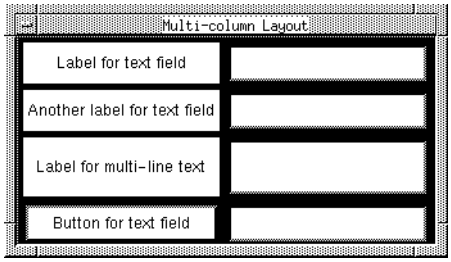


Figure 19-5 Multiple Column Layout with Form Widget

This layout cannot be achieved with a RowColumn because some of the rows are not all the same height. However, it can be achieved with a Form.

19.3 Column Layout Using the Form

The following section presents a systematic approach to creating complex column layouts. The first example is a two-column layout with a single row containing one Label and one TextField widget. Note that, unless otherwise noted, the value of all attachment offsets in this chapter is zero.

When you first create the Form and add its children, SPARCworks/Visual arranges the children down the left side of the Form by attaching the left side of each widget to the left side of the Form. The top of the first widget is attached to the top of the Form and the top of each successive widget is attached to the bottom of the one above it. Therefore, if you create a Form containing a Label and TextField, you get the layout shown in Figure 19-6.

The Form used in this example has horizontal and vertical spacing set to 5 pixels.

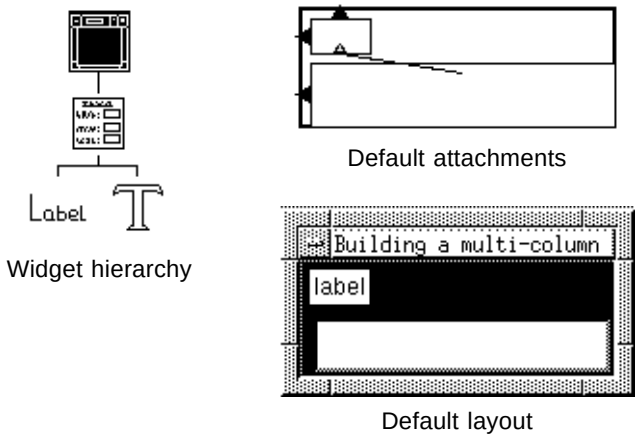


Figure 19-6 Building a Multi-Column Layout

Figures 19-7 to 19-11 illustrate how to start shaping this arrangement into a two-column layout.

First, move the TextField to its approximate position as in Figure 19-7.

Figure 19-7 Positioning the TextField Widget

Next, align the top and bottom of the Label with the top and bottom of the TextField as shown in Figure 19-8.

Figure 19-8 Aligning the Widgets

Attach the top and right side of the TextField to the top and right side of the Form, as shown in Figure 19-9.

Figure 19-9 Attaching the TextField to the Form

Attach the right side of the Label to the left side of the TextField, as shown in Figure 19-10.

Figure 19-10 Attaching the Label to the TextField

Finally, set the position of the left side of the TextField at 25%, as shown in Figure 19-11.

Figure 19-11 Setting the Position of the TextField

It may seem strange to fix the left side of the TextField by setting its position, while the right side of the Label is attached to the left side of the TextField. While it might seem more natural to fix the left side of the TextField by attaching it to the right side of the Label, this creates a circular attachment and the Form does not allow it. The 25% value used for the position is arbitrary at this stage. You cannot choose the final position until you know the widths of all the widgets, at least approximately.

19.3.1 Multiple Rows

It is easy to extend this procedure to multiple rows - just repeat the same steps. The only difference is that in the first row the extreme left and right positions (the left side of the Label and the right side of the TextField) are set by attaching them to the sides of the Form. For each subsequent row, however, these positions are set by aligning the widgets with the row above.

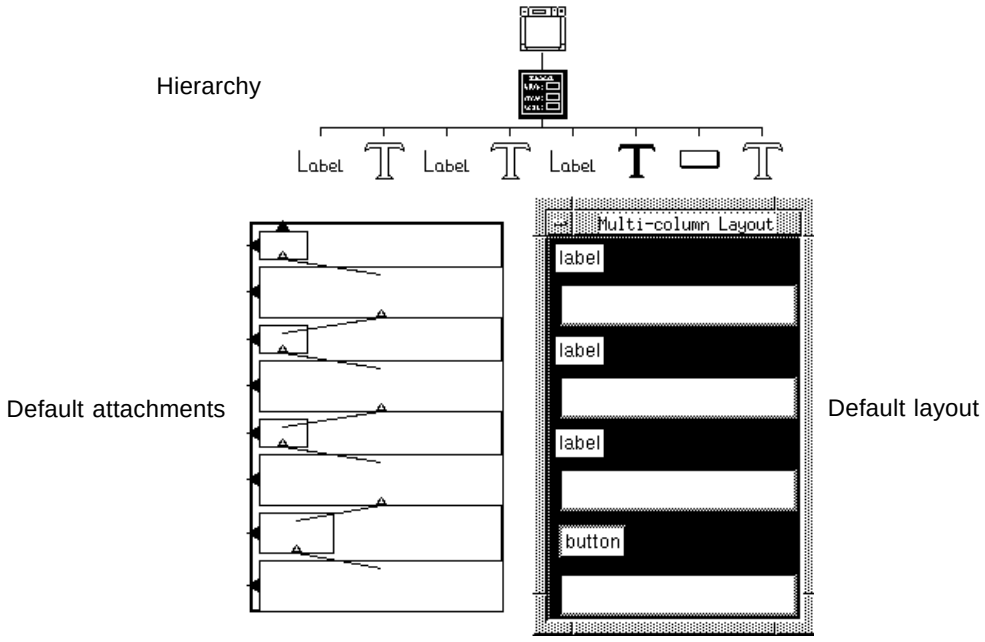


Figure 19-12 Multi-Column Layout – Initial State

Figure 19-12 shows the initial state of the dialog shown in Figure 19-5. In that dialog, the left column consist of three Labels and a PushButton while the right column contains three TextFields and a Text widget.

You can begin by setting the resources on the widgets that make up the dialog: the label text for the Labels and Pushbutton and the number of rows and edit mode of the multi-line Text widget.

It is not necessary to set these resources at this time; you can set them later if you prefer. However, setting them at this stage gives you an early impression of how the dialog will look and whether or not it is likely to work as you expect.

Set the resources to produce the layout shown in Figure 19-13.

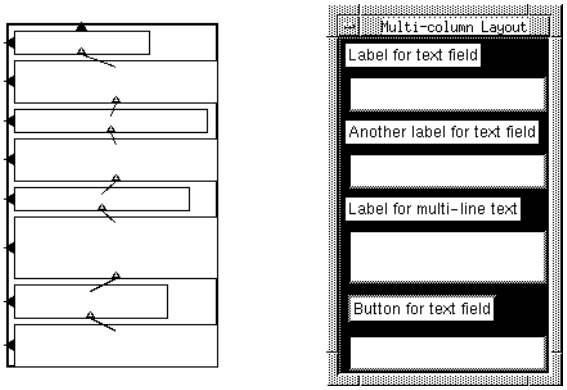


Figure 19-13 Multi-Column Layout with the Resources Set

Now you can apply the procedure illustrated in Figures 19-7 to 19-11 for each row of the column. The first step is to move the Text and TextField widgets to their approximate positions, as shown in Figure 19-14.

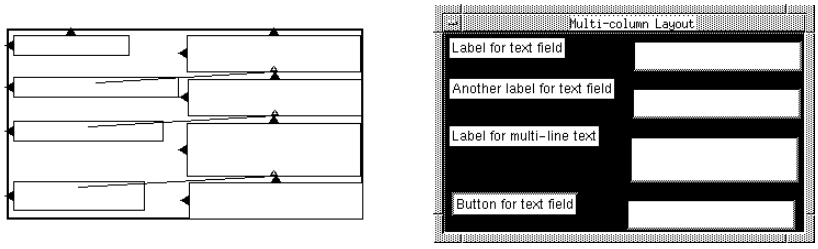


Figure 19-14 Approximate Positions

Next, align the top and bottom of the Labels and PushButton to the top and bottom of the corresponding Text or TextField widget as shown in Figure 19-15.

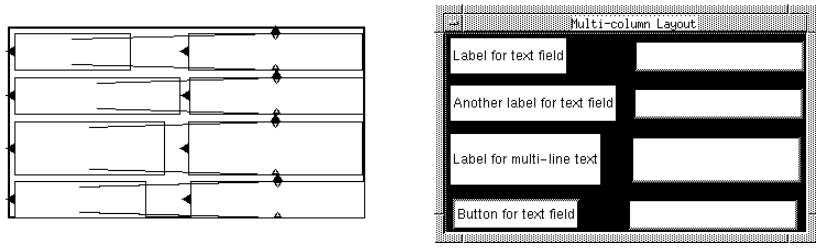


Figure 19-15 Aligning Labels and Text Widgets

As in Figure 19-9, you can now attach the TextField widget in the first row to the top and right sides of the Form using an offset of 5 pixels. Attach the top of each other TextField and Text widget to the bottom of the one above using an offset of 5 pixels.

Next, align the Text widgets with the top one. For the right sides, align the right side of each Text widget with the right side of the one above using an offset of 0 pixels. Alternatively, use “Group Align”, selecting the Text widgets from bottom to top. For the left sides, set positions to an arbitrary value such as 50%.

At this stage, align the left side of each Label and the PushButton with the left side of the widget above. Again, you can use “Group Align”, selecting the widgets from bottom to top. The layout should now resemble the one shown in Figure 19-16.

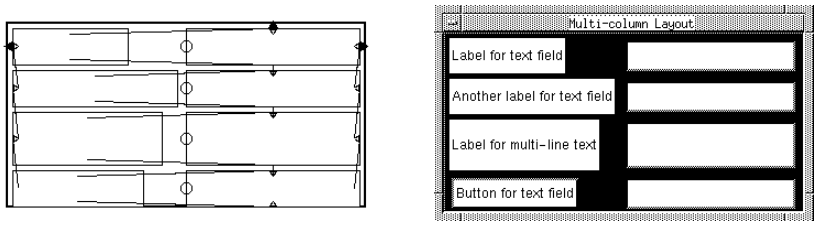


Figure 19-16 Aligning Into Columns

Attach the right side of each Label and the PushButton to the left side of the corresponding Text or TextField widget as shown in Figure 19-17.

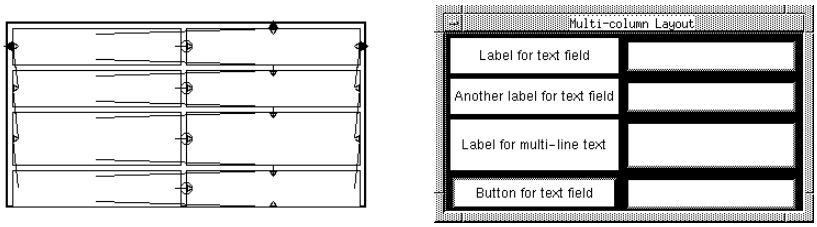


Figure 19-17 Labels Attached to Text and TextField Widgets

The final step is to adjust the position of the left side of the text widgets. This may require some trial and error. If the percentage is too high or too low, the Form is wider than necessary and wastes screen space. Some examples are shown in Figure 19-18.

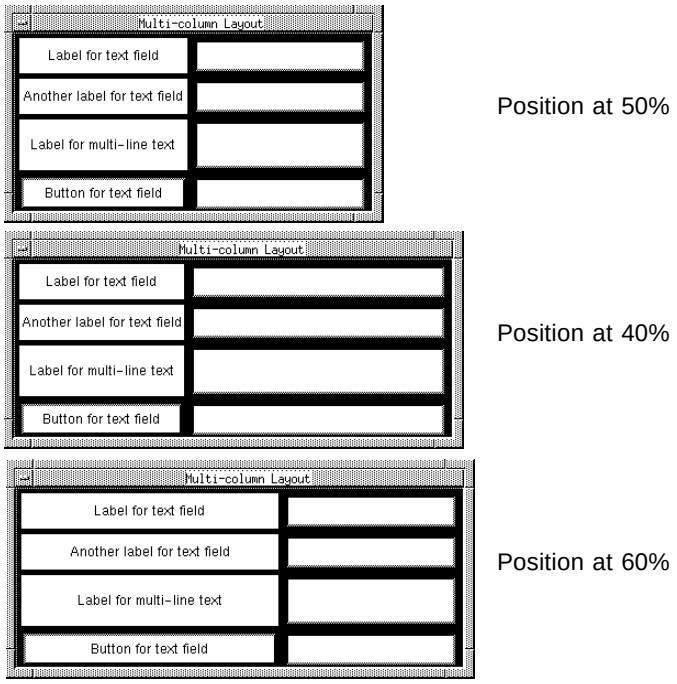


Figure 19-18 Setting the Position

19.3.2 Reset

Note that when you experiment with different position values, the Form grows each time you change it. This is because the Form is not too clever about constraints which change after the Form has been created. Reset the Form to see how it will really look in your application.

While the arrangement of attachments, alignments and positions used for the multiple column layout may seem complex, it is flexible and adaptable. In particular, it is relatively easy to add a new row to the dialog.

19.3.3 Adding a New Row

Adding a new row at the bottom of the dialog is easy; just add the new widgets to the design and set up the attachments as in the row above. Adding a new row in the middle of the dialog, however, is less straightforward.

19.3.4 Adding a Row in the Middle of the Dialog

The first step is to open a space for the new row. Because each row is only attached to the one above, and because the widget in the left column is attached at the top and bottom to the widget in the right column, you can move the whole bottom portion of the dialog down just by dragging the widget in the right column. This breaks all of its attachments and you must remake them later. Figure 19-19 shows the effect of pulling down the Text widget.

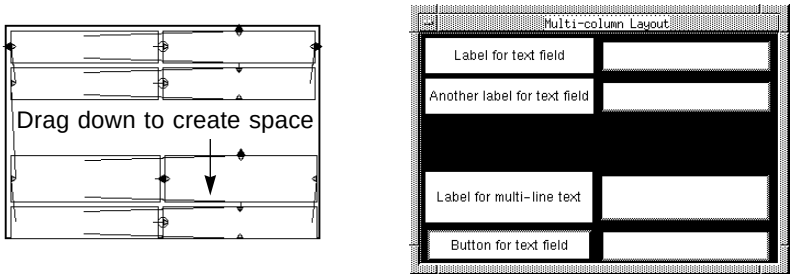


Figure 19-19 Making Space for a New Row

Now you can add the widgets for the new row, set their resources as required and move them to their approximate positions as shown in Figure 19-20.

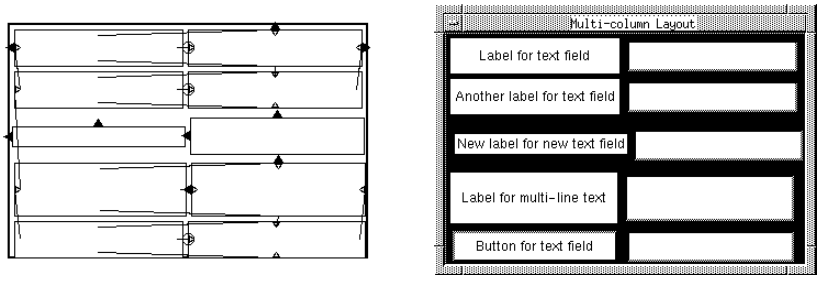


Figure 19-20 Adding the New Widgets

The next step is to align the widgets in the two columns as in Figure 19-21.

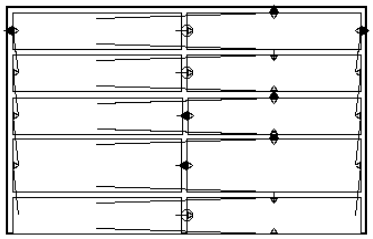


Figure 19-21 Alignments

Finally, attach each Text or TextField to the one above it and set the positions of the left sides of the Text or TextField widgets as in Figure 19-22.

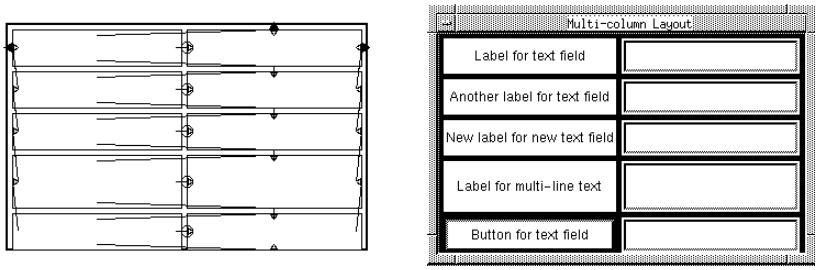


Figure 19-22 Attachments

As the new Label is a little too narrow, set the position of the left sides of the text widgets to 55% and reset the dialog to produce the result shown in Figure 19-23.

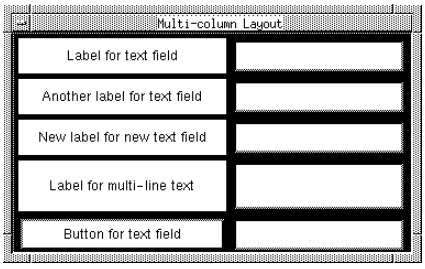


Figure 19-23 Position Set to 55%

19.3.5 Changing to Four Columns

Dialogs containing two columns, at least in part, are common. If there are so many items that a two-column layout becomes too tall, you can change it to a four-column layout.

For example, you can move the fourth and fifth rows of the example into a new pair of columns, 3 and 4. The first step is to break the attachments, shown in Figure 19-24, that keep these rows aligned with the rows above.

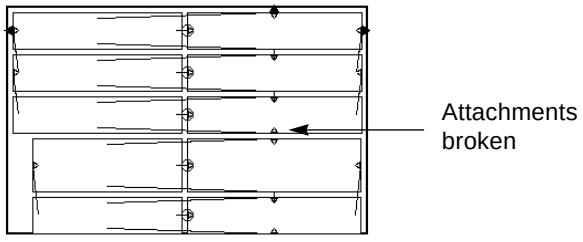


Figure 19-24 Breaking Attachments

Next you need to make room for two new columns on the right. This is done by setting positions on the first three rows which approximate the attachments they will have in the final Form. Figure 19-25 shows the result. When you change the positions, the Form becomes very wide and so you should reset the Form after changing the positions.

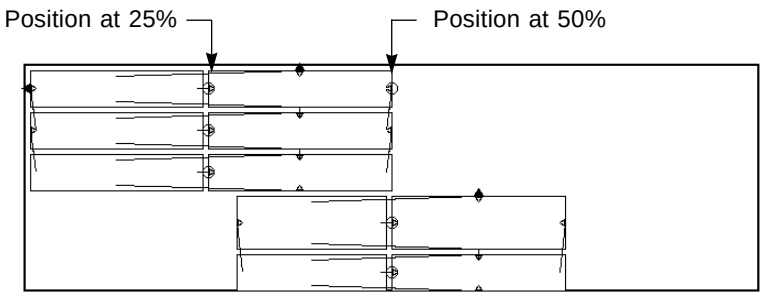


Figure 19-25 Reset Positions

Likewise, set approximate positions on the bottom two rows as shown in Figure 19-26. You must set the positions in the order shown to avoid temporarily putting the Form in a inconsistent state. If you do things in the wrong order, you get the message “Bailed out of edge synchronization”. While you can safely ignore this error message, you must dismiss the message box before continuing. Remember to reset the Form after changing the positions.

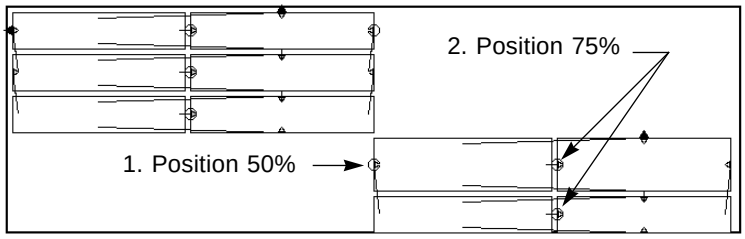


Figure 19-26 Position Right-Hand Columns

You can now move the right pair of columns up to their correct position by attaching the top Text widget to the top and right sides of the Form. This produces the result shown in Figure 19-27.

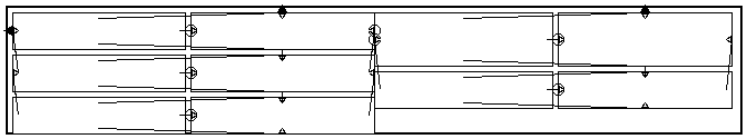


Figure 19-27 Attach to Top and Right of Form

The layout is almost finished. The right side of column 2 is currently positioned at 50%. Replace this position with an attachment to the left side of column 3, which is also positioned at 50%. Figure 19-28 shows the final layout.

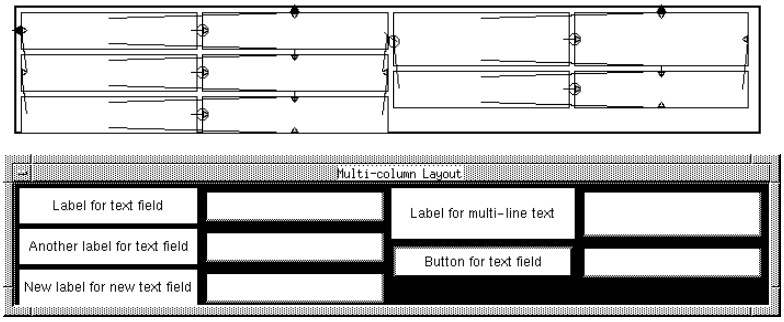


Figure 19-28 Final Layout

19.4 Edge Problems

A Form that is a child of a Shell draws a margin line round its inside edge. Any child widget of the Form that extends exactly to the edge of the Form occludes part of this margin line (Figure 19-29).

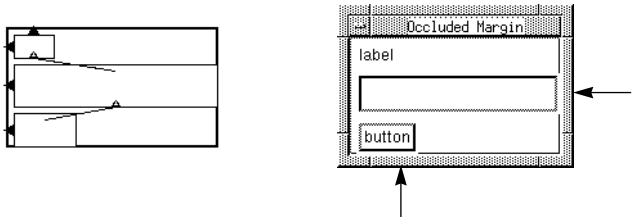


Figure 19-29 Occluded Margin

The simplest way to deal with this is to attach any widgets that overlap the margin to the sides of the Form with a small offset to reveal the margin. However, this can also produce undesirable resize behavior, as shown in Figure 19-30.

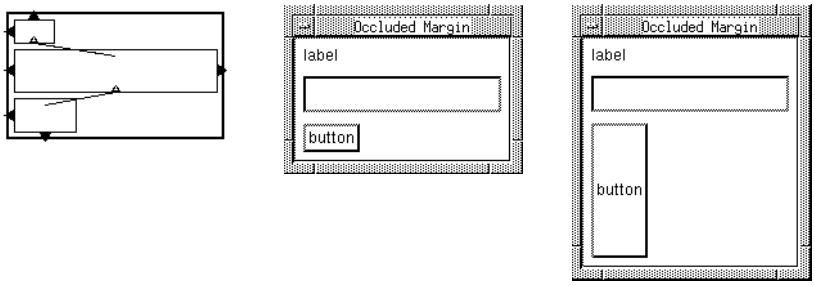


Figure 19-30 Extra Attachments and Resize Behavior

There are two other possible approaches to solving this problem, both of which involve introducing extra widgets into the design.

19.4.1 Invisible Widgets

The problem with the simple attachment in Figure 19-30 is that the button resizes when the Form does and it looks strange. An alternative is to introduce an extra, invisible widget. Although this also resizes with the Form, it does not look strange because it is invisible. A Separator gadget with the Type resource set to “No Line” is most effective. Figure 19-31 shows the widget hierarchy and attachments after adding an invisible widget to the previous example.

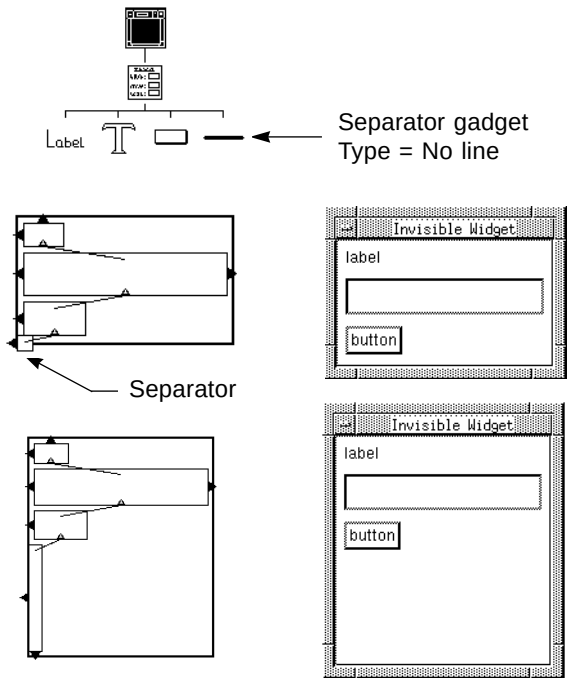


Figure 19-31 Invisible Widget

The bottom of the Separator is attached to the bottom of the Form by a small offset to prevent it from hiding the margin line. The top of the Separator is attached to the bottom of the PushButton with zero offset. The Separator now resizes vertically with the Form but the other components do not.

Note that the Form Layout Editor exaggerates the height and width of the Separator (and any other widgets under a certain size) to allow you to set its attachments using the mouse. Although this makes it look as if it is occluding the margin in the bottom left corner, in fact it is not.

You could also use a second Separator to keep the right side of the TextField widget inset from occluding the right edge of the Form (or even use the same Separator for both jobs). However, the horizontal resize behavior with the attachments shown in Figure 19-30 is appropriate for most purposes (the TextField widget resizes horizontally with the Form).

19.4.2 Doubled Forms

Another technique is to use a second Form as the invisible widget. This works because the Form only draws its margin line if it is the immediate child of a Shell. The intermediate Form, which is not a child of the Shell, does not draw a visible margin line and so its children can extend all the way to its edges without causing occlusion problems. Using an intermediate Form protects against margin occlusion on all four sides.

Figure 19-32 shows the widget hierarchy and appropriate attachments for this technique. The child Form is attached at all four sides to the parent Form with a small offset to make the margins visible.

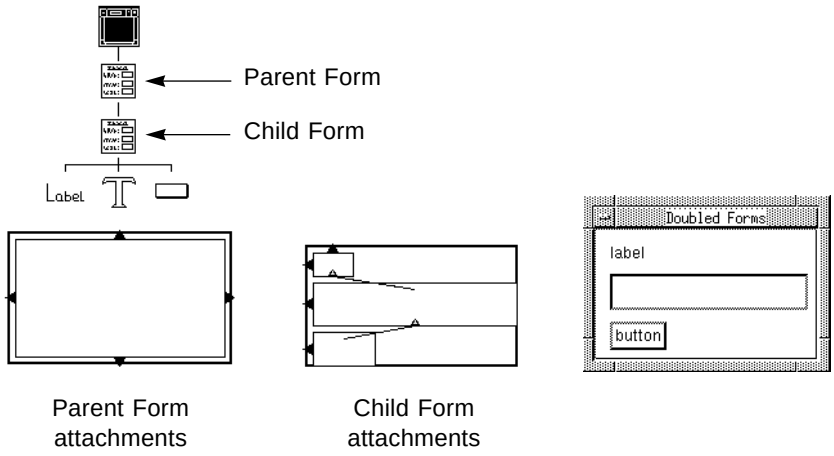


Figure 19-32 Doubled Forms

As an alternative to the parent Form, you can use a BulletinBoard, setting the margin width and height to a relatively small value such as 5 pixels. Because the Shell assumes that its child is a BulletinBoard (or a BulletinBoard derivative such as a Form), you should not use a different kind of container widget (such as a Frame) in this position.

19.5 Form Resizing

What happens when the user resizes a Form depends on how the attachments and positions are set up. In general, you should try to design every dialog so that if the user makes it bigger, more of the important information is displayed. For example, if a dialog contains a scrolling list, the user should be able to make the visible portion of the list longer by making the window taller. On the other hand, there is no reason to make widgets such as Labels and buttons grow with the window, since this conveys no additional information.

In practice, any Form layout must probably compromise between resize behavior, robust response to size changes within the widget (such as font changes), ease of implementation and ease of maintenance. This section offers some basic guidelines for creating Forms with desirable resize behavior.

19.5.1 Widget Resizing

A widget in a Form grows wider when the Form does only if it has constraints on both right and left edges; it grows taller with the Form only if it has constraints on both top and bottom.

The most straightforward way to lay out a dialog is to work from the top left corner towards the bottom right corner of the Form, attaching the top and left of each widget to the bottom and right of a previous widget. If you do this, none of the widgets resize with the Form. To make the widgets in a Form resize sensibly, you must be a little more sophisticated.

19.5.2 Horizontal Resizing

You have already seen an example of horizontal resizing with the column layout in the *Layout Editor* chapter. This rest of this chapter demonstrates some additional techniques.

The attachments you need to control resize behavior are made more complex by the Form's treatment of circular attachments. Consider a simple situation, where the attachments naturally go in only one direction.

19.5.3 Two-Widget Layouts

Layouts with only two widgets across the width of the Form are relatively simple. You only need to decide whether the extra width that results when the Form is resized is given to one widget or the other, or shared between the two.

Figure 19-33 shows one alternative. Widget 1 is attached to the Form at its left side but its right side is unconstrained; therefore, it finds its own natural width, wide enough to display the text label. Widget 2 is attached to Widget 1 on the left and to the Form on the right; therefore, its size varies to fill the part of the Form width that is not occupied by Widget 1. In other words, Widget 2 gets all the extra width.

When you first create the Form, it sets its own width to accommodate both widgets, producing the initial state shown in Figure 19-33.

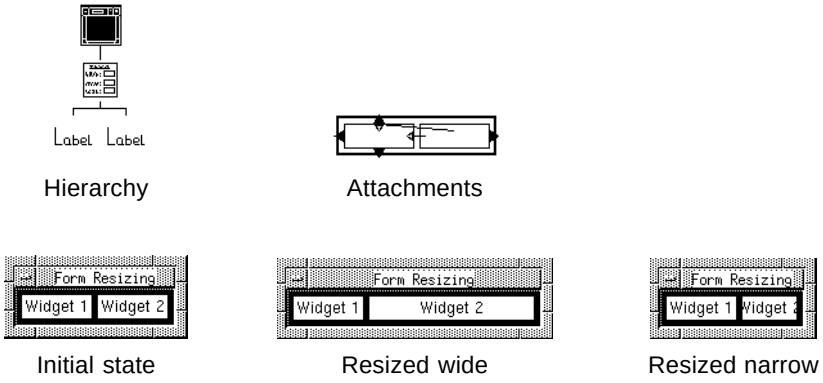


Figure 19-33 Two Widgets, Right Dominant

The attachments between the two widgets are reversed in Figure 19-34, i.e. the right side of Widget 1 is attached to the left side of Widget 2 instead of the other way around. The result is that Widget 2 stays the same size and Widget 1 gets all the extra space.

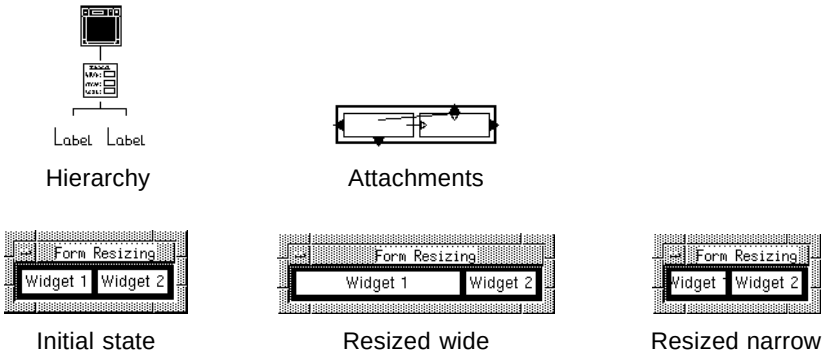


Figure 19-34 Two widgets, Left Dominant

19.5.4 Avoiding Circularity when Reversing Attachments

You can reverse a right-to-left attachment between two widgets simply by adding a new left-to-right attachment in the same place. The Layout Editor detects this situation when you add the new attachment and removes the old attachment. However, in the example just given, you will see a circularity error message when you do this because there are two attachments to be changed. To get rid of the circular attachment, you must also swap the attachment that aligns the tops of the two widgets. Since both widgets have the same height, this does not affect the appearance of your layout.

19.5.5 Proportional Spacing

If you want to share the extra space equally between the two widgets, you must use proportional positioning. You can set a position on Widget 1 and attach Widget 2 to it, set a position on Widget 2 and attach Widget 1 to it, or set positions on both. Any of these methods works as long as you avoid circular attachments. Figure 19-35 shows the three possibilities.

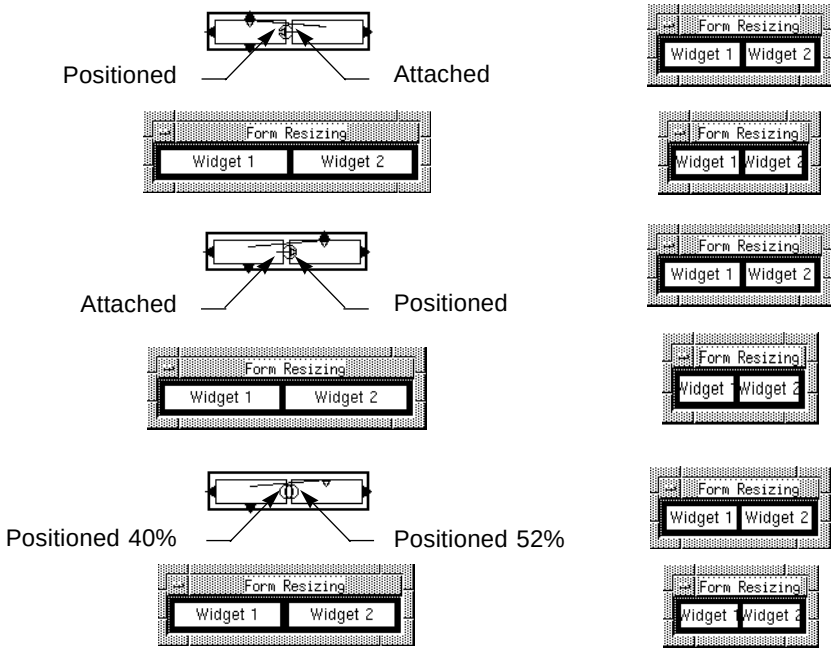


Figure 19-35 Two Widgets, Equal Shares

In Figure 19-35, the positions are set at about 50% and the two widgets share the width of the Form about equally. You can use different percentages to favor one widget or the other. In this example, the extra space is shared in proportion to the initial sizes of the widgets.

19.5.6 Three-Widget Layouts

With three widgets, there are more possibilities. The extra space can be given to any one of the three, or shared among them. Figure 19-36 shows the simple cases, where all the extra width goes to one of the three widgets. Notice that in every case the widget with both ends attached is the one that resizes with the Form.

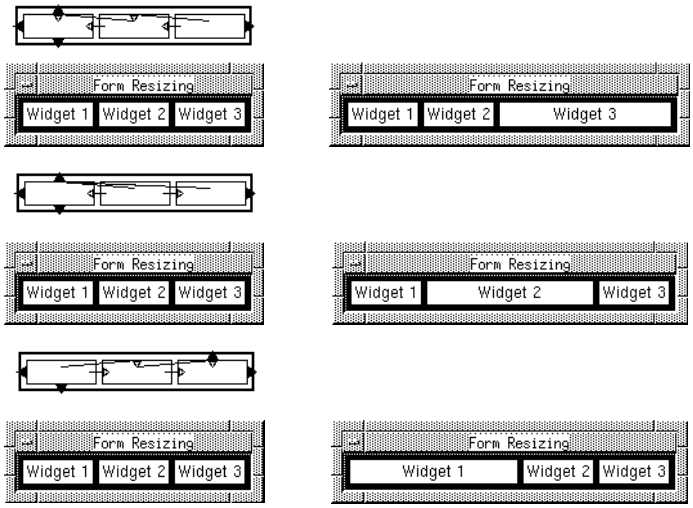


Figure 19-36 Three Widgets, One Dominant

To share the space among all three widgets, you can use simple proportional positioning, as shown in Figure 19-37.

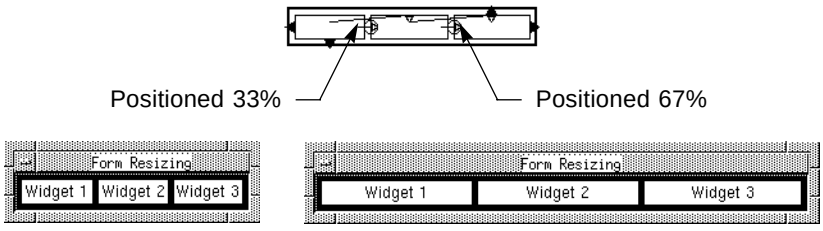


Figure 19-37 Three Widgets, Equal Shares

As with two widgets, you can use different percentages to favor one widget over another.

You can use combinations of attachments and positions to create layouts where one widget does not resize but the other two do. Figure 19-38 shows some examples.

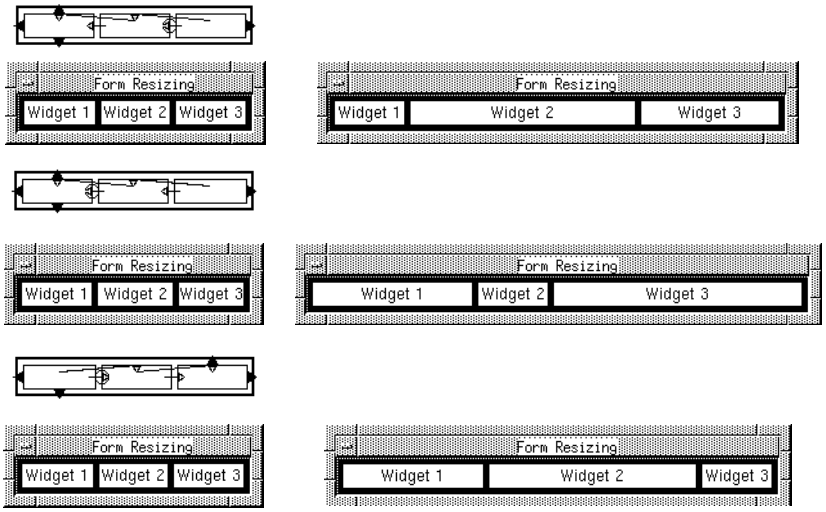


Figure 19-38 Combined Attachments and Positions

19.5.7 Widgets of Unequal Height

In all the examples so far, the widgets that share the width of the Form are all the same height. This gives you considerable freedom in arranging the attachments on the tops and bottoms of the widgets and lets you avoid circular attachments easily. If the widgets are of different heights, however, this is not so easy.

With a hierarchy like the one shown in Figure 19-39, you want a layout where the Text widget on the right makes maximum use of the available space, similar to the right-dominant layout in Figure 19-33. However, this requires attaching the widget on the right to the one on the left. You cannot do this in the example shown in Figure 19-39, because the Label widget on the left is already attached at the top and bottom to the Text widget to produce the correct vertical alignment.

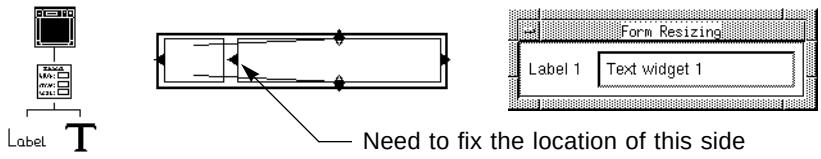


Figure 19-39 Two widget layout with Label and Text widget

There are three ways to solve this dilemma:

1. Attach the left side of the Text widget to the left side of the Form using an offset large enough so that the Text widget does not obscure the Label. Attach the right side of the Label to the left side of the Text widget as shown in Figure 19-40.
2. Position the left side of the Text widget at a given percentage and attach the right side of the Label to it as shown in Figure 19-41.
3. If there is only a single row, the alignments at top and bottom of the Label can be replaced with attachments to the Form and the left side of the Text widget can then be attached to the right side of the Label as shown in Figure 19-42.

The figures show these three approaches and their behavior when the Form resizes, the label changes and a font changes. Variations on these approaches display similar behavior.

In Figure 19-40 the Text widget is attached to the Form at both ends. The virtue of this layout is that the Text widget gets all the extra space that results if the Form is resized. However, it is only satisfactory if the user is not likely to change the application's configuration extensively; it does not behave well if the label or the font changes.

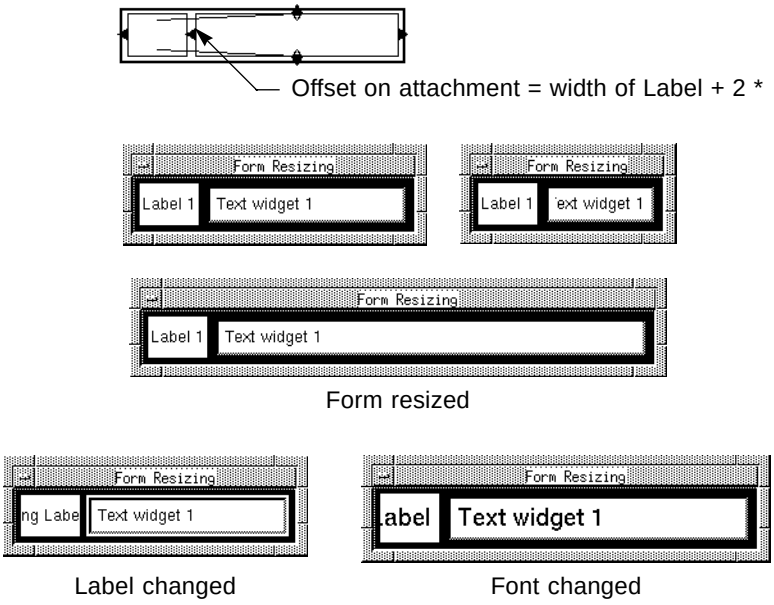


Figure 19-40 Text widget attached

In Figure 19-41, the left side of the Text widget is positioned. This is a more robust layout if the label or font changes as the label takes a share of any extra Form width. This is generally the most useful approach, especially for column layouts, as previously discussed.

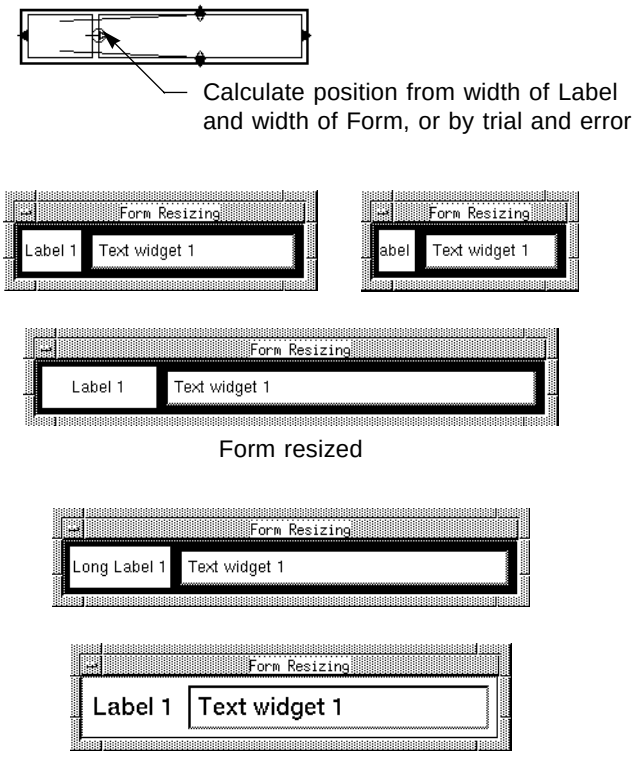


Figure 19-41 Text widget positioned

In Figure 19-42 the problem with circular attachments is removed by attaching the top and bottom of the label to the Form, instead of aligning them with the top and bottom of the Text widget. You can then attach the left side of the Text widget to the right side of the Label, producing a right-dominant layout similar to that used in Figure 19-33.

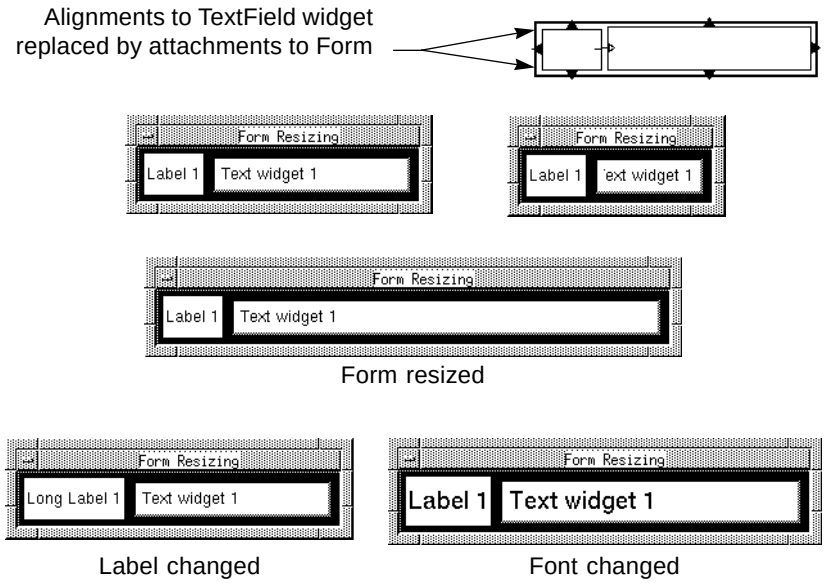


Figure 19-42 Removing circularity

This solution exhibits the best behavior. However, you cannot use it for column layouts, since each row must be enclosed in a separate Form and there is then no way to align the columns vertically. It also requires an extra Form widget for each row, which can create significant overhead.

19.5.8 Vertical Resizing

The solutions for vertical resizing are essentially the same as for horizontal resizing. However, there are usually fewer problems with circular attachments. This is because a label positioned above an object can usually be aligned with the left edge of the object, while a label to the left of an object may need to be centered in the available space. Figure 19-43 illustrates this.

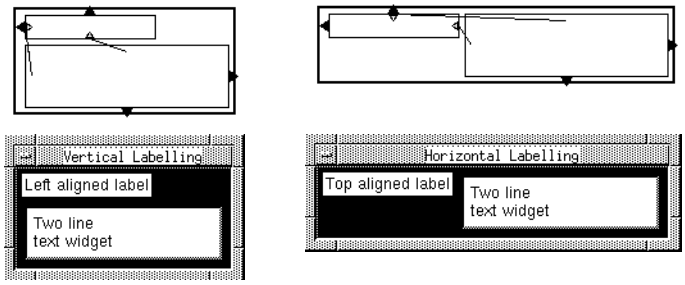


Figure 19-43 Vertical and Horizontal label placement

Figures 19-44 to 19-46 show examples of vertical layouts similar to the horizontal arrangements in Figure 19-36. In each case, any extra height is given to the multi-line Text widget.

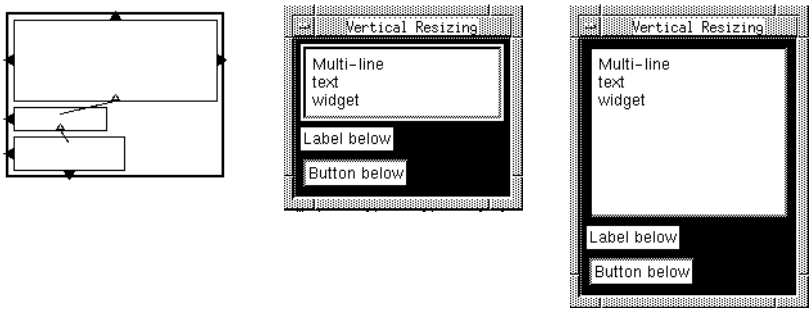


Figure 19-44 Resize top widget

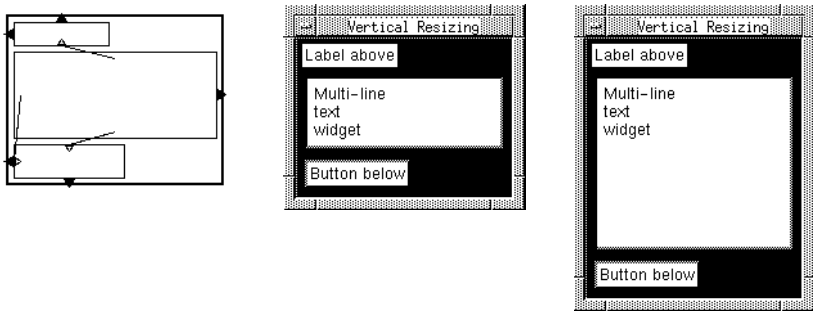


Figure 19-45 Resize center widget

Figure 19-46 Resize bottom widget

19.5.9 Initial Size

The initial size of a dialog is determined by a process of negotiation between the Shell, the Form and the widgets within the Form. Normally the Form tries to find a layout that satisfies all the constraints on its children and lets each be at least as large as it wants to be, which is at least the “natural” size. The Form then sizes itself to contain this layout and the Shell sizes itself to contain the Form.

Most widgets have a sensible natural size. Labels, for example, have a natural size determined by the text content of the label and the font; Text widgets normally size themselves according to the number of rows and columns specified by their resources.

If your dialog only contains widgets that have an acceptable natural size, you can let the Form work out the initial size for you. Problems only arise when the dialog contains widgets that do not have an acceptable natural size. You must then fix their size with constraints, which may make the initial size of the dialog seem too small.

If you have this problem, set the initial size of the dialog by setting the width and height resources of the top level Form. You cannot set the initial size using the width and height resources of the Shell, although you can set a minimum size using the appropriate Shell resources.

Translations 20

20.1 Introduction

The aim of SPARCworks/Visual is to let you develop as much of your application as possible without writing code. You still need to write code to implement the application functionality and link it to the user interface. You must also write code to control the behavior of the user interface. However, the X toolkit translations mechanism lets you change the way individual widgets respond to events; by using this, you may be able to avoid some coding.

Note that Translations are not supported on Windows. For this reason, the Apply button in the Translations dialog turns pink.

This chapter briefly describes translations and their use in SPARCworks/Visual. For more complete information on the translations mechanism, consult the *Bibliography*.

20.2 Translations and Actions

Widgets have behavior. For example, when a user presses mouse button 1 over a PushButton, it highlights. When the user releases the mouse button, the PushButton's appearance reverts to normal and the functions on the Activate callback list are invoked.

This behavior is not hard-wired into the PushButton widget. Instead, it is determined by the widget's *translation table*, which maps events to the actions to be taken in response to the events. When a widget is created, its translation table is initialized to contain a default set of entries. For example, the PushButton widget's default translation table includes these entries:

```
<Btn1Down>:Arm()
<Btn1Up>:Activate() Disarm()
```

To the left of the colon is an event specification; to the right are the names of the actions that the widget performs in response. A second table, the action table, is used to map the action name to the address of a function that performs it.

For example, the PushButton's default action table includes:

```
"Arm",Arm
"Activate",Activate
"Disarm",Disarm
```

The first item in each entry is an action name and the second is the name of a function. Convention and common sense dictate that the action and function names should be the same, or at least related in a well-defined way.

You can change the translation table of any widget within SPARCworks/Visual. You cannot change the action table of a widget. However, you can define new actions in an application-global action table which is searched after the one associated with the widget. This requires some coding, as described below.

20.3 Modifying a Translation Table

Modifying the translation table of a widget in SPARCworks/Visual is straightforward. To understand the procedure, do the following simple exercise in SPARCworks/Visual.

1. **Create a simple widget hierarchy containing a PushButton.**
2. **Select the PushButton icon in the widget hierarchy.**
3. **From the "Widget" menu, select "Translations...".**

This displays the translations dialog, shown in Figure 20-1.

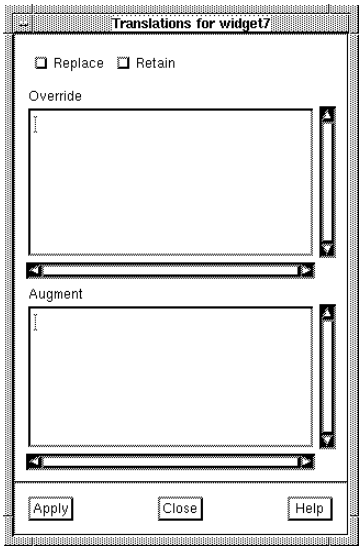


Figure 20-1 Translations Dialog

4. **Click in the lower section, under “Augment”, and type**: `Ctrl<Key>q: ArmAndActivate()`
5. **Click on “Apply”.**

This adds the new translation to the PushButton widget and you can now try its effect.

6. **Place the mouse pointer over the pushbutton in the dynamic display window.**
7. **Type** <*Ctrl-Q*>.

The effect is identical to clicking with mouse button 1. Note that translations do not work if the window does not have the input focus and that the input focus behavior depends on the configuration of your window manager.

You can also change the translations you have specified so that the button triggers on other events. Note that if you do this, the previous translation remains effective in addition to the new one until you reset the widget.

20

20.4 Augment, Override and Replace

The translations dialog has sections labeled “Override” and “Augment”. You can enter new translations in either section or both. They only differ if you specify a translation with the same event specification as an existing translation. If you type the new translation into the “Override” box, your new translation replaces the existing one. If you use “Augment”, the existing translation takes precedence.

The existing default translations for the widget are not affected when you add translations unless you override them. This is important because Motif widgets have many default translations that produce their expected behavior.

If you set the “Replace” toggle, however, all existing translations are removed and replaced by the translations you enter. Use “Replace” with caution. Do not confuse “Replace”, which removes all the default translations, with “Override”, which replaces them one by one.

20.5 Translation Table Syntax

The syntax of translation tables is complex. For a complete and definitive description, consult the X toolkit documentation.

Each entry in a translation table has the form:

```
[modifier_list]<event>[,<event>...][(count)][detail]:
      [action([arguments])...]
```

Square brackets ([]) indicate that an item is optional; an ellipsis (...) indicates that the item may be repeated.

20.5.1 Modifier List

The *modifier list* represents the state (pressed or not pressed) of the modifier keys (such as Control and Shift) and the mouse buttons (X believes that a mouse has five buttons). The most useful modifiers are Ctrl, Shift, Alt and Meta. These can be abbreviated as c, s, a and m.

If the modifier list is omitted, the state of the modifiers is unimportant:

`<Key>Q` matches *<Q>*, *<Ctrl-Q>*, *<Alt-Meta-Q>*, etc.

If a particular modifier is not mentioned in the list, its state is unimportant:

`Ctrl<Key>Q` matches <*Ctrl-Q*>, <*Ctrl-Meta-Q*>, <*Ctrl-Alt-Meta-Q*>,...

You can specify multiple modifiers in the modifier list:

`Ctrl Meta <Key>Q` matches <*Ctrl-Meta-Q*> but not <*Ctrl-Q*> or <*Meta-Q*>

To specify that a modifier must not be pressed, precede it with a tilde (~):

`Ctrl ~Meta<Key>Q` matches <*Ctrl-Q*> but not <*Ctrl-Meta-Q*>

To specify that the modifiers pressed must exactly match what you specify, start the modifier list with an exclamation mark (!):

`!Ctrl<Key>Q` matches <*Ctrl-Q*> but not <*Ctrl-Meta-Q*> or <*Ctrl-Q*> with a mouse button pressed.

The modifier "None" means that there must be no modifiers pressed at all.

`None<Key>Q` matches <*Q*> but not <*Ctrl-Q*> or <*Alt-Meta-Q*>, etc.

Normally, translations are not case-sensitive. `<Key>Q` matches both <*Q*> and <*q*>. You can specify that a translation is case-sensitive by preceding it with a colon (:).

`:<Key>Q` matches <*Q*> but not <*q*>

20.5.2 Event and Count

The *event* can be the name of an X event, or one of a number of aliases. Some of the most useful events are *Key* (a key press), *BtnDown* and *BtnUp* (for any mouse button) and *Btn*N*Down* and *Btn*N*Up (*where N is between 1 and 5). For a complete list of events and aliases, see the Xt documentation.

`<Key>a` matches <*a*>

`<Btn1Up>` matches a release of mouse button 1

You can specify a sequence of events in a translation, separated by commas.

`<Key>Q,<Key>A` matches <*Q*> followed by <*A*>, with no intervening event.

`<Btn1Down>,<Btn1Up>` matches a click of mouse button 1.

The count can be used with button press and release events to detect multiple clicks. The count is a number from 1 to 9, possibly followed by a plus (+).

`<Btn1Down>(2)` matches two presses of mouse button 1

`<Btn1Up>(3+)` matches 3 or more releases of mouse button 1

If a count is used, the button events must come close together (usually within 200 milliseconds of each other), or there is no match.

20.5.3 Detail

The final field in the event specification is the *detail*. This is normally used only with key events, where the detail specifies which key is to be pressed.

The value specified in the detail field is a *keysym*, as in the header *<X11/keysymdef.h>*, with the *XK_* prefix removed. For most keys, this is the same as the character on the key.

`<Key>a` matches *<a>*

For non-alphanumeric keys, check the name of the keysym. The keysym for "+" is *XK_plus*, so

`<Key>plus` matches <+>

Since matching is case-insensitive, this also matches the other symbol on the plus key, which is <=> on most keyboards.

Motif adds another level of complexity by translating certain incoming key events into Motif virtual keysyms. You should use these virtual keysyms in your translation tables instead of the X ones.

`<Key>osfDelete`, not `<Key>Delete`

The virtual keysyms are listed below. For details of their interpretation, see the *VirtualBindings(3X)* section of the *Motif Programmer's Reference*.

Table 20-1 OSF Virtual Keysyms

osfActivate	*osfAddMode*	*osfBackSpace*	*osfBeginLine*
osfCancel	*osfClear*	*osfCopy*	*osfCut*
osfDelete	*osfDown*	*osfEndLine*	*osfHelp*
osfInsert	*osfLeft*	*osfMenu*	*osfMenuBar*

Table 20-1 OSF Virtual Keysyms

osfPageDown	*osfPageLeft*	*osfPageRight*	*osfPageUp*
osfPaste	*osfPrimaryPaste*	*osfQuickPaste*	*osfRight*
osfSelect	*osfUndo*	*osfUp*	

You can also use the detail field with mouse button events to specify a particular mouse button. This is not commonly done since it is easier to specify the mouse button in the event field.

`<BtnDown>Button1` is the same as `<Btn1Down>`

20.5.4 Actions

The actions on the right side of the translation table entry are simple. Usually each action is just a name followed by parentheses. Although any number of string arguments can be given between the parentheses, most action routines expect no arguments. Arguments should not be quoted. Typical additional translations for a ScrollBar widget might be:

```
<Key>d:IncrementDownOrRight(0)
<Key>u:IncrementUpOrLeft(0)
```

You can specify multiple actions or none at all. Overriding an existing translation with one that has the same event specification but no action is a useful way of disabling part of a widget's default behavior.

In many cases, the actions used are the ones predefined by the toolkit. The *Additional Actions* section on page 448 discusses how to add your own actions.

20.6 Translation Table Ordering

When an event is received, the translation table is searched from the top down. The search terminates at the first entry whose event specification matches the event. This means you should organize your translation table with the most specific events first. For example, a translation table might contain the following entries:

```
<Key>q: action1()
Ctrl<Key>q: action2()
```

When the user types either <*Q*> or <*Ctrl-Q*>, the search terminates at the first entry and *action1()* is invoked in both cases. To make <*Ctrl-Q*> invoke action2, you must reverse the order of the entries.

For additional subtleties in ordering translation tables, see the X toolkit documentation.

20.7 Available Actions

By changing the translation table, you can make a widget perform actions in response to event sequences that would not normally trigger those actions. While you can write your own action routines, translations provide the most benefit when you can use one of the built-in actions of the widget.

The built-in actions of the Motif toolkit are documented in the *Motif Programmer's Reference*. Each widget description includes both the default translations and the actions they invoke. Some of the primitive widgets offer a particularly large set of actions.

If you add a translation that uses one of these actions, you can test it in SPARCworks/Visual immediately. Alternatively, a few built-in actions, such as the PushButton's *Activate()* action, invoke the functions in one of the widget's callback lists. In this case, it may be easier to specify a translation to call that action on the appropriate event sequence and put the code in an ordinary callback function.

20.8 Additional Actions

If you cannot find a built-in action to suit your needs, you can write your own *action routine* to perform the action. You can specify the name of your action routine in the translations dialog. When you do, SPARCworks/Visual displays an "Actions not found" message to warn you that the action is not known to the toolkit. However, SPARCworks/Visual still registers the translation and generates code to add it in your final application. You can then write code to define the new action. It is invoked by the X toolkit when the appropriate event occurs.

In order for the toolkit to find your action routines, you must register them using *XtAppAddActions()*. This takes a record of type *XtActionsRec*, which defines the mapping from action names to action routines. Code like this is usually put into your initialization:

```
/* Define the actions table. */
static XtActionsRec myactions[] ={
    {“action1”, action1},
    {“action2”, action2}
};
...
/* During initialization, merge the extra actions into the global actions table. */
    XtAppAddActions(app_context, myactions, XtNumber(myactions));
```

Since the actions table is global to the application, actions registered this way can be used from any widget.

The final step is to write the action routine itself. This is similar to a callback function but with different parameters:

```
void action1(Widget w, XEvent *event, String *params, Cardinal
      *num_params)
{
/* Whatever code is needed to implement the action goes here. */
}
```

Note – The action routine does not get any client data. It does get (via *params*) the string arguments specified in the translation table and their number.

20

Hypertext Help 21

21.1 Introduction

SPARCworks/Visual provides extensive on-line help which can be accessed from many different points in the application. This chapter describes the support that SPARCworks/Visual provides for building help into your application.

21.2 The Help Model

The model used to support help in your application is very simple. At certain points you may want to let the user request help, either by explicitly pressing on a Help button, or by invoking the Help callback associated with every Motif widget. Therefore, to provide context-sensitive help, all you need is a generic callback that takes as its client data the help (or a path to the help) to be displayed when the callback is invoked. In SPARCworks/Visual this help specification is defined as a document path and a marker that denotes some reference in that document. We supply a callback that can be used to interface with FrameMaker, which assumes that the document is a FrameMaker document and the marker is a hypertext marker in that document. Obviously you are free to re-implement the callback that you use in your application to make it interface with the help system of your choice. Likewise, you can achieve the same effect simply by using the ordinary callback mechanisms, or you can decide not to give your users any on-line help at all.

SPARCworks/Visual uses the pre-defined FrameMaker callback internally when displaying help. Therefore, in order to try out your help within SPARCworks/Visual you must have access to FrameMaker.

21.2.1 Motif's Help Callback

By default in Motif the Help callback is invoked by the Help action which is specified for every widget. This action is bound to the <*osfHelp*> key using a default translation in every Motif class.

21.2.2 FrameMaker and Hypertext

If you plan to use FrameMaker to build your help system, you must know how the FrameMaker hypertext system works. Complete documentation is provided in your FrameMaker manuals and a brief summary is given below.

FrameMaker lets you mark places in the text as either source or destination markers. Destination markers are places you can jump to from other source markers, either in this document or another document, or from your application. Source markers are places in the text that the user can select which cause an action, usually a jump to a destination marker. Source markers require a visual clue to tell the user he can click on them. The best way to do this is to specify a character format such as italics or underline to denote a source link. When the user clicks in a document at a place where the special character format is in effect, FrameMaker checks for a source marker in that area. If it finds one, it highlights the whole graphical area or section of text and executes the action specified by the marker.

You can insert these special markers into your FrameMaker document by using the "Marker" command from the Special menu.

1. **Select "Hypertext" from the Marker Type list.**
2. **To specify a destination marker, type:** `newlink` <*tag_name*> **in the Marker Text field.**
3. **To specify a source marker, type:** `gotolink` <*tag_name*>

Note – A tag can be specified either as <*tag_name*> for a tag in the current document, or as <*document_name*>:<*tag_name*> for a tag in a different document.

4. **Hypertext markers are only effective in a locked document. To toggle the lock of a document on or off, type:** <*escape*> <*F*> <*l*> <*k*>

5. **To exit from a locked document, type:** <*escape*> <*f*> <*c*>

This command gives you the option of saving the document in its locked state.

21.3 Setting up Help in SPARCworks/Visual

The term *tag* means the combination of a document and a marker. Help Tags are specified from the “Code generation” page of the Core resource panel. There are two Help tags that can be specified: one for the Help callback, which can be specified for any Motif widget, and one for the Activate callback, which can only be specified for PushButton and CascadeButton widgets.

Note that the Activate callback for a CascadeButton is only called if there is no pulldown menu associated with it.

To specify a tag, type the name of the document and the name of the marker, and press “Apply”. You should now be able to display that document by invoking the callback in the dynamic display, which is done by clicking on the button for an Activate callback, or pressing <*F1*> for a Help callback.

When you specify tag information in the Core resource panel, SPARCworks/Visual automatically adds the callback function *XDhelp_link()* to the appropriate callback list for the widget. You do not have to specify a callback in the Callbacks dialog for the widget.

21.3.1 Inherited Documents

You may be able to specify some of your help information by typing the marker only, without having to retype the name of the help file. If you specify a marker but no document name for the Help callback, SPARCworks/Visual uses the document for the Activate callback. If there is no Activate callback document, the Help callback document of the closest parent widget is used. If none of the widget's ancestors has a Help callback document, SPARCworks/Visual uses the default document specified in the Module Help defaults panel, as described below.

21.3.2 Module Defaults

There are numerous help defaults that can be specified on a per module basis. To specify these defaults, pull down the Module Menu and select "Help defaults". The resulting dialog is shown in Figure 21-1.

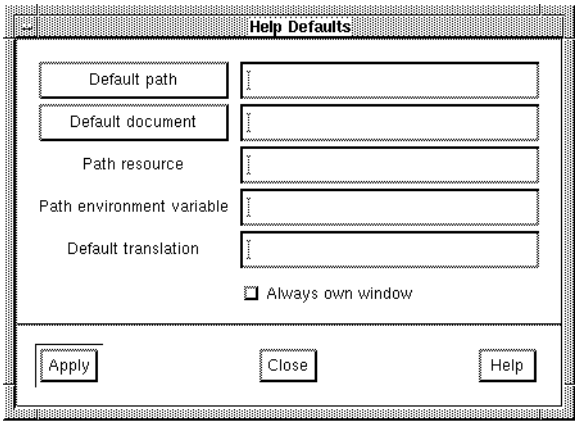


Figure 21-1 The Help Defaults Dialog

The Help Defaults Dialog lets you specify defaults for use both in building the help system in SPARCworks/Visual and in the finished application.

Default document:	This field specifies the name of the default document for use if no other document is specified on an individual widget.
Default path:	All help tag document names are assumed to be relative paths unless they begin with "/". This field specifies a default path to be added to the beginning of any relative path document names. The Default path is also used to find documents when you test your help in SPARCworks/Visual.
Path resource:	In the application, you can override the Default path setting by setting an application resource. This field specifies the name of the application resource.
Path environment variable:	You can also override the resource setting by setting an environment variable. This field specifies the name of the environment variable.
Default translation:	This field specifies an event to be added as a translation to every widget that has a marker specified for the Help callback. The translation calls the *Help()* action. This lets you designate a key combination as an application help key.
Always own window:	This toggle makes the default FrameMaker integration callback display the document in its own window. If this toggle is off, an existing window is used to display the new page. You can designate the use of a new window for individual pages using the "Own window" toggle in the Help documents and markers dialog.

21.3.3 Finding Help Documents and Markers

The “Activate callback” and “Help callback” buttons in the “Code generation” page of the Core resource panel can be used to display a dialog showing all the documents and markers that are currently referenced by your design.

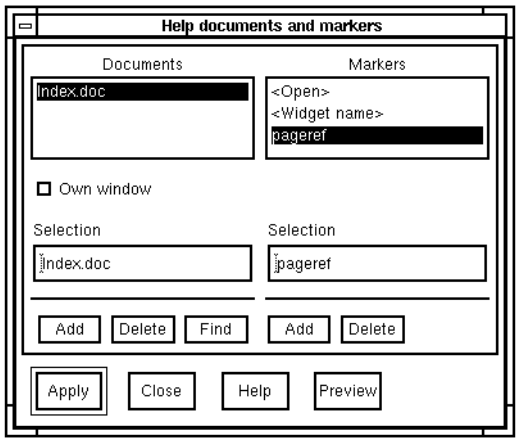


Figure 21-2 Help Documents and Markers Dialog

Add:	You can add a document or marker by typing the name in the appropriate selection field and pressing “Add”.
Delete:	You can delete a document or marker by selecting its name from the list and pressing “Delete”. You cannot delete documents or markers that are still referenced by a widget in the design.
Find:	This button pops up a file selection dialog to help you navigate in the file system. You can also display this dialog by selecting the “Default path” or “Default document” buttons in the Help defaults dialog.
Preview:	This button makes SPARCworks/Visual try to connect to FrameMaker and display the named document at the named marker.

Own window:	Each document has an “Own window” flag associated with it. This flag forces the system to display this document in its own window, not to share a common window with other documents. This toggle lets you designate the state of the flag for the selected document.
<*Open*>:	A special default marker that causes the document to be opened at the first page.
<*Widget name*>:	A special default marker that uses the widget name as a marker to jump to in the document.

21.3.4 Linking with the Default Callback Function

To use SPARCworks/Visual’s default help callback function, you must link in the following object files from the SPARCworks/Visual directory:

```
$VISUROOT/libhelplink/helplink.o
$VISUROOT/libhelplink/libframe/fmclient/xdr.o
```

where $VISUROOT is the path to the root of the SPARCworks/Visual installation directory.

You can also supply your own Help callback function. This function should also be called *XDhelp_link()* and should follow the same form as SPARCworks/Visual’s function. See the *Help Implementation* section below for suggestions on how to customize the Help callback function.

21.4 Help Implementation

The preceding descriptions show how the help system works within SPARCworks/Visual. If you use the default Help callback provided with SPARCworks/Visual, the help in your application will work the same way. However, you may want to customize your implementation.

The SPARCworks/Visual directory *libhelplink* contains all the sources for the FrameMaker integration callback *XDhelp_link*. The subdirectory *libframe* contains various source files, adapted from the FrameMaker release. The original files are in *$FMHOME/source/openmaker* if you want to check them out.

XDhelp_link() receives a *client_data* parameter that is a pointer to an *_XDHelpPair_t* structure:

```
typedef struct _XDHelpPair_s {
_XDHelpDoc_pdoc;
char**tag;
Boolopen_doc;
} _XDHelpPair_t, *_XDHelpPair_p;
```

This contains a pointer to an *_XDHelpDoc_t* structure, a pointer to a marker (*tag*), and an *open_doc* flag. If tag is *NULL* the developer has specified one of the default markers: <*Widget name*> if *open_doc* is *FALSE*, <*Open*> if *open_doc* is *TRUE*. The document structure is:

```
typedefstruct_XDHelpDoc_s{
char*doc;
char**path;
int handle;
Boolnew_window;
}_XDHelpDoc_t,*_XDHelpDoc_p;
```

The *doc* field is the name of the document. *path* is a pointer to the default path if one exists. This path is calculated as described above, taking account of the setting of the help path application resource and environment variable. *handle* is the document handle returned from FrameMaker, and *new_window* specifies whether the document is to have its own window.

The primary module contains static arrays of documents and markers, and an array of tag pairs that points into the other arrays. A pointer to an element in the tag pairs array is passed as the client data.

XDhelp_link() calls the appropriate routines to communicate with FrameMaker to display the document as requested. You are free to use a different implementation of *XDhelp_link()* to communicate with a different help system.

Internationalization 22

22.1 Introduction

X11 Release 5 offers numerous special features to help you develop applications that can be used in different languages without modification. Motif 1.2 takes advantage of some of these features. Internationalization is a complex and confusing subject and the mechanisms in place are not fully utilized or completely stable. With this caveat in mind, this chapter describes the current SPARCworks/Visual support of Motif's internationalization capabilities. This chapter does not try to explain the concepts of internationalization completely; it provides only an introductory section on locales and a simple tutorial on using an input method. For additional discussions, see the Motif documentation and the O'Reilly X11R5 books.

Although the following examples use Japanese, the principles are the same for any language. SPARCworks/Visual is released with a defaults file explicitly tailored for the ja_JP.jis7 locale.

22

22.2 Locale

A locale is an ANSI-C concept. It is a name that is used to identify a set of local information. For example, the locale might be set to be “En_UK” to denote that the location is English as used in the U.K., “En_US” for English as used in the U.S., or “ja_JP.ujis” for a version of Japanese. The setting of a locale makes certain C library functions operate in different ways, e.g. defining the sorting order, or date format. In the last example, the “.ujis” defines the encoding (the mapping between numbers and characters) to be used for all strings in the application.

If you do not have ANSI-C internationalization features, you can use a set of alternatives provided (optionally) by Xlib.

To run SPARCworks/Visual with the internationalization support, you must set an appropriate locale. This is normally done by setting the *LANG* environment variable:

```
setenv LANG ja_JP.jis7
```

This makes SPARCworks/Visual use the Japanese defaults file. The only noticeable difference at this stage is that certain resource panels appear wider. This is because some Text widgets provide for Japanese input, as explained later.

22.3 Font Sets

X11R5 defines a new structure called a *FontSet*. This is simply a collection of FontStructs. This may seem similar to Motif’s FontList structure, but is in fact completely different.

Part of a locale’s definition includes a specification of the font encodings required in order to display all possible text. For example, the ja_JP.jis7 locale requires that fonts with encodings for iso8859-1, jisx0208.1983 and jisx0201.1976 be present in order to display text correctly. Therefore, a FontSet is a collection of Font definitions that contains all the required encodings for the current locale.

To further complicate things, Motif's definition of a FontList is extended so that each entry in a FontList can be either a FontStruct or a FontSet. SPARCworks/Visual tries to simplify this by presenting all the FontStructs in a FontList as tags within a Font object. Those FontStructs that are part of a FontSet are additionally tagged with the encoding name.

This complexity may be clarified by the following exercise.

The Japanese locale described above requires the following three encodings:

- iso8859-1 for Latin characters
- jisx0208.1983 for Kanji ideographic characters
- jisx0201.1976 for Kana phonetic characters

1. **Pop up the Font selection dialog.**
2. **Enter a Font object and a tag name.**
3. **Set the "Font set" toggle on.**
4. **Use the Reg Filter Menu to show all iso8859-1 fonts.**
5. **Select an appropriate iso8859-1 font and press bind.**
   At this point, Xlib warns you that you have a FontSet that does not have sufficient encodings (charsets) to display text in the current locale. This is correct, since you still have two fonts to do.

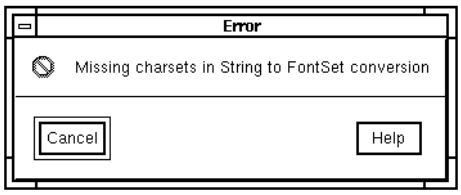


Figure 22-1 Missing Charsets Warning

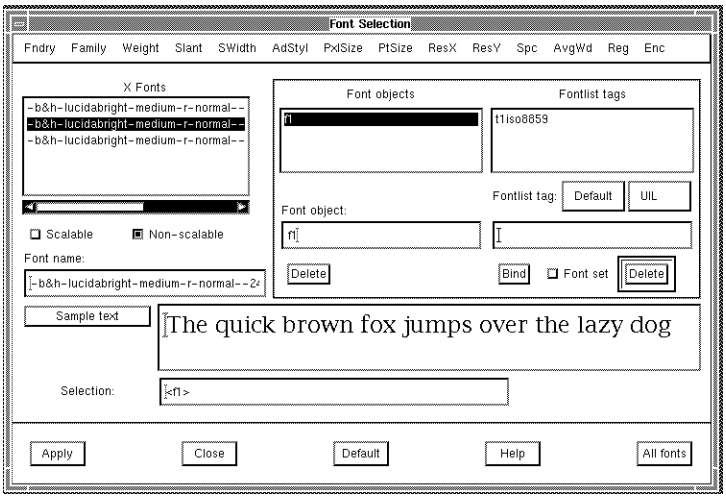


Figure 22-2 Initial Font in FontSet

6. **Select a font with jisx0208.1983 registry and bind it.**
   Xlib issues another warning of missing charsets.

7. **Select a font with jisx0201.1976 registry and bind it.**
   This time Xlib does not warn you because the FontSet is now complete.

Font object entries can be deleted and re-bound regardless of whether or not they are part of a font set. A FontList can have many entries which can be either FontSets or simple fonts. To achieve this, simply specify different Fontlist tags and set the Font set toggle as appropriate. The following example shows a single font object that has three entries: *t1*, *big* and *t2*. *t1* and *t2* are complete FontSets for the ja_JP.jis7 locale; *big* is a simple font.

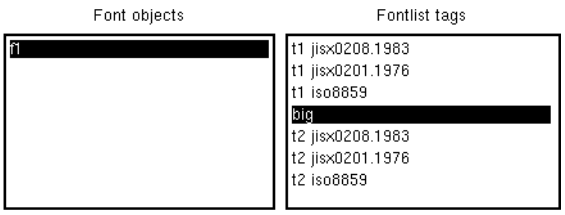


Figure 22-3 A Font Object with Three Entries.

22.4 Creating International Text

In order to clarify how internationalized text works, the following simple example shows how to display Japanese text on a label. The methods used by SPARCworks/Visual are then explained. To achieve equivalent results in your applications, you can apply similar techniques using the support provided by SPARCworks/Visual.

For a language such as Japanese, more characters can be displayed than there are keys on the keyboard, or numbers that can be represented by 8 bits. In addition, text displayed by Japanese symbols must be mixed with Roman numerals and Roman text. These problems are addressed by coding the text in a special way. Instead of the familiar ASCII mapping, *multi byte* text strings are used which contain escape sequences to denote changes in encoding. Multiple characters in the text signify a single character from the matching font.

To type these strings into an application obviously requires some additional support. X11R5 provides a mechanism known as *input methods* which are used to handle this special input. The MIT X release contains some sample implementations of input methods. The examples shown below should work with most input methods, whether supplied with MIT X, by your X vendor, or written by you.

In order to try out the examples, you must have the input system running. Assuming you have it correctly installed:

1. **Run the input method.**
2. **Run SPARCworks/Visual.**
   To build an application that displays a Japanese label:
3. **Create a simple hierarchy with a Label in it.**
4. **Pop up the Label's resource panel.**
   Notice that the resource panel is wider than usual and that there is an extra Text widget at the bottom. This Text widget is inserted by the input method to let you do your Japanese text input.
5. **Click in the labelString Text widget.**
6. **Delete the text content ("label").**
   Now you can enter your Japanese text into the Text widget at the bottom. Most input methods let you enter text in Kana phonetic symbols and convert each character to Kanji as you go. Here is a simple example:

7. **In the Text widget, type** `ni`
   The text entry now displays:

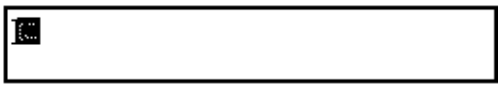


Figure 22-4 ni Displayed in Kana.

8. **Press <*Ctrl-W*>.**
   This is the conversion character. The Kana is converted to Kanji and displays as:

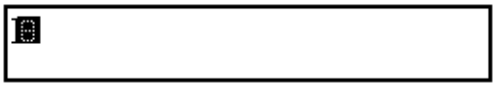


Figure 22-5 ni Displayed in Kanji.

9. **Type** hon **<*Ctrl-W*>.**
   A Kanji character for “hon” is displayed. However, it is not the Kanji character we want, since there are many possible Kanji characters for “hon”.

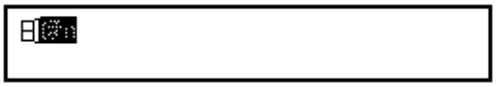


Figure 22-6 hon Displayed in Kanji.

To display the possible Kanji characters:

10. **Press <*Ctrl-W*> again.**
    This pops up a menu of possible Kanji characters.

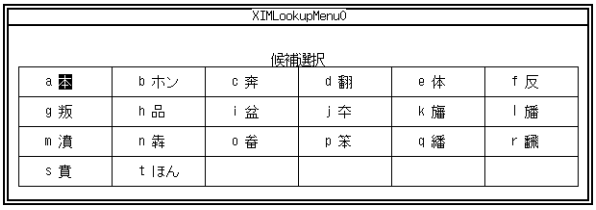


Figure 22-7 Possible Kanji Characters for hon.

11. Select one of the possible Kanji characters.
Use the arrow keys and space bar to make your selection. The Text widget now displays:

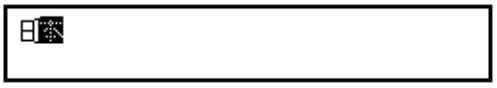


Figure 22-8 Two Kanji Characters.

12. Type go *<Ctrl-W>***.**
This is another Kana character that has many possible Kanji options. Use the popup menu again to choose the correct character.

The highlighting on the new character indicates that you are still editing it.

To indicate that you have finished with the character:

13. Type a space.
The Text widget now looks like Figure 22-9.

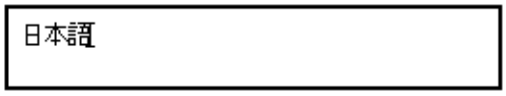


Figure 22-9 The Kanji for “Japanese language”.

14. Click on “Apply”.
The label now displays:

Figure 22-10 A Multi Byte String Displayed with the Wrong Font.

In order to display the text correctly, the label must have a font containing a FontSet that matches the current locale.

15. Create a font object and apply it to the label.
Use the procedure described in the previous section, using the default Fontlist tag.

Figure 22-11 The Label Displayed with a 24 Point FontSet.

16. Generate C code.

```
static void initialise_objects( Widget parent )
{
    char *from_s; /* For conversions */
    XrmValue from_value, to_value; /* Ditto */
    XmString temp_xmstring; /* For building XmString objects */
    if ( _xd_initialised ) return;
    _xd_initialised = 1;
```

/* This is a string representation of a font list with a FontSet on the default tag. The colon (:) at the end designates it as a FontSet. */

```
    from_s = "-jis-fixed-medium-r-normal--24-230-75-75-c-240-
       jisx0208.1983-0;-sony-fixed-medium-r-normal--24-230-75-75-c-
       120-jisx0201.1976-0;-b&h-lucidabright-medium-i-normal--24-0-
       100-100-p-0-iso8859-1:";
    from_value.size = strlen(from_s)+1;
    from_value.addr = (char *) XtMalloc ( from_value.size );
```

```
    (void) strcpy ( from_value.addr, from_s );
    XtConvert( parent, XmRString, &from_value, XmRFontList,
       &to_value);
    XtFree ( from_value.addr );
    font_resources.f1 = *(XmFontList*)to_value.addr;
}
void create_widget0 (Widget parent)
{
    ...
/* This is the string with all the escape sequences in it. */
    xmstrings[0] = XmStringCreateLtoR("\033$BF|
       \033(B\033$B%[\033(B\033$B8l\033(B ",
       (XmStringCharSet)XmFONTLIST_DEFAULT_TAG);
    XtSetArg(al[ac], XmNlabelString, xmstrings[0]); ac++;
    XtSetArg(al[ac], XmNfontList, font_resources.f1); ac++;
    widget5 = XmCreateLabel ( widget1, "widget5", al, ac );
    ac = 0;
    XmStringFree ( xmstrings [ 0 ] );
    ...
}
```

The default Japanese defaults file for SPARCworks/Visual make it possible to type Japanese text in the text value fields and the places where you can input an XmString. Input methods cannot presently be used in the XmString editor or the Font selector.

22.5 International Text Input

In the previous section, you used the internationalized text input to put the Japanese text into the label resource. This section describes how to make this text input work in your application.

The toolkit does most of the work for you. What you need to do is to invoke the mechanisms for any given Text widget. The rule is simple: if, upon creation, a Text widget has a FontList which includes a FontSet that matches the current locale, the Text widget makes a request to the enclosing Shell to establish a connection to the input method. The toolkit then takes over, diverting text input for that Text widget to the input method and returning to the Text widget the text sent back.

You can see from above that in order for your application to use an input method for a Text widget, all you have to do is to set the *fontList* resource for it at creation time. The simplest way to do this is to use SPARCworks/Visual to set the *fontList* resource on the appropriate widget. Although this works correctly for your generated application, there is a drawback. SPARCworks/Visual always creates the widget before resources are applied to it, even on reset or paste. Therefore, to see the input method working in the dynamic display window, you must force the font to be specified at creation time. This is best accomplished by setting the *textFontList* resource on a parent Bulletin Board or Shell widget. Obviously, this has the disadvantage that all the child Text widgets have the input method.

To see a Text widget working with an input method from within SPARCworks/Visual, use the following steps.

1. **Create a dialog with a Form child.**
2. **Select the Form and pop up the “Fonts” page of its resource panel.**
3. **Click on the “Text font” button.**
4. **Specify a font object with a FontSet attached to the default fontlist tag.**
5. **Create a Text widget child.**
6. **Add other widgets as required.**

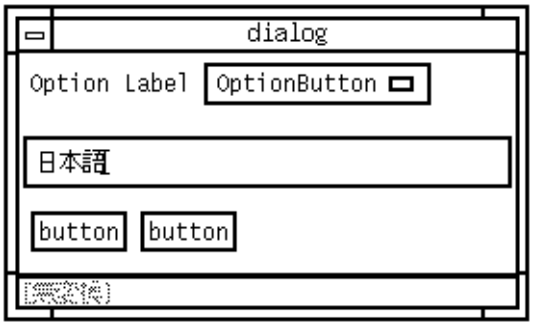


Figure 22-12 A Simple Dialog with Input Method Attached.

The default *main()* program includes a call to *XtSetLanguageProc()*, which initializes the locale handling routines. If you are writing your own *main()* program, you must call this routine if you are using any internationalization features.

22

Command Summary 23

23.1 Introduction

This chapter is a quick reference guide to SPARCworks/Visual commands. It includes:

- All menu commands in the main menu bar
- Some additional SPARCworks/Visual commands
- Instructions for accessing on-line help
- A table of keyboard accelerators

This chapter is designed as a quick reference for experienced users. More detail on the commands is found in the appropriate chapters of the *Tutorial* and in the on-line help.

Some menu items execute immediately, while others display a dialog where you must enter further information before anything happens. Commands that display a dialog have three dots (...) after the command name.

Many commands have keyboard accelerators which allow you to execute them with a single keystroke. The accelerators are listed in a table at the end of this chapter.

23

23.2 The File Menu

Commands on SPARCworks/Visual's File Menu control only design files. Design files are the principal SPARCworks/Visual files and, by convention, have the suffix *.xd*. They contain a design hierarchy (which may consist of multiple windows), the resource values set on the resource panels (including callbacks) and links.

23.2.1 New

Clears the construction area so that you can begin a new design. If you have not saved your work since the last change, you are asked if you want to save before the construction area is cleared.

23.2.2 Open...

Opens an existing design file. You are prompted for the name of the file. If you have not saved your work since the last change, you are asked if you want to save before the new file is read in to replace it.

23.2.3 Read...

Merges the contents of a file into your current design. Since all design files begin with a Shell widget, this adds one or more windows to your design.

Variable names of widgets must be unique across the entire design. When you combine two designs with "Read", any variable names in the file that duplicate names already in your design are silently removed and replaced with local names of the default form *widget*<*n*>.

To merge parts of windows from one design to another, use the "Copy to File" and "Paste from File" commands in the Edit Menu.

23.2.4 Save

Saves the current design using its previously specified filename. If your current design is new and has never been saved before, you are prompted for a filename as in "Save as...".

23.2.5 Save as...

Saves the current design under any filename. The SPARCworks/Visual file browser is displayed for you to specify a filename. Use this command when saving a new design file for the first time.

23.2.6 Print...

Prints the current hierarchy to a printer or a file. To print to a file, click on the "File" toggle and enter the filename in the text box under "File". To send to a printer, click on the "Command" toggle and enter the command, such as *lpr*, in the same text box, which is now labeled "Command". Because the output is Postscript, a Postscript printer or viewer is required.

The option menus in the Print Dialog let you specify the page size, orientation, pages and scale. In the "Scale" option menu, the reduced scale option prints the diagram two-thirds of its actual size. Note that if the "Scale to fit" option is not selected, the diagram prints on as many pages as required. The "Pages" option menu lets you print all the hierarchies in your design if your design contains more than one window, or just the hierarchy currently displayed in the construction area.

Selecting the "Show names" toggle lets you print the widget names of the widgets. Selecting the "Print headings" toggle puts a border around the hierarchy and prints a title, which you can specify in the "Title" text field. The title can have only a single line of text.

23.2.7 Exit

Leaves SPARCworks/Visual. If you have not saved your work since the last change, you are asked whether you want to save before SPARCworks/Visual exits.

23.3 The Edit Menu

The Edit Menu has commands for editing the design hierarchy. All Edit options operate on the currently selected widget including all its children in the hierarchy.

23.3.1 Undo

Not yet available.

23.3.2 Cut

Removes the currently selected widget from the design hierarchy and copies it into the SPARCworks/Visual clipboard.

23.3.3 Copy

Copies the currently selected widget to the clipboard.

23.3.4 Paste

Copies the contents of the clipboard into the hierarchy as a child of the currently selected widget.

If the clipboard is empty, or if the widget in the clipboard cannot be made a child of the currently selected widget, then “Paste” is disabled.

23.3.5 Clear

Deletes the currently selected widget. Cleared widgets are not put into the clipboard and therefore cannot be pasted.

23.3.6 Copy to File...

Copies the currently selected widget to a clipboard file. You are prompted for the file name you want to use.

23.3.7 Paste from File...

Copies the contents of a clipboard file into the hierarchy as a child of the currently selected widget. You are prompted for the name of the clipboard file. If the widget at the root of the clipboard file hierarchy cannot be made a child of the currently selected widget, this operation shows an “Invalid hierarchy” error message.

"Paste" always makes the pasted widget the last child of the selected widget. To reorder children of a widget, use dragging, discussed below.

23.3.8 Search...

Displays the search dialog. This is explained fully in the *Search* Chapter on page 7-1.

23.3.9 Move By Dragging

You can move a widget to another position in the hierarchy by dragging its icon with mouse button 1. This is the equivalent of using "Cut" followed by "Paste".

23.3.10 Copy By Dragging

You can copy a widget to another location in the tree by dragging its icon with mouse button 2. This is the equivalent of using "Copy" followed by "Paste".

23.4 The View Menu

The View Menu has commands that affect the appearance of the main SPARCworks/Visual window. Each command in this menu is a toggle and can be turned on or off. The View options only affect the appearance of the SPARCworks/Visual screen, not your design.

23.4.1 Show Widget Names

Displays the name of each widget in your design underneath the widget icon in the design area. The name shown is the unique variable name of the widget.

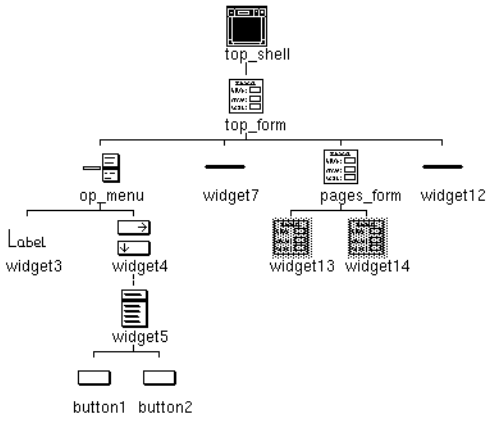


Figure 23-1 Show Widget Names

23.4.2 Show Dialog Names

Displays the widget name assigned to each Shell in your design underneath its icon in the window holding area. “Show dialog names” is particularly useful for navigating in designs that have multiple windows. Note that the Shell icon shrinks to allow room for the dialog name.

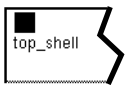


Figure 23-2 Show Dialog Names (Window Holding Area Shown)

23.4.3 Left Justify Tree

Changes the appearance of the hierarchy in the design area from a centered tree with branches spreading in both directions to a left-justified one with its branches spreading to the right.

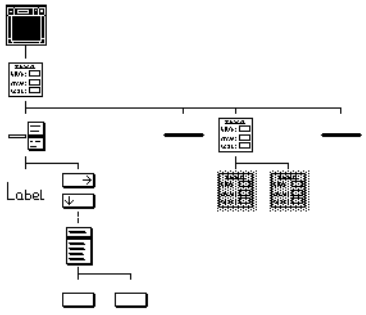


Figure 23-3 Left Justify Tree

23.4.4 Shrink Widgets

Reduces the size of widgets in the construction area. Use this option when the hierarchy you are building becomes large and you want to see the whole or a large proportion of the hierarchy. The widgets are shrunk to a uniform small square. As with the other View options, your actual design is not affected.

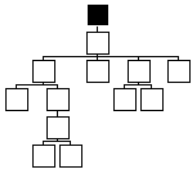


Figure 23-4 Shrink Widgets

23.4.5 Annotations

Annotates widgets in the construction area according to specified criteria. You can request that the tree be annotated if the widget has:

- Search
- Callbacks
- Methods
- Links
- A pre-create prelude

- A pre-manage prelude

You can also request that the tree be annotated to indicate the results of the previous search operation. To turn on an annotation, select the appropriate toggle from the Annotations pullright menu. You can tear off the pullright menu to use as an annotation reference, as shown in Figure 23-5. Annotations are explained more fully in the *Annotations* section of the *Search* chapter on page 148.

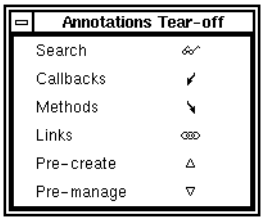


Figure 23-5 "Annotations" Tear-off Menu

23.4.6 Structure Colors

Color-code widgets in the construction area. When this option is on, separate colors are used to indicate those widgets in the hierarchy which have been designated as functions, data structures, C++ classes, or children only (via the "Code generation" page of the Core resource panel). These designations are discussed in the *Structure Colors* section of the *Structured Code Generation and Reusable Definitions* chapter on page 210.This option has no effect if all your widgets are default structures.

To turn on this option, click on the "Show colors" toggle in the pullright menu. You can tear off the pullright menu to use as a color reference, as shown in Figure 23-6. To tear off the menu, click on the dashed line.

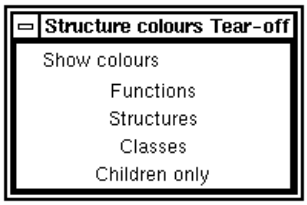


Figure 23-6 "Structure Colors" Tear-off Menu

23.5 The Palette Menu

The Palette Menu contains options which change the appearance of the Widget Palette.

23.5.1 Layout

Specifies whether the Palette is to be displayed with icons, labels, or both. Click on the appropriate toggle button in the pullright menu.

23.5.2 Separate Palette

Displays the Widget Palette in a separate sub-dialog. The SPARCworks/Visual Window will then have more room for displaying the Design Area. You can select from and add to the separated Widget Palette in the normal manner.

23.5.3 Show Palette

Re-displays and raises the separated Widget Palette to the top of the window stack. This is a useful option when you have closed down the separated Widget Palette, or have lost it underneath all your separate application dialogs. This option is disabled if a separate palette has not been chosen.

23.5.4 Define

Designates the currently selected widget as a widget definition. This option is a short-hand method of creating widget definitions. The currently selected widget (if specified as a class) will form a new definition using some default configuration data deduced by context.

23.5.5 Edit Definitions

Displays the Edit Definitions panel which allows you to turn portions of your widget hierarchy into reusable objects selectable from the Widget Palette.

23.6 The Widget Menu

The Widget Menu has commands that apply to individual widgets. All these commands apply to the currently selected widget in the hierarchy.

23.6.1 Resources...

Displays the resource panel for the currently selected widget. You can also display the resource panel for most classes of widgets by clicking twice on the widget's icon in the design hierarchy.

Resource panels are discussed in detail in the *Using the Resource Panels* chapter. Some individual resource settings are also discussed in the *Widget Reference* chapter.

23.6.2 Core Resources...

Displays the Core resource panel, where you can set resources inherited from the Core, Primitive and Manager superclasses. Core resources include foreground and background colors and whether or not a widget is sensitive to events.

The "Code Generation" page of the Core resource panel lets you specify individual widgets as local, global, or static, regardless of whether they are explicitly named. If you are using C++, this page also lets you specify public, private, or protected status for the selected widget.

23.6.3 Layout...

Displays the Layout Editor. This option is available for the three classes of layout widget - the Form, BulletinBoard and DrawingArea. You can also display the Layout Editor for these widget classes by double-clicking on the widget icon in the design hierarchy. For more information, see the *Layout Editor* chapter.

23.6.4 Constraints...

Displays the constraints panel for any widget that is a child of a constraint widget - a PanedWindow or Form. In the case of the Form, this panel should usually be used only to view the constraint resources rather than to reset them, because Form attachments can be set more reliably in the Layout Editor. The Constraints panel is discussed in the *Using the Resource Panels* chapter.

23.6.5 Translations...

Lets you specify translations for the currently selected widget. A translation is a key sequence that is mapped to a specified action on the widget. Translations can be one of two kinds, “Override” or “Augment”. “Override” translations override any translations already set on the given key sequence. “Augment” translations are added to the list of translations already set.

If you use the “Replace” toggle on this dialog, the translations you type into the upper box replace any translations already set on this widget. When “Replace” is set, you cannot use the “Augment” text box.

For more information, see the *Translations* chapter.

23.6.6 Code Preludes...

Displays a dialog that lets you add pieces of code to the code generated for the currently selected widget. The most commonly used types are pre-create and pre-manage preludes. Pre-create preludes are inserted just before the widget is created. A typical use of pre-creation preludes is to set widget resources that can only be set at widget creation time.

Pre-manage preludes are inserted just before the widget’s callbacks are added. They are typically used for setting up client data for the callbacks.

Pre-create and pre-manage preludes are discussed in the *Generating Code* chapter. Code preludes for private, protected and public methods pertain to C++ widgets. For more information, see the *Adding Class Members* section of the *C++ Code Tutorial* chapter.

23.6.7 Method Declarations

Allows you to insert declarations for application-defined methods into a C++ class.

23.6.8 Reset

Destroys the currently selected widget and all its children and recreates them. This is useful after you set certain resources on the resource panels or in the Layout Editor. Sometimes creating a widget and then changing a resource value gives a different result from creating the widget initially with the changed value. Therefore, if your dynamic display does not reflect changed resource settings the way you expected, resetting widgets can often solve the problem.

23.6.9 Edit Links...

Allows you to you set up or remove links from widgets. Links are pre-defined Activate callbacks and can be set only on button-type widgets. Links can show, hide, enable, or disable any widget. You can set multiple links on one button.

23.6.10 Fold/Unfold

Hides / displays the children of a selected widget. Use this toggle to save space when your design hierarchy becomes too large to fit in the construction area. When the selected widget is folded, its children are hidden, though they are not removed from the design. The folded widget's icon in the tree is grayed out.

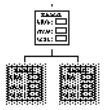


Figure 23-7 Folded Widgets

23.6.11 Definition

Turns the currently selected hierarchy of widgets into a reusable object definition. The definition can be inserted into the widget palette and then selected just like any other palette widget.

23.7 Widget Name and Variable Name

Two text boxes at the top of the main SPARCworks/Visual screen let you specify a variable name and widget name for the currently selected widget.

The variable name is the name used to identify that widget in the generated code. Variable names must be unique. If you do not specify a variable name, SPARCworks/Visual assigns a unique name of the form *widget<n>*. The number *n* is not guaranteed to stay the same every time you open the design file; therefore, you should never refer to a default name in your code.

Explicitly named widgets are global in scope; those not explicitly named are local, unless you change this status on the Core resource panel.

The widget name can be the same as the variable name or different. Widget names do not have to be unique in a design. Groups of widgets that share a widget name also share any resource settings that are generated into the X resource file. See the *Generating Code* chapter for more information.

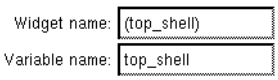


Figure 23-8 Text fields for Widget and Variable Name

23

23.8 The Module Menu

The commands on this menu let you specify lines of code that are inserted into the primary module of generated code at specified points. Unlike code preludes, which are attached to individual widgets, module preludes and headings apply to the whole module.

23.8.1 Module Prelude...

Displays a dialog box that lets you enter lines of code to be entered at or near the beginning of the generated code file.

The Module Heading is inserted at the beginning of the primary code module and at the beginning of the stubs file, if generated. The Module Heading is typically used for SCCS ids, versions and other identifying information.

The Module Prelude is inserted after the generated SPARCworks/Visual *#include* directives, if any, and is typically used for *#include* or *#define* directives or *extern* declarations required by your code preludes.

The Resource Prelude is inserted after the SPARCworks/Visual generated comment in the X resource file. It can be used to set global application resources or to #include another resource file.

23.8.2 Help Defaults

Displays a dialog which allows you to specify defaults for the help system.

The Default Path field denotes the path used for help document names. It is also used in SPARCworks/Visual as a fallback location for generated code.

The Default Document field is used if a marker for help is set but no document is specified for the widget or any of the widget ancestors.

The Path Resource and Path environment variable fields allow you to provide a dynamic Help document context: code will be generated so that any values specified will be used to override any setting of the Default path: the runtime environment variable takes precedence over the static resource setting.

The Default Translation field will cause a translation to be added to every widget which has a help marker; the associated Action for this translation will always be Help. For example, a translation ‘Ctrl<key>A’ will mean that unless specified otherwise Control A will cause the Help message window to be displayed for a widget which has a help marker.

The ‘Always own window’ toggle, when set, causes code to be generated so that each access to the Help System will result in a new Help Document window.

23.8.3 Windows compliant

This toggle (only present when SPARCworks/Visual is in Windows mode) is used to indicate that the design is Windows compliant; that is, it is possible to generate MFC code for it. The toggle is set to indicate that the design is compliant. If the design becomes non-compliant, because an old design is read in or by means of cut and paste, the toggle is unset and the Windows Compliance Failure dialog is displayed (see the *Compliance Failure* section of the *Cross Platform Development* chapter). This toggle can be reproduced on the toolbar.

23.8.4 Application class...

The MFC C++ flavors use an instance of the CWinApp class to represent the application. By popping up the Application class dialog you can change the base class name, the class name and the instantiate as name for this instance. This item is only present when SPARCworks/Visual is in Windows mode.

23.9 The Generate Menu

When your design is complete, the Generate Menu can be used to generate code for it in three languages: C, C++, or UIL. It can also generate an X resource file containing resource values explicitly set in your design. Whichever language you use, you should first generate the code files and then generate an X resource file to complement the code files. The procedure for generating code, linking and running is discussed in the *Generating Code* chapter.

All commands on this menu use the same dialog. The Generate Dialog includes a file browser similar to that used by the File Menu commands such as “Save” and “Open”.

The Generate Dialog also has several toggle switches that control generation of code segments (in code files only), types of resources (in code and X resource files) and a masking policy switch that works in conjunction with the resource type toggles. These switches, shown in Figure 23-9, are explained in the *Generating Code* chapter.

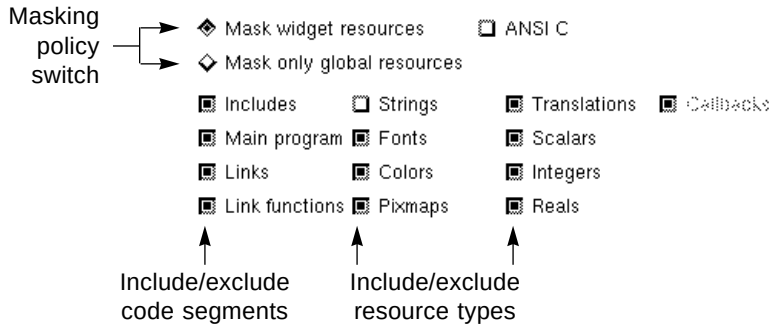


Figure 23-9 Toggles in Generate Dialog

23.9.1 C

Displays the C tear-off or pullright menu.

23.9.2 C++

Displays the C++ tear-off or pullright menu.

23.9.3 UIL

Displays the UIL tear-off or pullright menu.

23.9.4 X Resource File...

Generates an X resource file from your design. By convention, generated X resource files have the suffix *.res*. They must be copied to the filename expected by X in order to take effect.

You must use the same application class name for the X resource file that you used when you generated the primary module, or X will not be able to associate your resources with the generated application.

23.9.5 Windows Resources...

Generate a Windows resource file and the associated bitmap (*.bmp*) and icon (*.ico*) files. The Windows resource file typically has the extension *.rc*. This option is only present when SPARCworks/Visual is in Windows mode.

23.9.6 Makefile...

Generates a Makefile to build your application. Before you generate a Makefile, you must generate the code files that you want to include in it. Generating the code files sets the names for these files in the design file. Until you do this, the Makefile generation feature doesn't know the names of these files and can't add them to the Makefile.

23.10 Tear-Off Menus

The procedure for generating code is basically the same regardless of your choice of language. The pullright menus for C, C++ and UIL can be torn off into a separate window by clicking on the dashed line at the top of the menu. You can also invoke commands from the pullright menu in the usual way without tearing off the menu. After you have displayed or torn off one of these menus, you can select one of the following commands.

23.10.1 C, C++, or UIL...

Generates the primary code module in the language of your choice.

23.10.2 Stubs or C++ Stubs...

Generates a stubs file, containing function declarations and empty braces for any callback functions you have designated in your design. Use of a stubs file ensures that your callbacks are declared with the proper syntax. You can then write code between the braces to add functionality.

23.10.3 Externs, C++ Externs, or Externs for UIL...

Generates a header file that declares all global objects and functions in your design. For convenient access to your global widgets and defined objects, *#include* this file in your stubs or callbacks file.

23.10.4 C, C++, or UIL pixmaps...

Generates static declarations of all pixmaps in your design into a separate file. This file is meant to be included as a header file and by convention has the suffix *.h*.

23.10.5 C for UIL...

Generates a C file that performs those functions in your application not covered by the UIL file. This command applies only to UIL applications. It exists because UIL does not have all the capabilities offered in SPARCworks/Visual. You must first generate a UIL file and specify the name of that file in the "Uid File" text field.

23.11 The Help Menu

The Help Menu offers the most general help messages.

To get specific help, click on the "Help" button on any dialog box. SPARCworks/Visual's help is organized as a hypertext network. Each help screen displays a list of related topics, as shown in Figure 23-10. To display help for one of the related topics, click on the topic, then click on the "Follow link" button.

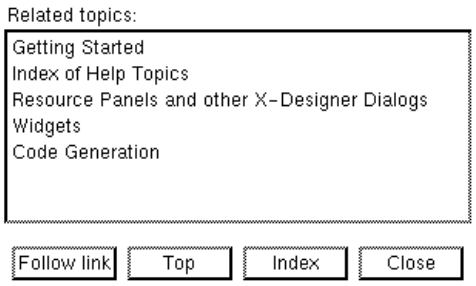


Figure 23-10 Links to Related Topics in Help Screen

Click on “Top” to go to the top help screen. Click on “Index” for a complete list of help topics.

23.11.1 About SPARCworks/Visual

Displays the SPARCworks/Visual copyright screen and the version of the software.

23.11.2 Palette Icons...

Displays a screen with pictures of all the Motif widget icons. You can click on any of these icons to get information about that widget class. The Palette Icons screen can be iconified independently of the main SPARCworks/Visual screen.

23.11.3 Help...

Displays the top-level help screen in the hypertext index. This screen offers a very general help message and the opportunity to follow links into more specific subjects.

23.12 Keyboard Shortcuts

The following table lists the keyboard accelerators for the SPARCworks/Visual commands.

Menu	Command	Accelerator
File	New	Control+N
	Open...	Control+O
	Read...	Control+R
	Save	Control+A
	Save as....	Control+V
	Print...	Control+P
	Exit	Control+E
Module	Module Prelude...	Control+F9
Edit	Cut	<keypad>Cut
	Copy	<keypad>Copy
	Paste	<keypad>Paste
	Clear	Control+F1
View	Show widget names	Control+W
	Show dialog names	Control+D
	Left justify tree	Control+L
	Shrink widgets	Control+F3
Widget	Resources...	Control+F4
	Core resources...	Control+C
	Constraints	Control+F5
	Translations...	Control+F6
	Code preludes...	Control+F7
	Reset	Control+T
	Edit links...	Control+F8
	Fold/unfold	Control+F

Menu	Command	Accelerator
Generate	C...	Alt+C
	C++...	Alt+<Plus>
	UIL...	Alt+U
	C for UIL...	Alt+L
	X Resources...	Alt+X
	Makefile...	Alt+K
Pixmap Editor	Undo	Alt+Backspace
	Cut	<keypad>Cut
	Copy	<keypad>Copy
	Paste	<keypad>Paste
Layout Editor	Widget Names	Control+W
	Class Names	Control+N
	Edge Highlights	Control+E
Help	About SPARCworks/Visual...	Control+F10
	Help...	F1 or HELP

23

Widget Reference 24

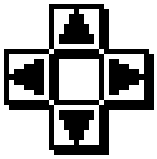
24.1 Introduction

This chapter provides a brief description of each of the widgets in the widget palette. It also describes some of the quirks of each widget and gives hints for their effective use. Only basic information is given here. For a full description of each widget, including all its resources, see the *Motif Programmer's Reference Manual*.

Each widget description starts with a list of resources grouped by page in the resource panel. Core resources are not listed to avoid repetition. Resources in **bold** typeface are frequently set and so are of interest to users regardless of their level of expertise. Resources in normal typeface are less commonly used and you may require more knowledge to use them effectively. Resources that are in *italics* are not effective but still appear on the resource panel.

Note that if you invoke SPARCworks/Visual using the command small_visu, the widget icons are smaller and slightly different from those shown here.

24.2 ArrowButton



Settings	Callbacks	Toggles
Direction	**Activate**	Widget
	Arm	Gadget
	Disarm	

The ArrowButton widget provides a button with an arrow on it instead of a text label. The arrow can point up, down, left, or right. Choose one of the four directions by clicking on the appropriate picture on the ArrowButton resource panel which is shown in Figure 24-1.

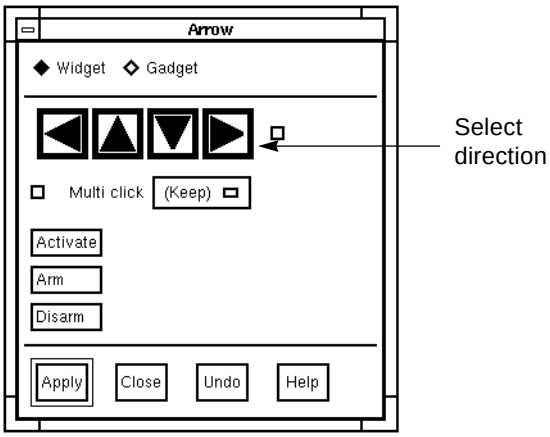
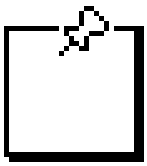


Figure 24-1 ArrowButton Resource Panel

Unlike other button widgets, ArrowButtons are not derived from the Label and cannot display text or pixmap labels.

24.3 BulletinBoard



Display
Title
Margin height
Margin width
Horizontal spacing
Vertical spacing
Fraction base
Cancel button
Default button

Settings
Dialog style
Resize policy
Shadow
Allow overlap
Auto unmanage
Default position
No resize
Rubber positioning

Fonts
Text font
Button font
Label font

Callbacks
Focus
Map
Unmap

The BulletinBoard widget is the most basic container widget. It is most commonly used internally by Motif to implement other container or composite widgets such as the Form, SelectionBox and MessageBox. These derived widgets are often more useful than the BulletinBoard itself.

As a container widget, the BulletinBoard does not impose any particular layout on its children. It provides absolute positioning, margin constraints and lets you specify whether the widgets inside are allowed to overlap or not. Resizing the BulletinBoard does not move or resize the widgets in it.

The BulletinBoard is most useful for transient dialogs that are not meant to be resized. For resizable dialogs, use a Form or DialogTemplate. You can also use a BulletinBoard for cases where complicated positioning is required and C code is to be written for this purpose.

Clicking twice on a BulletinBoard in the construction area displays the Layout Editor. Only the Move and Resize options of the Layout Editor can be used with a BulletinBoard. Attachments between widgets and position attachments are not available. For more flexible layout options, use a Form widget.

To display the resource panel, select “Resources...” from the Widget pulldown menu.

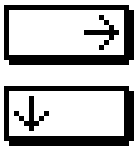
If a BulletinBoard is the child of a Shell, the BulletinBoard’s “Title” resource is used as the title of the Shell’s window.

Note that the Title, Dialog style, Default position and No resize are disabled if the BulletinBoard is a child of the Form.

The “Auto unmanage” resource, when set to “Yes” makes the dialog erase whenever you click on a button child of a BulletinBoard. This behavior, which is the default behavior in Motif, is useful for transient dialogs but can be confusing in main windows. SPARCworks/Visual explicitly sets this resource to “No” for the BulletinBoard and two of its derivatives, the DialogTemplate and the Form. With other BulletinBoard derivatives, SPARCworks/Visual does not override the default “Yes” setting and so the dialog does erase if you click any button. To restore your dynamic display, reset the Shell or select the Shell icon in the window holding area.

The “Default position” resource controls how the position of the window on the screen is determined. If you set this resource to “No” on a BulletinBoard (or derivative) that is a child of a Shell, the window is displayed in the position determined by the *x* and *y* resources of the BulletinBoard, not those of the Shell. As this behavior is dependent on the window manager, it may not be consistent.

24.4 CascadeButton



Display
Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select pixmap
Select insensitive pixmap

Toggles
Widget
Gadget

Callbacks
Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Margins
Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Keyboard
Accelerator
Accelerator text
Mnemonic
Mnemonic charset
Mapping delay

Settings
Type
Resize
Push button
Shadow
Fill on Arm
Fill on select
Indicator on
Indicator type

The CascadeButton widget is used to display a menu. It is derived from the Label and shares its resource panel. A CascadeButton can only be used as the child of a MenuBar or Menu. When it is the child of a MenuBar, a pulldown menu is displayed and when it is the child of a Menu, a pullright menu is displayed. Sample hierarchies showing these specific uses of the CascadeButton are located in the Menu widget description.

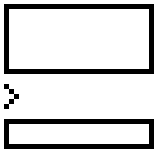
The only permissible child of a CascadeButton is a Menu. When a user clicks on a CascadeButton, a menu is displayed.

In Motif, a Menu is not technically a child of a CascadeButton but of the button's parent. To indicate this, the connection between a CascadeButton and its menu is drawn with a dotted line instead of the normal solid line.

You can set keyboard mnemonics for CascadeButtons to let the user navigate through the menus without using the mouse.

Note that the Mapping delay resource can be used only when the Button is used to instigate a pullright menu.

24.5 Command



Display
No match string
Pattern
Max history items
Command
History item count
Text columns
Directory mask
Directory

Settings
Dialog type
Minimize buttons
Must match
File type
Work area placement

Labels
Apply label
OK label
Cancel label
Help label
List label
Prompt string
Prompt string
Directory label

Callbacks
Apply
Cancel
OK
No match
Command changed
Command entered

The Command widget is a composite widget used to select a command from a scrollable history list of commands. Commands can be typed into a text area at the bottom of the widget. When a command is entered, it is added to the end of the history list. A Command is derived from the SelectionBox and shares its resource panel. It also inherits some BulletinBoard resources. To display the BulletinBoard resource panel, click on “Bulletin Board Resources” in the resource panel.

A Command contains a ScrolledList widget for the command history region, a Label widget for the command line prompt and a Text widget for the command entry region. These components are contained in a BulletinBoard widget that is not visible in the design hierarchy. You can change the default resource settings for the component widgets but you cannot delete them. To change the prompt, change the “Prompt string” resource in the resource panel of the Command, not in the resource panel of its Label child.

A Command is usually used in a Shell or MainWindow.

You can add multiple children to a Command. The first child becomes the work area. This can be a container widget containing additional widgets. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the Command widget’s hierarchy as shown in Figure 24-2. The additional children can include a MenuBar and any number of PushButton widgets.

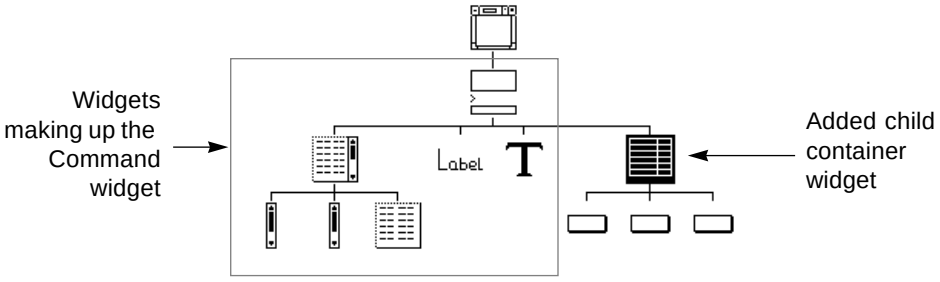
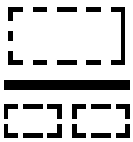


Figure 24-2 Command Widget Hierarchy

24.6 DialogTemplate



Display
Message text
OK label
Cancel label
Help label
Symbol pixmap

Settings
Default button
Dialog type
Alignment
Minimize buttons

The DialogTemplate widget is usually used as the child of a Shell for a broad range of dialogs. It provides a standard layout that includes, from top to bottom, a menu bar, a work area, a Separator, and a button box.

The DialogTemplate is a specially configured MessageBox and shares its resource panel. It also inherits some BulletinBoard resources. To display the BulletinBoard resource panel, click on “Bulletin Board Resources” in the resource panel.

The Separator is a component part of the DialogTemplate. You must add the other elements of the standard layout if you want them: for example, a MenuBar, any type of widget for the work area and buttons of any type for the button box, as shown in Figure 24-3. The work area can be a container widget, such as a Form, with children. The DialogTemplate always arranges its children in the standard order from bottom to top, regardless of the order in which you add them to the hierarchy.

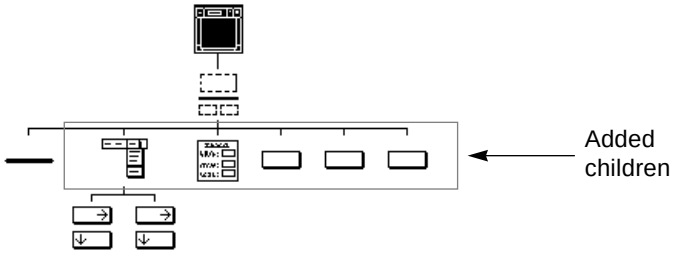
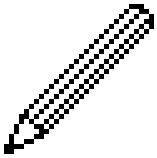


Figure 24-3 Standard Hierarchy Using the DialogTemplate

The areas of the standard layout are constrained to be the same width, with the buttons in the button box evenly spaced in one or more rows. The buttons are automatically rearranged as needed when the window resizes.

24.7 DrawingArea



Margins
Width
Height

Resize policy

Callbacks
Expose
Input
Resize

The DrawingArea widget provides an area in which an application can display output graphics. For example, the design hierarchy in the main SPARCworks/Visual window is drawn in a DrawingArea contained in a ScrolledWindow.

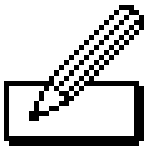
Clicking twice on a DrawingArea in the construction area displays the Layout Editor. Only the Move and Resize options of the Layout Editor can be used with a DrawingArea. Attachments between widgets and position attachments are not available.

To display the resource panel, select “Resources...” from the Widget pulldown menu.

Although a DrawingArea can have any number and types of children, it is not very useful for managing the geometry of other widgets. Other container widgets such as the Form should be used for this purpose instead.

SPARCworks/Visual cannot help you with drawing in the drawing area. To do this, you must write C code containing X graphics calls. This code is normally put in the “Expose” callback.

24.8 DrawnButton



Display
Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select color
Select pixmap
Select insensitive pixmap

Margins
Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow

Keyboard
Accelerator
Accelerator text
Mnemonic
Mnemonic charset
Mapping delay

Settings
Type
Resize
Push button
Shadow
Fill on Arm
Fill on select
Indicator on
Indicator type
Multi click
Set
Visible when off

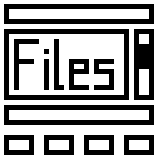
Callbacks
Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Toggles
Widget
Gadget

The DrawnButton widget is similar to a PushButton except that its face must be drawn by the application instead of being drawn automatically. It can be used to provide a button that has a context-sensitive appearance. The DrawnButton is derived from the Label widget and shares its resource panel.

To display a picture on a button, it is usually easier to use a PushButton with a pixmap for the image. Drawing the picture on a DrawnButton requires writing C code containing X graphics calls, which is normally put in the “Expose” callback.

24.9 FileSelectionBox



Display
No match string
Pattern
Max history items
Directory spec
Visible item count
Text columns
Directory mask
Directory

Settings
Dialog type
Minimize buttons
Must match
File type
Work area placement

Labels
Apply label
OK label
Cancel label
Help label
File list label
Selection label
Filter label
Directory label

Callbacks
Apply
Cancel
OK
No match
Command changed
Command entered

The FileSelectionBox widget is a composite widget that lets users browse through the file system and select a file. The file browser in SPARCworks/Visual is an example of a FileSelectionBox. The Generate Dialog is a FileSelectionBox with a work area child. The FileSelectionBox is derived from the SelectionBox and shares its resource panel.

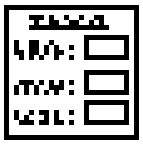
The FileSelectionBox combination includes two ScrolledLists, two TextFields, four Labels, a Separator and four PushButtons, which are gadgets. These components are contained in a BulletinBoard widget that is not visible in the design hierarchy. To display the resources inherited from the BulletinBoard, click on “Bulletin Board Resources” in the resource panel.

While a FileSelectionBox can be used anywhere that a BulletinBoard can, it is usually placed in a Dialog Shell that is popped up for file selection.

To change the labels of button or label widgets from the defaults, change the resources in the resource panel of the FileSelectionBox, not in the resource panels of the individual widgets.

You can add multiple children to a FileSelectionBox. The first child becomes the work area. This can be a container widget containing additional widgets. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the FileSelectionBox widget’s hierarchy. The additional children can include a MenuBar and any number of PushButton widgets.

24.10 Form



Display
Title
Margin height
Margin width
Horizontal spacing
Vertical spacing
Fraction base
Cancel button
Default button

Settings
Dialog style
Resize policy
Shadow
Allow overlap
Auto unmanage
Default position
No resize
Rubber positioning

Fonts
Text font
Button font
Label font

Callbacks
Focus
Map
Unmap

The Form widget is a container widget that provides both absolute and relative positioning of its children widgets. It is commonly used to lay out widgets in a dialog, either as a child of a Shell or as the work area in a DialogTemplate or similar widget.

The layout of widgets in a Form is specified by using attachments on children of the Form. Different types of attachments let you specify different types of spatial relationships such as a fixed location within the Form, a relative location within the Form, or a fixed distance between widgets. These capabilities allow considerable flexibility and reliable behavior when widgets or windows are resized. SPARCworks/Visual lets you specify these attachments interactively in the Layout Editor. To display the Layout Editor, click twice on a Form in the construction area. For more information, see the *Layout Editor* chapter.

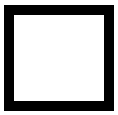
You can view the attachments set on any child of a Form by using the Constraints panel. Select any child of the Form, pull down the Widget Menu and select “Constraints...”. Use of this panel is described in the *Using the Resource Panels* chapter.

The Form is derived from the BulletinBoard and shares its resource panel. To display the resource panel, select “Resources...” from the Widget Menu.

The “Auto unmanage” resource, if set to “Yes”, makes the dialog erase whenever you click on a button child of a Form. This behavior, the default behavior in Motif, is useful for transient dialogs. However, because a Form is often used for main windows, SPARCworks/Visual explicitly sets this resource to “No” in the case of the Form. Set it to “Yes” if you want a Form to auto unmanage.

For more details, see BulletinBoard.

24.11 Frame



Display
Margin width
Margin height
Title widget
Title spacing
Shadow type
Title alignment (horizontal)
Title alignment (vertical)

The Frame widget is used to provide a border, possibly with a title, around a widget that otherwise has none, to enhance the border of a widget that already has one, or to create a border around a group of widgets. A Frame can be used to provide three-dimensional effects, like indenting a DrawingArea.

A Frame can have two children. The first is placed inside the Frame and the second (which is optional) is used as a title. The second child is usually a Label.

To create a border around a group of widgets, they must be placed in a container widget such as a RowColumn or a Form, that is the child of a Frame, as shown in Figure 24-4.

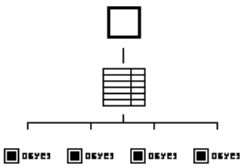


Figure 24-4 Hierarchy Showing a Frame Widget as a Border

The Frame sizes itself to match the size of its children.

24.12 Label

Display
Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select color
Select pixmap
Select insensitive pixmap

Margins
Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Keyboard
Accelerator
Accelerator text
Mnemonic
Mnemonic charset
Mapping delay

Settings
Alignment
Type
Resize
Push button
Shadow in
Fill on arm
Fill on select
Indicator on
Indicator type
Multi click
Set
Visible when off

Callbacks
Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

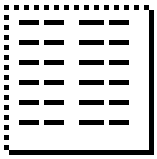
Toggles
Widget
Gadget

The Label widget provides a static display area for text or pixmap images. Labels are commonly used to display descriptive text strings or icons or logos. Labels can be placed in menus to provide unselectable titles for groups of menu items. Several widgets, such as PushButtons and ToggleButtons, are derived from Label and share the same resource panel.

A string in a Label can use multiple lines and fonts. Multiple fonts are supported using the Compound String Editor, which is discussed in the *Compound String Editor* chapter.

If you set a pixmap for a Label, the Label does not display it until you also change the “Type” setting to “Pixmap”.

24.13 List



Display
Margin width
Margin height
Spacing
Visible items
Top item
Double click interval
Font

Settings
Automatic selection
Selection policy
Size policy
Scroll bar display

Callbacks
Browse
Default
Extended
Multiple
Single

Items
Item

The List widget is used to display a list of text items, one or more of which can be selected, depending on the setting of the “Selection policy” resource.

For a scrolling list of text items, use a ScrolledList widget, a composite widget that contains a List widget.

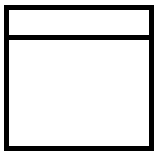
The Items page of the List resource panel lets you add items to the list so you can see what the list looks like. To add an item, enter its text into the “Item” resource box and select “Add”. To remove an item from the list, enter its text into the “Item” resource box and select “Remove”. While items must be added in the order in which you want them to appear, they can be deleted in any order.

To see additional items, change the “Visible items” resource.

The Motif toolkit provides a large number of functions for manipulating Lists such as adding, removing and replacing items. For further details, see the Motif documentation.

Note that each item in a List is a compound string (XmString). It is therefore theoretically possible to use different fonts for different items, or for different parts of a single item. In practice, limitations of the Motif toolkit make this inadvisable.

24.14 MainWindow



Scrolled window margins
Width
Height
Spacing

Callbacks
Traverse obscured

Main window margins
Width
Height

Scroll bar display
Scroll bar placement
Scrolling policy
Visual policy
Show separators
Command location
Message window

The MainWindow widget is derived from the ScrolledWindow and shares its resource panel. It provides a standard layout for an application's primary window. This standard layout includes, from top to bottom:

- A menu bar
- A command area with history, a prompt and an input area
- A work area
- A message area

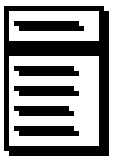
The MainWindow is a composite widget with three Separators and two ScrollBars. You must add the widgets for each element in the standard layout. Use a MenuBar for the menu bar and a Command for the command area. A Text or TextField is usually used for the message area. You must give the message area widget a variable name and specify that name as the "Message window" resource of the MainWindow.

The work area can be almost any other kind of widget. It can be a container widget with other widgets as children. A MainWindow ordinarily displays a scrolled window onto a work area whose size is fixed. If your work area is a Form, you may want to change the "Scrolling policy" resource to "Application defined". This removes the scroll bars and lets the Form resize with the window so that you can use the features of the Layout Editor to control resize behavior. Note: "Scrolling policy" does not take effect in the dynamic display but works correctly in the generated code.

If you do not add a work area to a MainWindow, the generated code produces warning messages when you run it.

Careful use of resources can make a Form emulate the behavior of a MainWindow. Experience has shown that it is often more convenient to use.

24.15 Menu



Display
Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings
Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled *
Radio always one
Radio behavior
Resize height
Resize width
Tear-off modal

Keyboard
Accelerator *
Menu post *
Mnemonic
Mnemonic charset

Callbacks
Map
Unmap
Entry

* Only if used as a popup menu. It is insensitive in all other cases.

The Menu widget provides pulldown, pullright and popup menus and is a specially configured RowColumn widget and shares its resource panel.

The active items in a menu can be PushButtons, ToggleButtons, or CascadeButtons. Menus can also contain Separators and Labels for display purposes.

To create a pulldown menu, add a Menu as a child of a CascadeButton that is a child of a MenuBar or OptionMenu. When a user clicks on the CascadeButton, the menu appears.

To create a pullright menu, add a Menu as a child of a CascadeButton that is a child of a Menu. When a user clicks on the CascadeButton, the menu appears. Pullright menus are only permitted in menus that are pulled down from a MenuBar, not in OptionMenus.

To create a popup menu, add a Menu as a child of a DrawingArea. When a user clicks on the DrawingArea with the right mouse button, the menu appears. A DrawingArea can have more than one popup menu as a child. In this case, the menu that pops up in the dynamic display depends on which menu is selected in the design hierarchy.

To create a Tear-off menu, set the Tear-off modal resource to enabled. Note that some Motif versions have a bug where, if this resource is not hard-coded and is part of the applications resource file, a call to XmRepTypeInstallTearOffModalConverter() must be made from either the main program or the Menu's pre-create prelude for the resource to take effect.

Figure 24-5 shows design hierarchies for the three types of menus.

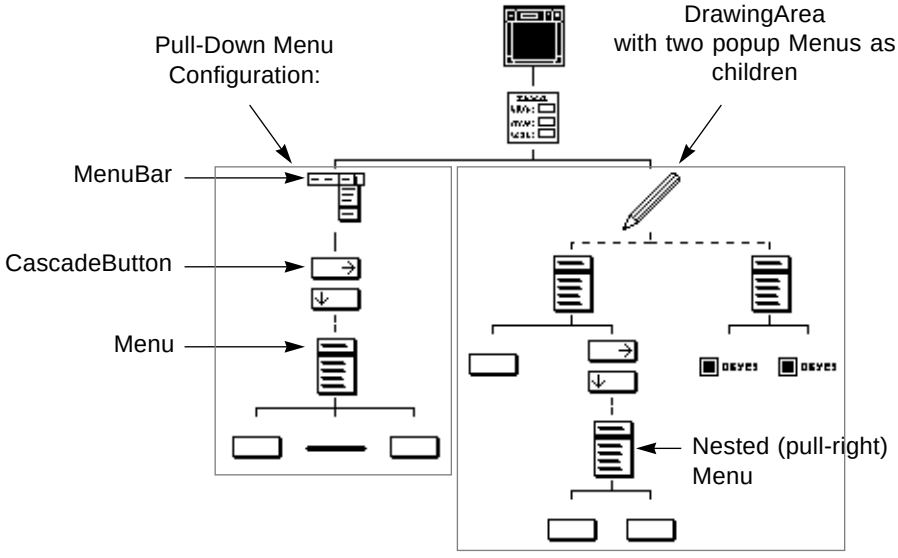


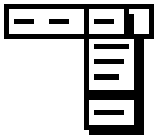
Figure 24-5 Sample Hierarchy Using Menus

Unlike pulldown and pullright menus, popup menus must be explicitly managed by the generated code. SPARCworks/Visual does not do this automatically because popup menus are context-sensitive in most applications. You can do this by using the input callback of the DrawingArea to position and manage the menu, or with an action routine using the translations mechanism. Normally, the menu is positioned using XmMenuPosition() and managed using XtManageChild().

To create a Menu with mutually exclusive toggle buttons, set the “Radio behavior” resource to “True”.

In Motif, Menus are technically siblings, not children, of the DrawingAreas or CascadeButtons from which they appear. However, SPARCworks/Visual displays its hierarchy as if the Menus were children of these widgets because the DrawingArea or CascadeButton affects the Menu’s behavior as a parent widget does. SPARCworks/Visual uses a dotted rather than a solid line to connect the Menu to its CascadeButton or DrawingArea. The dotted line indicates that the connection is not a true Motif parent-child relationship.

24.16 MenuBar



Display
Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings
Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard
Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks
Map
Unmap
Entry

The MenuBar widget displays a set of CascadeButtons from which you can pull down menus. The MenuBar is a specially configured RowColumn and shares its resource panel.

MainWindow, DialogTemplate and SelectionBox provide standard layouts that can include a MenuBar. If you do not use one of these to contain the MenuBar, you must use a Form and attach the MenuBar to its top, left and right sides. For further information about the differences, see the descriptions of the MainWindow, DialogTemplate, SelectionBox and Form. For a design hierarchy that includes a MenuBar with a typical configuration of children, see Figure 24-5 on page 24-511.

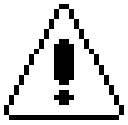
The default resource settings provide a standard menu bar as defined in the *Motif Style Guide*. You can change the “Packing” resource setting from “Tight” to “Column”.

- “Tight” makes all buttons the minimum size to accommodate their text
- “Column” makes all buttons the same size

If you use “Column” packing, the “Alignment” resource can be set to center the labels on the buttons. Changing other resources is not recommended.

A MenuBar positions all its CascadeButtons close together starting at the left. If your menu bar has a “Help” button, the *Motif Style Guide* recommends placing it at the right end of the menu bar. To designate a CascadeButton as the “Help” button, enter its variable name as the “Help widget” resource of the MenuBar.

24.17 MessageBox



Display	**Settings**	**Callbacks**
Message text	Default button	Cancel
Ok label	Dialog type	**Ok**
Cancel label	Alignment	
Help label	Minimize buttons	
Symbol pixmap		

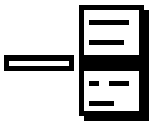
The MessageBox widget displays a message to the user. SPARCworks/Visual’s error messages are examples of MessageBoxes. The MessageBox is a composite widget that consists of three PushButton gadgets, two Labels and a Separator. These components are contained in a BulletinBoard that is not visible in the design hierarchy. To view the inherited BulletinBoard resources, click on “Bulletin Board Resources” in the resource panel.

Although a MessageBox can be used anywhere that a BulletinBoard can be used, it is usually placed in a Dialog Shell that is popped up to alert the user.

To display a message or pixmap in the message area, or to change the labels of buttons, change the resources in the resource panel of the MessageBox, not in the resource panels of the component widgets.

You can add a MenuBar and any number of button widgets as children of a MessageBox, as well as a single widget of another type, which becomes the work area. The work area can be a container widget, such as a Form, with children.

24.18 OptionMenu



Display
Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings
Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard
Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks
Map
Unmap
Entry

The OptionMenu widget is used to display a one-of-many choice without using the screen space required by a set of radio buttons. The page selectors in SPARCworks/Visual's resource panels are examples of OptionMenus. The OptionMenu is a specially configured RowColumn and shares its resource panel.

An OptionMenu is a composite widget that includes a Label and a CascadeButton. You should add a Menu child to the CascadeButton, with a PushButton for each choice. You can use Separators to divide groups of options. Figure 24-6 shows a sample hierarchy. Note that you cannot have a cascading option menu.

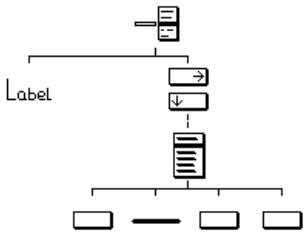
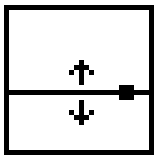


Figure 24-6 Sample Hierarchy for the OptionMenu

Set the label identifying the OptionMenu by changing the “Label” resource in the resource panel of the Label. Do not change the label of the CascadeButton as this displays the current setting of the OptionMenu.

24.19 PanedWindow



Margin width	Margin height
Sash width	Sash height
Sash indent	Sash shadow
Spacing	
Refigure	Separator

The PanedWindow widget is used to lay out a set of widgets in a vertical column of uniform width. Each child widget is laid out in a vertical partition that is separated from adjacent children by a movable separator like a window sash. The user can move the sash to determine how much vertical space is allotted to each child. Since the height of a PanedWindow is less than the aggregate height of its children, a PanedWindow saves vertical space without sacrificing functionality. The children of a PanedWindow can be container widgets that control the layout of other widgets.

The PanedWindow is a constraint widget. The Constraints panel applies to any child of the PanedWindow, not to the PanedWindow itself. You can display the Constraints panel by selecting “Constraints” from the Widget menu when one of the PanedWindow’s children is selected. The *Resource Panels* chapter discusses how to use this panel.

The Constraints panel lets you set the “Minimum” and “Maximum” height resources for the child. These provide limits on the height of the widget’s partition and positioning of the sashes.

The children in a PanedWindow are constrained to be the same width as the widest child.

You may need to reset a PanedWindow whenever you rearrange or resize its children.

24.20 PushButton

Display
Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select color
Select pixmap
Select insensitive pixmap

Margins
Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Settings
Alignment
Type
Resize
Push button
Shadow
Fill on Arm
Fill on select
Indicator as
Indicator type
Multi click
Set
Visible when off

Callbacks
Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Toggles
Widget
Gadget

Keyboard
*Accelerator **
*Accelerator text **
*Mnemonic **
*Mnemonic charset **
Mapping delay

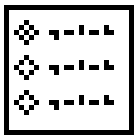
The PushButton widget displays a button that can be “pressed” by clicking a mouse button over it. It is derived from the Label and shares its resource panel. Like the Label, it can display either text or a pixmap.

There are different kinds of buttons for different needs, such as the ArrowButton and DrawnButton. For a button that pops up a menu, use a CascadeButton.

Setting the “Show as default” resource is not recommended since a BulletinBoard parent often changes this setting. The BulletinBoard decides which button to make the default.

* Sensitive when PushButton is child of Menu.

24.21 RadioBox



Display
Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings
Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard
Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks
Map
Unmap
Entry

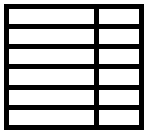
A RadioBox widget is used to contain a group of ToggleButtons that act as *radio buttons,* meaning that they are mutually exclusive. Selecting one toggle in the group deselects the previously selected one. A RadioBox is a specially configured RowColumn and shares its resource panel. You can set the “Packing”, “Columns”, and “Orientation” resources to create multiple columns as for RowColumn.

The ToggleButtons in the RadioBox are gadgets.

You can make a RowColumn act like a RadioBox by setting its “Radio behavior” resource to “Yes”. This configuration of the RowColumn provides more flexibility than the RadioBox does: for example, to have Labels, Separators, or other widgets inside the box with the ToggleButtons, or to use the widget version of the ToggleButton instead of the gadget.

The Menu also has a “Radio behavior” resource that can be set to make its ToggleButton children act as radio buttons.

24.22 RowColumn



Display
Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings
Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard
Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks
Map
Unmap
Entry

The RowColumn widget is used to arrange child widgets in a grid. It is often used for arranging items such as groups of buttons or toggles. For example, a Menu is a specially configured RowColumn widget. Other widgets that are based on RowColumn are OptionMenu, MenuBar, Menu and RadioBox.

A RowColumn can have any number of children. The default arrangement of RowColumn items is one vertical column. To create multiple columns, set the “Packing” resource to “Column”, then set the “Columns” resource.

Items are read in order starting down the first column when the “Orientation” resource setting is “Vertical” and across the first row when the “Orientation” resource setting is “Horizontal”. When the “Orientation” resource setting is “Horizontal”, the “Columns” setting refers to the number of horizontal rows.

Because a RowColumn widget is not designed to have its layout changed dynamically, it may not display the changes you expect. If its children seem to be the wrong size on the dynamic display, try resetting the RowColumn.

Note – When you use multiple columns, a RowColumn forces all items to be the same width. Sometimes this results in wasted space, as in the “Before” view of Figure 24-7, where the left column has short Labels and the right column has long TextFields. You can resize the TextFields to match the width of the Labels. Note that although the new value is accepted in the resource panel, the difference in width is not apparent in the dynamic display until you have changed the value for all of the TextFields, as shown in the “After” view of:

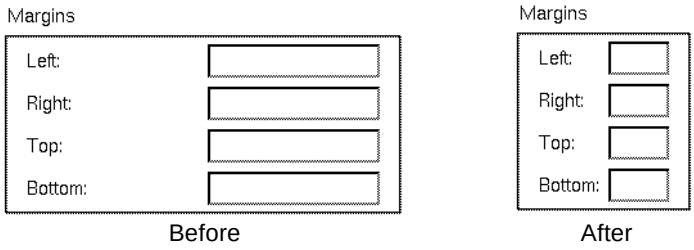


Figure 24-7 Resizing Widgets in a RowColumn

To create columns of unequal width, use a Form instead of a RowColumn. You can also nest RowColumns to create layouts that are more complex than rows and columns.

To create a group of radio buttons inside a RowColumn, use ToggleButtons and set the “Radio behavior” resource of the RowColumn to “Yes”.

24.23 Scale

Display	Settings	Callbacks
Decimal points	**Orientation**	Drag
Minimum	**Direction**	**Value changed**
Maximum	**Show value**	
Value		
Title		
Scale width		
Scale height		
Scale multiple		
Font		

The Scale widget offers a range of values to choose from and displays a slider that can be moved to change the current value. You can drag the slider to move it continuously, click in the trough with the left mouse button to move the slider incrementally, or click in the trough with the middle mouse button to move the slider to the cursor location.

A Scale can have children of almost any type. These are usually Labels, which the Scale lays out evenly along its length.

Changing the orientation of a Scale can have strange effects. If problems occur, try resetting the Scale or its parent.

24.24 ScrollBar

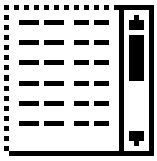
Display	Settings	Callbacks
Slider size	**Orientation**	Decrement
Minimum	Direction	Drag
Maximum	Show arrows	Increment
Value		Page decrement
Increment		Page increment
Page increment		To bottom
Initial delay		To top
Repeat delay		**Value changed**
Trough color		

The ScrollBar widget lets users view data that requires more space than the display area provides. ScrollBars are rarely used alone. It is easiest to use them as part of a composite widget such as a ScrolledWindow, ScrolledList, or ScrolledText.

Each ScrollBar is represented as a rectangle with an arrow pointing outward at each end and a slider inside it. The display area is scrolled either by moving the slider or by clicking on an arrow. You can drag the slider to move it continuously, click in the trough or on the arrows with the left mouse button to move the slider incrementally, or click in the trough with the middle mouse button to move the slider to the cursor location. You can edit the resources to control the amount by which the display area scrolls on each scrolling action.

A ScrollBar cannot have children.

24.25 ScrolledList



Scrolled window margins
Width
Height
Spacing

Main window margins
Width
Height

Callbacks
Traverse obscured

Scroll bar display
Scroll bar placement
Scrolling policy
Visual policy
Show separators
Command location
Message window

The ScrolledList widget is a composite widget that displays a scrollable list of items. A ScrolledList is a specially configured ScrolledWindow that contains a List widget. It shares the resource panel of the ScrolledWindow. The resources of a List widget child can be set in the normal way.

A ScrolledList resizes itself whenever you add or delete items from the List so that its width always matches that of the widest item in the list. In some versions of the Motif toolkit, the ScrolledList may become confused about its correct width.

To prevent unwanted resizing, you must constrain a ScrolledList in some way. You can constrain it in a Form by using attachments and positions. However, if the Form also contains other widgets, this can produce strange results. To avoid this, use a ScrolledList in a Form containing nothing except a ScrolledList, as shown in Figure 24-8:

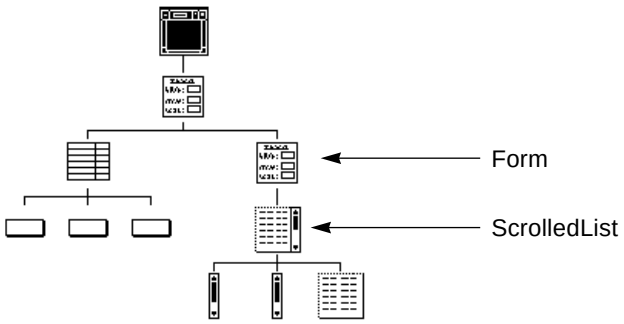
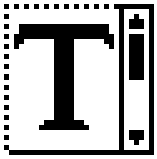


Figure 24-8 Effective ScrolledList Placement

You can then place this Form in another Form with other widgets. Attach the ScrolledList to its parent Form on all four sides and set the “Resize policy” of the Form to either “Grow” or “None”. You can set the width and height of the Form to define a reasonable size for the ScrolledList, or fix the initial size of the Form, and therefore the ScrolledList it contains, by using attachments.

Constraints set in the Form supersede the ScrolledList’s “Visible items” resource setting and the width of individual items in the list.

24.26 ScrolledText



Scrolled window margins
Width
Height
Spacing

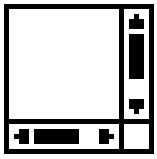
Main window margins
Width
Height

Callbacks
Traverse obscured

Scroll bar display
Scroll bar placement
Scrolling policy
Visual policy
Show separators
Command location
Message window

The ScrolledText widget is a composite widget that provides a scrollable text area. A ScrolledText is a specially configured ScrolledWindow that contains a Text widget. It shares the resource panel of the ScrolledWindow. The resources of the Text widget child can be set in the usual way.

24.27 ScrolledWindow



Scrolled window margins
Width
Height
Spacing

Main window margins
Width
Height

Callbacks
Traverse obscured

Scroll bar display
Scroll bar placement
Scrolling policy
Visual policy
Show separators
Command location
Message window

The ScrolledWindow widget is used to display data that requires more space than is available. It is a composite widget consisting of two scroll bars and a viewing area onto a visible object that can be larger than the ScrolledWindow. A ScrolledWindow can have one child of almost any type.

Although the visible object can be any kind of widget, it is commonly a DrawingArea or a composite widget containing other widgets. For example, a ScrolledWindow can be used to scroll through a form or table of widgets by placing a Form or RowColumn in it. For a scrollable list or text display, use the ScrolledList or ScrolledText widget.

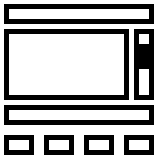
If you do not add a child to a ScrolledWindow, the generated code produces warning messages when you run it.

If the “Scrolling policy” resource is set to “Automatic”, the toolkit handles scrolling for you and the scroll bars are created automatically.

If the “Scrolling policy” resource is set to “Application defined”, you must respond to movements of the scroll bars by changing the information displayed in the ScrolledWindow’s child. In this case, SPARCworks/Visual generates code to create the scroll bars for you if any resource, callback, or name is set.

The effect of the resources that control scroll bar behavior - “Scrolling policy” and “Scroll bar display” - is not reflected in the dynamic display but they work correctly in the generated code.

24.28 SelectionBox



Display
No match string
Pattern
Max history items
Text String
Visible item count
Text columns
Directory mask
Directory

Settings
Dialog type
Minimize buttons
Must match
File type
Work area placement

Labels
Apply label
Ok label
Cancel label
Help label
List label
Selection label
Filter label
Directory label

Callbacks
Apply
Cancel
Ok
No match
Command changed
Command entered

The SelectionBox widget is a composite widget used to select one or more items from a scrollable list. The SelectionBox combination includes a ScrolledList for the item list, two Labels, a Separator and four PushButtons, which are gadgets. These components are contained in a BulletinBoard widget that is not visible in the design hierarchy. To view resources inherited from the BulletinBoard, click on “Bulletin Board Resources” in the resource panel.

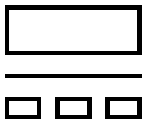
While a SelectionBox can be used anywhere that a BulletinBoard can be used, it is usually placed in a Dialog Shell that is popped up to get a selection from the user.

To change the labels of button or label widgets, change the resources in the resource panel of the SelectionBox, not in the resource panels of the individual widgets.

You can add multiple children to a SelectionBox. The first child becomes the work area. This can be a container widget containing additional widgets. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the SelectionBox widget’s hierarchy. The additional children can include a MenuBar and any number of PushButton widgets.

The four PushButtons provided are labeled “OK”, “Apply”, “Cancel”, and “Help”. The “Apply” PushButton can be displayed by setting the “Managed” toggle in the PushButton’s Core resource panel.

24.29 SelectionPrompt



Display
No match string
Pattern
Max history items
Text String
*Visible item coun*t
Text columns
Directory mask
Directory

Settings
Dialog type
Minimize buttons
Must match
File type
Work area placement

Labels
Apply label
OK label
Cancel label
Help label
List label
Selection label
Filter label
Directory label

Callbacks
Apply
Cancel
OK
No match
Command changed
Command entered

The SelectionPrompt widget is used to prompt the user for text input. It is a composite widget consisting of a Label used for a question or prompt, a Text box into which the answer is typed and three PushButtons (“OK”, “Cancel”, and “Help”). An “Apply” PushButton is also provided. It is displayed by setting the “Managed” toggle in that PushButton’s Core resource panel. These components are contained in a BulletinBoard that is not visible in the design hierarchy. To view the resources inherited from the BulletinBoard, click on “Bulletin Board Resources” in the resource panel.

A SelectionPrompt is a specially configured SelectionBox and shares its resource panel. Most of the information about the SelectionBox applies to the SelectionPrompt, except that the SelectionPrompt does not include a List. While a SelectionPrompt can be used anywhere that a BulletinBoard can be used, it is usually placed in a Dialog Shell that is popped up to query the user for input. To change the prompt or the labels of the buttons, change the resources in resource panel of the SelectionPrompt, not in the resource panels of the individual widgets. The SelectionPrompt can have multiple lines.

The PushButtons in a SelectionPrompt are gadgets. You can add multiple children to a Prompt. The first child becomes the work area. This can be a container widget. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the Prompt widget’s hierarchy. The additional children can include a MenuBar and any number of PushButtons.

24.30 Separator

Margins	**Toggles**
Type	Widget
Orientation	Gadget

The Separator widget is a line used to separate objects visually. A Separator cannot have children. Set the “Orientation” resource to specify a vertical or horizontal line. Set the “Type” resource to specify a different line type such as a double line or a dashed line.

Separators can be used to separate items in a Menu or RowColumn or to separate widgets in a dialog box. To separate widgets in a Form, make a Separator a child of the Form along with the other widgets. The Separator is very small until it is constrained in some way. To stretch it the length or width of the Form, attach it to both sides of the Form, or to other widgets on each side. Setting the size of a Separator explicitly is not recommended. A Separator with a “Type” of “No line” can be used as an invisible widget.

Separators are often used inside Menus to divide items into groups. The Separator appears between its adjacent siblings, as shown in Figure 24-9.

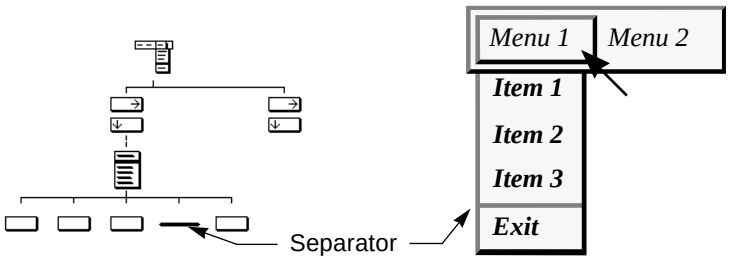


Figure 24-9 Use of Separator Inside a Menu

You can use Separators inside a RowColumn. Figure 24-10 shows a sample hierarchy and the resulting dynamic display. When you use Separators in a RowColumn, set the orientation of the Separators explicitly to “Vertical” or “Horizontal”. Separators in a RowColumn span a cell the size of every other element in the array. This can produce more white space around the Separator than is pleasing. If you want different proportions, use a Form for your column layout.

Set the RowColumn’s “Spacing” resource to 0 to eliminate a gap between adjacent separators.

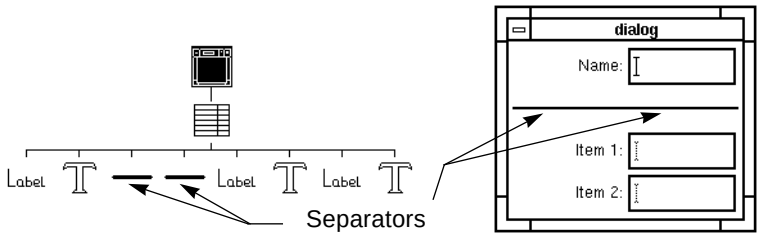
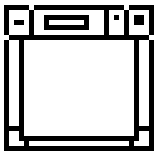


Figure 24-10 Use of Separators in a RowColumn (Horizontal Orientation, 4 Rows)

24.31 Shell



Display	Settings	Dimensions
Title	Delete response	Base width
Mwm menu	Keyboard focus	Base height
Icon mask	Input	Width inc
Icon pixmap	**Transient**	Height inc
Icon name *	Allow resize	Min width
Label font	Override redirect: No	Min height
Button font	Iconic *	Max width
Text font	Unit type	Max height
Input method	Window gravity	Min aspect X
Pre-edit type	Initial state	Min aspect Y
	Save under	Max aspect X
	Audible warning	Max aspect Y
		Timeout
		Min aspect Y
		Max aspect X
		Max aspect Y

Callbacks	Toggles
Pop down	**Application shell**
Pop up	**Top level shell**
	Dialog shell

* Insensitive if Shell is set to Dialog Shell.

The Shell widget forms the interface between your design and the Motif window manager. Every SPARCworks/Visual design hierarchy must have a Shell as its root widget.

Motif has various kinds of Shells but SPARCworks/Visual groups them into a single widget. The SPARCworks/Visual Shell can be configured as an Application, Top level, or Dialog Shell using its resource panel.

The Application Shell is used as the main application window. Your application must have at least one (and usually only one) Application Shell. Top level Shells look and act like Application Shells. Typically, they are used for all primary windows in the application except the first. Dialog Shells are used for secondary windows such as pop-up dialogs. If an Application or Top level Shell is closed or iconified, all associated Dialog windows also disappear.

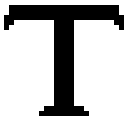
An Application or Top level Shell appears as a Dialog Shell in the dynamic display but the generated code produces the correct type of Shell. To check the icon pixmap, set the “Transient” resource to “No”, then reset the Shell. This produces the full set of decorations, allowing you to iconify the dynamic display window.

A Shell can only have one child, which can be of any type. However, much of the Shell’s behavior is based on the assumption that its child is a BulletinBoard, Form, or similar container widget, since the Shell exercises no geometry management over its descendants. A Shell is not visible until it has a child.

Setting a Shell’s width and height on its Core resource panel does not control the window size. To control initial window size, set the *minimum* width and height resources of the Shell, or set the width and height of the Shell’s child.

To control the initial position of a window, set the “Default position” resource of the Shell’s child to “No”, and set the *x* and *y* resources of the child, not the Shell.

24.32 Text



Display
Value
Cursor position
Margin width
Margin height
Maximum length
Top position
Selection threshold
Blink rate
Columns
Rows
Font

Settings
Edit mode
Auto show cursor
Editable
Pending delete
Cursor visible
Resize height
Resize width
Word wrap
Verify bell
Scroll horizontal
Scroll vertical
Scroll left side
Scroll top side

Callbacks
Activate
Focus
Losing focus
Gain primary
Lose primary
Modify verify
Motion verify
Value changed

Toggles
Text
Text Field

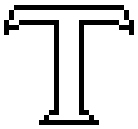
The Text widget provides an area for entering multi-line text. A wide range of callbacks is provided to deal with input verification and validation.

To use multi-line text, you must set the “Edit mode” resource to “Multi line”. To change the height of the Text widget to display multiple lines of text, you can change the “Rows” resource setting to a number greater than 1. Changing the number of Rows may or may not be effective, depending on the type of widget used as the Text widget’s parent.

To create a scrollable text editing area, use a ScrolledText, a composite widget that includes a Text widget. Although the Text widget can be the child of a ScrolledWindow, this configuration does not work well. If you use this configuration, change the “Edit Mode” resource to “Multi line” and increase the number of Rows and Columns to exceed the size of the ScrolledWindow viewing area.

The Motif toolkit provides functions for accessing and modifying the text in the widget. For details, see the Motif documentation.

24.33 TextField



Display
Value
Cursor position
Margin width
Margin height
Maximum length
Top position
Selection threshold
Blink rate
Columns
Rows
Font

Settings
Edit mode
Auto show cursor
Editable
Pending delete
Cursor visible
Resize height
Resize width
Word wrap
Verify bell
Scroll horizontal
Scroll vertical
Scroll left side
Scroll top side

Callbacks
Activate
Focus
Losing focus
Gain primary
Lose primary
Modify verify
Motion verify
Value changed

Toggles
Text
Text Field

The TextField widget is a variant of the Text widget that provides an area for entering only a single line of text. It shares the Text's resource panel and has all the Text's editing features except multi-line capability.

You can change from TextField to Text by using the toggle. However, to get multi-line capability, you must also set the "Edit mode" resource to "Multi line".

The Motif toolkit provides functions for accessing and modifying the text in the widget. For details, see the Motif documentation for details.

24.34 ToggleButton

Display	Settings	Callbacks
Label	Alignment	*Activate*
Font	Type	*Cascading*
Pixmap	*Resize*	Arm
Insensitive pixmap	*Push button*	Disarm
Cascade pixmap	*Shadow*	*Expose*
Arm color	*Fill on arm*	*Resize*
Arm pixmap	Fill on select	**Value changed**
Select color	Indicator on	
Select pixmap	Indicator type	
Select insensitive pixmap	*Multi click*	
	Set	
	Visible when off	

Margins	Keyboard	Toggles
Top	*Accelerator* *	Widget
Bottom	*Accelerator text* *	Gadget
Left	*Mnemonic* *	
Right	*Mnemonic charset* *	
Width	*Mapping delay*	
Height		
Spacing		
Default shadow		
Indicator size		

* Sensitive when ToggleButton is child of Menu.

The ToggleButton widget provides a simple on/off toggle for indicating "yes/no" choices. It is derived from the Label and shares its resource panel.

ToggleButtons can be made into mutually exclusive radio buttons by placing them inside a RadioBox, or inside a Menu or RowColumn whose "Radio behavior" resource is set to "Yes". Radio buttons have a different shape from normal toggles, as shown in Figure 24-11.

Figure 24-11 Radio Buttons and Normal Toggle Buttons

You can configure the ToggleButton to resemble a PushButton that appears to push in and out to represent on and off settings. To do this:

1. **Set the “Shadow Thickness” Core resource to 2. This draws a border around the button.**
2. **Set the “Indicator on” resource to “No”. This suppresses the small square indicator.**
3. **Set the left margin to 0. This removes the space which was occupied by the indicator.**

24.35 Mapping Motif Widgets to Windows

Following is a list of the Motif widgets which can be selected from within SPARCworks/Visual in Windows mode along with the way in which they are mapped to a Windows class.

ApplicationShell

Maps to CFrameWnd.

TopLevelShell

Maps to CDialog.

DialogShell

Maps to CDialog.

MainWindow and ScrolledWindow

Map to CWnd unless they are the child of a Shell, in which case they are ignored for Windows.

Frame, RadioBox and ToggleButton

Map to CButton.

BulletinBoard, Form, RowColumn and DialogTemplate

Map to CWnd if they are structured as a C++ class.

DrawingArea

Maps to CWnd unless its parent is a ScrolledWindow, MainWindow or Shell in which case it is ignored for Windows. Otherwise it is forced to be structured as a C++ class.

MenuBar, PopupMenu and CascadeButton

Map to a CMenu and cannot be structured as a C++ class.

OptionMenu

Maps to a CComboBox and cannot be structured as a C++ class.

FileSelectionBox

Maps to a CFileDialog class.

Paned Window

Maps to CSplitterWnd.

Label

Maps to a CStatic.

PushButton

Maps to a CButton if XmNlabelType is XmLABEL or to a CBitmapButton if XmNlabelType is XmPIXMAP.

Separator

This is not mapped to an object on Windows - instead it is added as a Menu attribute, if part of a menu. If not in a menu, it is ignored.

Scale and Scrollbar

Map to CScrollbar unless they are part of a ScrolledWindow in which case the appropriate style is add to the enclosing class and they are ignored as widgets.

TextField and Text

Maps to CEdit.

List

Maps to CListBox.

ScrolledText

This maps to CEdit with appropriate scrolling styles and the ScrolledWindow part is ignored.

ScrolledList

This maps to CListBox with appropriate scrolling styles and the ScrolledWindow part is ignored.

24.36 Mapping Motif Resources to Windows

Although Windows uses resources, the way in which they are used is different from X/Motif. Resources used by Windows are compiled into the application. There is also a far more restricted set than on Motif.

SPARCworks/Visual only generates bitmaps, icons and accelerators as Windows resources. Other Motif resources are mapped to visual window attributes or written into the source code.

24.37 Window Styles

When a Windows object is created, *window styles* can be specified. These are bit flags which are or'd together. The following example shows how a toggle button would be created:

```
Create ( "Classical", WS_CHILD | WS_VISIBLE | WS_TABSTOP |
       BS_AUTORADIOBUTTON, rect, this, IDC_shell_classical);
```

The second parameter to this method, which is a method inherited from a basic MFC class, is the window style. When you set resources in SPARCworks/Visual, suitable window styles are chosen. Below is a list of the window styles available for each widget which can be mapped to a Windows object. The list also shows when they are used and the corresponding Motif resource.

24.37.1 Shells

All Shells have:

- WS_POPUP
- WS_CAPTION
- WS_SYSMENU
- WS_MINIMIZE - if *XmNinitialState* is set to Iconic
- WS_VSCROLL and WS_HSCROLL - if child is MainWindow or ScrolledWindow and the appropriate scrollbar is named, has a resource set or has a callback or method set

24.37.2 ApplicationShell

In addition to Shell styles, has:

- WS_THICKFRAME
- WS_MINIMIZEBOX
- WS_MAXIMIZEBOX

24.37.3 TopLevelShell

Exactly the same styles as ApplicationShell.

Note – This does not mean that ApplicationShell and TopLevelShell are exactly the same on Windows - they are different classes.

24.37.4 DialogShell

In addition to Shell styles, has:

- WS_THICKFRAME - unless *XmNoResize* is set to True on the BulletinBoard derived child

24.37.5 MainWindow and ScrolledWindow

- Only supported if *XmNscrollingPolicy* is set to XmAPPLICATION-DEFINED
- WS_CHILD

- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_VSCROLL and WS_HSCROLL - if the appropriate scrollbar is named, has a resource set or has a callback or method set

24.37.6 Frame

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_GROUP
- BS_GROUPBOX

24.37.7 BulletinBoard, Form, RowColumn, DrawingArea and DialogTemplate

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if is *XmNtraversalOn* is True

24.37.8 RadioBox

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_GROUP
- BS_GROUPBOX - if parent is not a Frame

24.37.9 MenuBar, PopupMenu and CascadeButton

These widgets are not windows on Windows, as all other objects are - they are CMenu objects. CMenu is not derived from CWnd. This means that they have no window styles associated with them. Their children (or the children of the PulldownMenu in the case of the CascadeButton) are generated by a call to AppendMenu for each child. The following flags are passed to AppendMenu depending on the type of child:

- MF_POPUP - for a CascadeButton which has a PulldownMenu
- MF_STRING - for CascadeButtons without a PulldownMenu, PushButtons, Labels and ToggleButtons which do not have a valid pixmap object for *XmNlabelPixmap*
- MF_GRAYED - if *XmNsensitive* is False (for the CascadeButton in the case of a MenuBar)
- MF_MENUBREAK - if the item (or the CascadeButton in the case of a MenuBar) starts a new column

The following apply to calls to AppendMenu from a PopupMenu or CascadeButton only:

- MF_DISABLED - for Labels if *XmNsensitive* is True
- MF_SEPARATOR - for separators
- MF_CHECKED - for ToggleButtons which have *XmNset* True

24.37.10 OptionMenu

- WS_CHILD
- CBS_DROPDOWNLIST
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_GROUP - if *XmNnavigationType* is not *XmNONE*
- The SetFont method is called if the widget has a font object resource set for the *XmNbuttonfontList* resource or if it inherits a font object set for an enclosing BulletinBoard or Shell

24.37.11 FileSelectionBox

This maps to a CFileDialog class. Since the Create method is not called explicitly for a CFileDialog (instead InitDialog and DoModal are called) there are no styles. Instead, resources are mapped to parameters passed to the New method:

- OpenFileDialog - always TRUE
- lpszDefExt - always NULL
- lpszFileName - set to the value of *XmNdirSpec* if specified, otherwise NULL
- dwFlags - always OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT
- lpszFilter - if *XmNpattern* is specified this value is set as follows:

```
"<XmNfilterLabelString>(<XmNpattern>)|XmNpattern|All
files(*.*)|*.*||"
```

If *XmNpattern* is not set this parameter is NULL

- pParentWnd - the main window

24.37.12 PanedWindow

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- The CreateView method is called to create each of the child panes. This requires that the child pane classes can support dynamic creation (i.e. have the IMPLEMENT_DYNCREATE macro). SPARCworks/Visual will generate the appropriate macro invocations to support dynamic creation of child pane classes.

24.37.13 Label

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True

- An alignment (SS_LEFT, SS_CENTER, SS_RIGHT) depending on the alignment of the label (determined either from *XmNalignment* or from parent's *XmNentryAlignment* if parent is a RowColumn)
- SS_ICON - if *XmNlabelType* is set to *XmPIXMAP* and *XmNlabelPixmap* is set to a Pixmap object
- The caption parameter to the Create method is the value of *XmNlabelString* if set, otherwise the widget name
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource. If the widget is being created (i.e. is not a component) then SetFont will be called if an ancestor BulletinBoard or Shell has *XmNlabelFontList* set
- The SetIcon method is called if the widget has a valid Pixmap object set for *XmNlabelPixmap*

24.37.14 PushButton

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- BS_OWNERDRAW - if *XmNlabelType* is set to *XmPIXMAP*
- BS_DEFPUSHBUTTON - if the widget is set as the default button for an ancestor BulletinBoard which is itself a descendant of a DialogShell or a TopLevelShell and there are no CWnd objects intervening between the button and the CDialog
- The caption parameter to the Create method is the value of *XmNlabelString* if set, otherwise the widget name
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource. If the widget is being created (i.e. is not a component) then SetFont will be called if an ancestor BulletinBoard or Shell has *XmNlabelFontList* set.

24.37.15 ToggleButton

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False

- WS_TABSTOP - if *XmNtraversalOn* is True
- BS_AUTORADIOBUTTON - if *XmNindicatorType* is set to XmONE_OF_MANY, otherwise BS_AUTOCHECKBOX
- The SetCheck method is called if *XmNset* is set
- The caption parameter to the Create method is the value of *XmNlabelString* if set, otherwise the widget name
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource

24.37.16 Scale and Scrollbar

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- SBS_HORIZ or SBS_VERT - depending on the setting of *XmNorientation*
- SetScrollRange is called if the widget is a ScrolledWindow component or if either *XmNmaximum* or *XmNminimum* are set
- SetScrollPos is called if *XmNvalue* is set

24.37.17 TextField and Text

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_BORDER - if *XmNshadowThickness* is greater than zero
- WS_GROUP if *XmNnavigationType* is not *XmNONE*
- ES_MULTILINE and ES_WANTRETURN if *XmNeditMode* is set to *XmMULTI_LINE_EDIT*
- ES_READONLY if *XmNeditable* is set to false
- WS_VSCROLL - if the parent is a ScrolledText and *XmNscrollVertical* is the default or set to true
- WS_HSCROLL - if the parent is a ScrolledText and *XmNscrollHorizontal* is the default or set to true

- The SetWindowText method is called if the *XmNvalue* resource is set
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource or if it inherits a font object set for an enclosing BulletinBoard or Shell
- The LimitText method is called if the *XmNmaxLength* resource is set

24.37.18 List

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_GROUP if *XmNnavigationType* is not *XmNONE*
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_BORDER - if *XmNshadowThickness* is greater than zero
- WS_VSCROLL and WS_HSCROLL - if the parent is a ScrolledList
- LBS_EXTENDEDSEL - if *XmNselectionPolicy* is *XmEXTENDED_SELECT*
- LBS_MULTIPLESEL - if *XmNselectionPolicy* is *XmMULTIPLESELECT*
- LBS_DISABLENOSCROLL - if parent is ScrolledList and *XmNscrollbarDisplayPolicy* is not *XmAS_NEEDED*
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource or if it inherits a font object set for an enclosing BulletinBoard or Shell

24

Troubleshooting in SPARCworks/Visual 25

25.1 Introduction

This chapter is intended as a quick reference to some common questions and problems new SPARCworks/Visual users may have. It is organized loosely by functionality:

- SPARCworks/Visual Interface
- Resource Panels
- Layout Editor
- Links
- Code Generation

The subheadings in this chapter, unlike those elsewhere in this manual, do not describe features of SPARCworks/Visual but symptoms of problems. Scan the left-hand margin for a brief description of your problem.

25.2 SPARCworks/Visual Interface

This section discusses problems you may encounter if SPARCworks/Visual cannot find the correct resource file. SPARCworks/Visual must be installed so that X looks at the SPARCworks/Visual resource file. These problems refer to the SPARCworks/Visual interface, not to the dynamic display.

Labels Don't Display Correctly

Symptom: The labels on the SPARCworks/Visual buttons, prompts, and menu commands do not display correctly.

Cause and Solution: These labels are only available when the correct resource file is read. If the labels are not available, X substitutes variable names. Reconfigure your system so SPARCworks/Visual reads the SPARCworks/Visual resource file. There are several ways to do this in X. Consult your system administrator.

Only a Few Labels Are Wrong

Symptom: Most of the SPARCworks/Visual display is correct, but a few button labels or other resources are wrong.

Cause and Solution: Your configuration may be reading an obsolete version of the SPARCworks/Visual resource file. Check the software version and make sure SPARCworks/Visual is reading the resource file that came with that version.

Blank Help Screens

Symptom: Help screens come up blank.

Cause and Solution: Either your X environment is not finding the correct resource file, or the resource file is not accessing the correct search path. Make sure the "helpDir" resource contains the search path for your SPARCworks/Visual help database.

25.3 Resource Panels

This section discusses problems you may encounter when you use the resource panels. If you encounter problems with resource values at run time of the generated application, see the *Code Generation* section of this chapter.

For advice on setting resources for specific widgets and combinations of widgets, see the *Widget Reference* chapter and your Motif documentation. Note that many apparent problems with resource settings can be solved by resetting the widget involved or the Shell.

When you set values in SPARCworks/Visual's resource panels, SPARCworks/Visual actually resets resources of widget instances in the dynamic display. Sometimes the results are not what you expect:

Resource Settings Are Rejected

Symptoms:

- The value you typed into a field of the resource panel is not accepted
- The value in the resource panel reverts to the former value
- The value in the resource panel changes to some other value
- The dynamic display reflects the value on the resource panel, but the value is not what you typed

Cause: Motif cannot set the new value you specified. The most common reason is that the selected widget is being constrained by another widget, usually its parent. This is particularly common with size and position resources such as the width and height resources in the Core resource panel, which are frequently overridden.

For example, if a RadioBox contains a group of ToggleButtons, the width of the individual ToggleButtons is determined by their text and margin resources; the width of the RadioBox is calculated from the width of its widest child; and all the ToggleButtons are forced to be the same width as the widest one. A width setting on the Core resource panel is overridden by the calculated value. If the RadioBox is in turn the child of a Form, attachments set in the Layout Editor can override the RadioBox rules.

Solution: Check your design for constraints that may be overriding the setting. Use the constraints imposed by other widgets to achieve the desired effect.

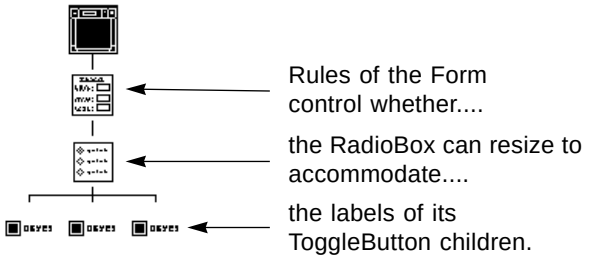


Figure 25-1 Resource Relationships

Resource Settings Don't Take Effect

Symptom: A new value is accepted on the resource panel, but the dynamic display does not immediately reflect the change as you expect.

Solutions: Reset the widget. If that doesn't work, reset the Shell. If resetting the Shell doesn't work, consult the *Widget Reference* chapter. You may need to set two or more resources to achieve a single effect. For example, to display a pixmap on a label or button, you must set the "Pixmap" resource to specify the pixmap you want, and set the "Type" resource to "Pixmap."

A few resources are never reflected in the dynamic display, as discussed below.

Geometry Resources Are Overridden

Symptom: The width and height resources on a widget's Core resource panel are overridden.

Cause and Solution: The width and height Core resources of any widget can be overridden, either by resources specific to that widget class or by geometry rules of the widget's parent. Consult the *Widget Reference* chapter or Motif documentation for information about resources of the widget and its parent that may be controlling its size.

Cause and Solution: The width and height Core resources of a Shell can be overridden by the corresponding resources of the Shell's child. One way to control the size of a dialog is to set the width and height resources of the Shell's child. Another way is to set the *minimum* width and height resources of the Shell, which are not overridden by the child.

Cause and Solution: The *x,y* position resources of a Shell can be overridden by the position resources of the Shell's child if the Shell's child is a BulletinBoard, or a derivative of the BulletinBoard such as a Form, DialogTemplate, or MessageBox. To use the Shell's ***x,y*** position resources to control the dialog's position, set the BulletinBoard's "Default position" resource to "No."

Resources Are Not Reflected After Resetting

Symptom: The resource panel accepts your resource settings, but they are not reflected in the dynamic display even when you reset the widget.

Cause: Certain resources can only be set when a widget is first created. These resources cannot be changed once the widget is added to the dynamic display. They include various resources related to scrollbars such as "Scrolling policy." SPARCworks/Visual still lets you change the value on the resource panel, and even though the value is not reflected in the dynamic display, the new setting takes effect at run time of the generated application. For details about individual resources, consult the *Motif Programmer's Manual*.

Cause: The "Dialog Style" resource of BulletinBoards and widget classes that derive from the BulletinBoard can be set to "Modeless," "System Modal," or "Application Modal." Modal dialogs disable all other dialogs until they receive an answer from the user, so if a setting of "System Modal" or "Application Modal" were effective in the dynamic display, you would not be able to do anything else until the dynamic display was closed. You can still select one of these settings; SPARCworks/Visual disables the setting while you are building the hierarchy and generates it correctly in the code.

Solution: If you have resources of this kind in your design, you can only see the final results after the design is finished and you generate the code. You can see the results at intermediate stages by generating code and running a prototype using the following steps:

1. **Generate a primary code module. Include all types of resources and a** *main()* **program.**
2. **Generate a stubs file (only necessary if your design has callbacks). Leave the function braces empty.**
3. **Compile and link the generated files.**
4. **Run the resulting program.**

Expected Fonts Do Not Display In SPARCworks/Visual

Symptom: A widget with a default font setting does not display the font you expect in the dynamic display.

Cause: When a widget does not have an explicit font setting, Motif searches back through the design hierarchy to find the widget's nearest BulletinBoard, BulletinBoard derivative, or Shell ancestor, and uses the font setting of that ancestor for the current widget. Therefore, if several buttons all have default font settings but are children of different BulletinBoards, some may show different fonts from others.

Solution: Setting a font on the BulletinBoard or Shell instead of on individual widgets is a convenient way of setting all the fonts at once. To get uniformity you must use the same font on all parent widgets with explicit font settings. A font object is a convenient way to do this.

Font Change on Parent Doesn't Affect Children

Symptom: Changing the font on a BulletinBoard, BulletinBoard derivative, or Shell has no effect on its children.

Cause and Solution: The new font setting on a parent widget does not affect the children until you reset the parent widget. Reset the parent widget.

Cause and Solution: Font settings on parent widgets do not affect their children if the children have explicit font settings of their own. Make sure the children have default font settings.

CascadeButtons in DialogTemplate Don't Display

Symptom: The DialogTemplate does not resize to accommodate CascadeButtons in the MenuBar.

Cause: This is a bug in some versions of Motif. The DialogTemplate resizes properly to accommodate the button box and work area, but if the MenuBar exceeds the width of the button box and work area, it is cut off.

Solution: Add the button box and work area children to the DialogTemplate before you add the CascadeButtons to the MenuBar. In many cases the width of your work area or button box will create enough width to accommodate the MenuBar.

Solution: If your MenuBar is still too wide to display, use a Form as the work area, or put the work area inside a Form, then use the Layout Editor to add extra space around the work area.

Solution: You can force the DialogTemplate to be wider by setting its “Width” resource on the Core resource panel.

25.4 Layout Editor

This section discusses problems you may encounter when you use the Layout Editor. Note that many apparent problems in the Layout Editor can be solved by resetting the Form.

Widget Becomes Very Small or Very Large

Symptom: The widget becomes very small or very large.

Cause and Solution: In some versions of Motif there is a bug in the Form that appears when the Form is a child of a DialogTemplate. When you reset the Form, any container widgets inside the Form become very small. If you have this problem, resetting the DialogTemplate instead of the Form corrects it. To do this, you must go from the Layout Editor screen to the main SPARCworks/Visual screen, select the DialogTemplate in the construction area, and give the “Reset” command. To resume working in the Layout Editor, first select the Form again in the construction area.

Cause and Solution: Attachments can change the size of a widget. For example, attaching the edges of a widget to the corresponding edges of the Form forces the widget to span the full width or height of the Form. Reset the Form. If resetting the Form doesn’t work, remove some of the attachments from the widget and reset again.

Methods of breaking attachments are listed below. For a more complete discussion, see the *Layout Editor* chapter.

- Use “Undo” to remove the most recent attachment
- Move the widget
- In the “Attach” mode, click just inside the edge of the widget or drag from just inside the edge of the widget toward the widget’s center
- Replace the old attachment with a new attachment

- Select the widget in the design hierarchy, bring up its Constraints panel, and reset the attachment "Type" for that edge to "None".

"Circular Dependency" Error Message

Symptom: A "circular dependency" message appears after you make an attachment.

Cause and Solutions: If you add an attachment that results in a circularity involving two or more widgets, Motif detects the circularity and returns the error message. Click on "Undo." If you still get the error message, carefully inspect your layout for an attachment loop and remove one of the attachments.

"Bailed Out..." Error Message

Symptom: A "bailed out" message appears after you make an attachment.

Cause and Solutions: This Motif message indicates that your layout contains attachments that contradict one another without being circular. It usually occurs with "Self" or "Position" attachments. Use "Undo" or move the widgets remove the contradictory attachments, then reset the Form.

Widgets Overlap the Boundary of the Form

Symptom: Widgets at the edge of a Form cause breaks in the Form's boundary line.

Causes: Widgets whose edges coincide with the sides of the Form can overlap the line drawn around the Form, causing undesirable breaks in the line. This only occurs if the Form is the immediate child of a Shell. Three conditions in the Form can cause the overlap:

- A widget is attached to the edge of the Form with an offset of 0 or 1 pixel
- A widget is attached to the edge of the Form with a default offset and a vertical or horizontal spacing value of 0
- There are no attachments between the bottom or right edge of the Form and the widgets closest to those edges

Solution: Attach widgets to the top or left edge of the Form with an offset of 2 or more pixels. You can use an explicit offset, or you can set the Form's vertical and horizontal spacing resources to the offset value and use the default offsets. Make sure the widgets at the bottom and right side of the layout are attached to the edge of the Form with an offset or spacing of 2 or more pixels.

Solution: Put the Form inside another manager widget, such as another Form or a DialogTemplate. This is the simplest and most flexible solution.

25.5 Links

This section discusses problems you may encounter when you use the "Edit links" command in the Widget Menu. For additional information about links, see the *Code Generation* section that follows.

"Add" Is Disabled

Symptom: The "Add" option is grayed out.

Cause and Solution: The Link facility requires the target widget to have an explicit variable name. If the target widget is a Shell, its immediate child must also have an explicit name. SPARCworks/Visual grays out the "Add" option if it does not find explicit names. Name the appropriate widgets.

A Link Stops Working

Symptom: A link that used to work stops working. The link appears in the "Edit links..." dialog with a blank space instead of an icon.

Cause and Solution: If you change the name of a widget, SPARCworks/Visual does not automatically update links that refer to that widget and they cease to be functional. Remove the obsolete links and replace them with new ones.

Links Don't Update When You Select Another Button

Symptom: The "Links" panel doesn't behave like the resource panels. If you edit links on one button then select another button, the "Links" panel still shows the links from the previously selected button.

Cause: SPARCworks/Visual interprets the selection of any new widget as a target widget for a potential new link on the previously selected button.

Solution: Pull down the Widget Menu and select "Edit links" again to display and edit links on the second button. You do not have to close the Links panel first.

25

25.6 Code Generation

This section discusses problems you may encounter when you generate code. Some of these problems result because SPARCworks/Visual offers you so much flexibility in arranging your files. For example, you should make sure to generate Link functions in only one file, and to generate Includes in the files where they will be needed. Read the section on *Arranging Your Files* in the *Code Generation* chapter.

"No Application Shell" Warning

Symptom: SPARCworks/Visual displays a "No Application Shell" warning message when you try to generate the primary module with a *main()* program.

Cause and Solution: Your design does not contain the required Application Shell. Bring up the resource panel for the Shell of the main window in your design, click on the "Application Shell" toggle, then click on "Apply."

Links are Undefined

Symptom: Link functions are undefined at link time.

Cause and Solution: If you generate Links with your primary module, you must generate the actual code for the links - the Link functions - into one of your code files, either the primary module or a stubs file. Be sure to generate the link functions into an appropriate file.

Global Widgets Are Undefined

Symptom: Global widgets are undefined when you compile the stubs file.

Cause and Solution: Declarations of global widgets and objects are generated into the primary module, but not into the stubs file. To generate a header file that declares them, use the "Externs..." option and *#include* the resulting header file with your callbacks. This is preferable to writing your own *extern* declarations or copying the ones generated in the primary module, because it is less error-prone and more complete. The Externs file can be regenerated when necessary to reflect changes in the design.

Application Does Not Use Resources from X Resource File

Symptom: The generated application doesn't use the resource values from the generated X resource file. For example, variable names appear on labels and buttons instead of the label strings, colors are wrong, or fonts are wrong. The exact symptoms depend on which resources were generated into the X resource file and which were hard-wired.

Cause and Solution: You did not regenerate the X resource file when you regenerated code. If you have added or removed widgets from your design, default widget names may change and no longer correspond to those in the generated X resource file. Regenerate the X resource file.

Cause and Solution: X cannot find the X resource file when you run your application. You may need to rename the X resource file, usually with the same name as the application class and without a suffix. For more information, see your X documentation.

Cause and Solution: X cannot recognize the widget names in the file because a different application class name was used for the generated code file and the X resource file. Regenerate the application and the X resource file, being sure to use the same application class name for both. Be sure to use a unique application name to avoid confusion with other resource files your system may be accessing.

Default Resources Change At Run Time

Symptom: A color, font, or other resource is different at run time from that in the dynamic display.

Cause: Some resources shown in the dynamic display are inherited from SPARCworks/Visual. If not explicitly set on the resource panels, these resources may inherit values from other sources at run time, depending on the platform where the program is run.

Solution: To ensure the correct colors and fonts, set explicit values for them on the resource panels. Foreground and background colors can be set on each Shell in the design and are then inherited by all children of the Shell. Fonts can be set on BulletinBoards, derivatives of the BulletinBoard, or Shells, and apply to all their children.

Unexpected Results Occur When Widgets Share a Widget Name

Resource values can be shared among widgets with a common widget name, but only if they are read from the X resource file into the resource data base. The following rules apply:

- Sharing of resources occurs only at run time. Resource values are not shared among widgets in the dynamic display
- Hard-wired resources are not shared
- Object bindings are not shared

Symptom: Resource values are different at run time from values in the dynamic display. When a resource is generated to the X resource file, the result is different from when it is hard-wired.

Cause: These are expected results when widgets share a widget name. The dynamic display and hard-wired resource settings disregard common widget names. Resources generated into the X resource file, however, affect all widgets with a common widget name.

Symptom: A color or font is not shared among widgets with a common widget name, even if that resource was generated to the X resource file.

Cause and Solution: The color or font used an object binding instead of a color or font setting. Use a simple color or font setting, or set the resource on a common parent of the widgets that you want to share the value.

Symptom: An explicitly set resource value is overridden at run time.

Cause: If widgets share a widget name and resources are generated to the X resource file, only one value is used even if more than one was set.

Solution: Generally it is better to avoid common widget names unless widgets are to share all resource values. However, you can force any resource value to be restricted to a single widget by hard-wiring it. Use the masking toggles on the resource panels to do this.

Application Defaults A

A.1 Introduction

SPARCworks/Visual has its own set of application resource settings. This appendix briefly describes the resources you are most likely to change to suit your personal preference. They can be altered in or appended to any application resource file, according to the configuration of your system. For example, you might only want to change the SPARCworks/Visual application resource file, $VISUROOT/`app-defaults/`visu. $VISUROOT is the path to the root of the SPARCworks/Visual installation directory.

In this section, the resource names appear in **bold type** and the default values in *italic*. To turn these into a resource file setting, simply add a line to the appropriate resource file. For example, in the following line:

```
visu.autoSave: true
```

autoSave is the name of the resource and *true* is the setting.

visu contains many resources that are not mentioned here. Most of these only need to be changed if you are working in a foreign language or have some other special requirement. For information on resources not documented in this appendix, see the comments in visu.

A resource can have different settings for the large and small-screen versions of SPARCworks/Visual. Use the application name visu for the large-screen version and small_visu for the small-screen version. Resources set under the name visu also apply to small_visu, unless there is a specific setting under the name small_visu.

A

Refer to the *Configuration chapter* for information on customizing the SPARCworks/Visual interface using some resources not covered in this chapter.

A.2 General

nameFont

The font in which widget names are displayed. Default for visu:
--helvetica-medium-r-normal--12-*-*-*-*-*-*-**
Default for small_visu:
--helvetica-medium-r-normal--10-*-*-*-*-*-*-**

warnOnClose

Causes SPARCworks/Visual to warn you if you try to close a dialog with outstanding changes. Default: *true*.

warnOnSelect

Causes SPARCworks/Visual to warn you if you try to select a different widget with outstanding changes on the currently selected widget. Default: *true*.

dialogsTopLevel

Causes SPARCworks/Visual to create Top Level Shells rather than Dialog Shells for the dynamic display. This alters the way that the dialogs are iconified and their stacking properties. Default: *false*.

visu**dialogs.transient:false* produces a similar effect.

smallScreen

Makes SPARCworks/Visual use the small icons. Default: *false* for visu, *true* for small_visu.

intrinsicsHeadersPrefix

The prefix of the X intrinsics header file names. An important resource that depends on where X is installed on your system. Default: *X11*.

definitionsFileName

The file containing the specifications for the definitions which you have configured onto the palette. The value of this resource is expanded by */bin/sh* and so can contain environment variables, etc. Default: *$HOME/.xddefinitionsrc*.

A.3 Windows

windows

If this resource is set, SPARCworks/Visual will run in Windows Mode. See the *Cross Platform Development* chapter for further details. Default *false*.

mfcTextWarningBackground

The color used to indicate that a resource is not used in Windows flavor code.

mfcCarriageReturn

If this resource is set SPARCworks/Visual will generate carriage return characters as well as newline characters in files generated for Windows. Default *true*.

A.4 Filters

resourceFilter

Filter provided when generating X resource files. Default: **.res*.

uilFilter

Filter provided when generating UIL. Default: **.uil*.

cFilter

Filter provided when generating C. Default: **.c*.

stubsFilter

Filter provided when generating C stubs. Default: **.c*.

externsFilter
Filter provided when generating C externs. Default: *.*h*.

cPixmapsFilter
Filter provided when generating C pixmaps. Default: *.*h*.

uilPixmapsFilter
Filter provided when generating UIL pixmaps. Default: *.*uil*.

c++Filter
Filter provided when generating C++. Default: *.*c*.

c++StubsFilter
Filter provided when generating C++ stubs. Default: *.*c*.

c++ExternsFilter
Filter provided when generating C++ externs. Default: *.*h*.

visualC++Filter
Filter provided when generating C++ for Windows flavor. Default: *.*cpp*.

visualC++StubsFilter
Filter provided when generating C++ stubs for Windows flavor. Default: *.*c*pp.

visualC++ExternsFilter
Filter provided when generating C++ externs for Windows flavor. Default: *.*h*.

objectFileSuffix
Object file suffix used in generated Makefiles. Default: .*o*.

executableFileSuffix
Executable file suffix used in generated Makefiles. Default: empty string.

uidFileSuffix

uid file suffix used in generated Makefiles. Default: *.uid*.

A.5 Generation

makefileTemplate

Defines the default Makefile. For details, see the *Configuration* chapter.

motifMakeTemplateFile

Points to the default makefile template for Motif flavor. Default: *$*VISUROOT*/make_templates/motif*.

mmfcMakeTemplateFile

Points to the default makefile template for Motif MFC flavor. Default: *$*VISUROOT*/make_templates/mfc*.

c++BaseClassHeader

Header file for C++ that contains the base class definitions. Include quotes ("") or angle brackets (<>) as required; defaults to angle brackets. To disable, set it to an empty string. Default: *xdclass.h*.

xpmHeader

Header file for definitions required by XPM library. Include quotes ("") or angle brackets (<>) as required; defaults to angle brackets. Default: *xpm.h*.

generateXFuncCLinkage

Causes SPARCworks/Visual to generate *_XFUNCPROTOBEGIN* and *_XFUNCPROTOEND* macros around the help link externs. Default: *true*.

generateRedefineDefaultWidgetName

In C++ base class declarations, SPARCworks/Visual generates the default value for the widget name parameter to the *create()* member function: *widget_name = NULL*. In derived classes this should not be necessary as derived classes inherit such default parameter values from their base classes. However, many compilers require it if the *create()* function is called without a *widget_name* parameter. SPARCworks/Visual will never generate such code, but you may have existing source where this is the case. Default: *true*.

A.6 Help

helpKey

The key to invoke the help callback. Default: *<Key>F1*. If your keyboard has a Help key, try *<Key>Help*.

helpDir

The directory containing the help files. SPARCworks/Visual looks in this directory for the visu.*cdb* file. If you are using text help for user-defined widgets or definitions, SPARCworks/Visual looks in this directory for a *UserDocs* subdirectory. Default: *$*VISUROOT*/help where $*VISUROOT *is the path to the root of the* SPARCworks/Visual *installation directory*.

userHelpCatString

Separator string used to build help file names for user-defined widgets and definitions. Default: *"."*

Text file help for user-defined widgets is defined by file and tag pairs. The file and tag are concatenated to produce a filename relative to *helpDir/UserDocs*. The value of *userHelpCatString* is used as a separator between document and tag when creating the string. For example, if *userHelpCatString* is *"."* the help file is *document.tag*. An alternative setting would be *"/"* to produce *document/tag*. The help file is assumed to be ASCII text.

A.7 Auto Save

autoSave

Activates the auto save facility. Default: *false (*not active).

autoSaveThreshold

The number of changes made before an auto save occurs. Default *20*.

autoSaveExt

The extension to the filename added by auto save. Default *.sav*. For example, ***fred.xd*** becomes ***fred.xd.sav***.

A.8 Layout Editor

The following resources control colors in the Layout Editor:

formFillColor

The background color of the Layout Editor. Default: *OldLace*.

formStrokeColor

The color used to outline the Form in the Layout Editor. Default: *Black*.

widgetFillColor

The color used to fill the boxes representing the widgets in the Layout Editor. Default: *Blue*.

widgetStrokeColor

The color used to outline the widgets. Default: *Black*.

widgetDestinationColor

The color used to denote the last selected widget when doing align and distribute operations. This is the reference widget for the operation. Default: *Red*.

widgetSelectColor

The color used to denote selected widgets when doing align and distribute operations. These are the widgets that will be moved by the operation. Default: *Green*.

attachmentColor

The color used for the lines representing attachments in the Layout Editor. Default: *Black*.

A.9 Hierarchy Colors

The following resources are used when drawing widgets in the design hierarchy:

widgetForeground

The foreground color of bitmap-type widget icons. Default: *Black*. This refers to the color in which the pixmap or icon is drawn. For bitmap icons, it must contrast with the *widgetBackground* resource. It is unused for color pixmap icons.

widgetBackground

The background color of the widget icon. This color shows through the sections of color pixmaps that are set to color *none*. Default: *OldLace*.

highlightForeground

The foreground color used for drawing bitmap-type icons when the widget is highlighted. For bitmap icons, it must contrast with the *widgetHighlightBackground* resource. It is unused for color pixmap icons. Default: *Black*.

If you have a monochrome display, the default setting may cause some icons to show as all black. In this case set it to the same value as *widgetBackground*.

highlightBackground

The background color used when the widget is highlighted. For bitmap icons, it must contrast with the *widgetHighlightBackground* resource. Default: *Red*. This color shows through sections of color pixmaps that are set to color *none*.

If you have a monochrome display, the default setting may cause some icons to show as all black. In this case set it to the same value as *widgetForeground*.

A.9.1 Structure Colors

The following resources control the colors that indicate widgets that are designated as structures or C++ classes. They are effective only when the "Structure colors" toggle is set:

widgetFunctionBackground

Background color for widgets designated as function structures. Default: *red*.

widgetStructBackground

Background color for widgets designated as data structures. Default: *green*.

widgetClassBackground

Background color for widgets designated as C++ classes. Default: *blue*.

widgetChildrenBackground

Background color for widgets designated as Children only. Default: *purple*.

A.9.2 Background for Definitions and Instances

The following resources control displays of definitions and instances:

widgetInstanceBackground

The background color in the tree for instances. Default: *yellow*.

widgetDefinitionBackground

The background color in the tree for definitions. Default: *cyan*.

widgetInstanceBitmap

On monochrome displays only, this is the background bitmap in the tree for hierarchies that are instances. Default *25_foreground*.

widgetDefinitionBitmap

On monochrome displays only, this is the background bitmap in the tree for hierarchies that are definitions. Default *25_foreground*.

A.10 Work-arounds

tileOriginBug

Works around problems on some servers which have difficulty displaying the tree icons. Default *false*.

alternateFolding

Works around problem on some servers which cannot do the stippling for folding. Draws folded icons with a cross through them. Default *false*.

xorByInvert

In order that color map cells are not used up needlessly, SPARCworks/Visual simply does XOR drawing using whatever happens to be in the color map. This will sometimes cause interactive drawing to be invisible. You can change the mechanism to use INVERT rather than XOR which may produce different results. Default *false*.

freeStaticColors

Some servers do not allow the application to free colors cells in a static color map (more typically they simply ignore the request). Set this resource to false if you have a display with a default visual type of static (typically a VGA type screen), and you SPARCworks/Visual is crashing with an X error when it tries to free a color. Default *true*.

A.11 FrameMaker

These resources are used to specify parameters used when using FrameMaker to develop help for your application.

frameMakerBinary

The binary that is executed to display FrameMaker help files. Default *viewer*.

frameMakerTimeout

Number of seconds to wait for FrameMaker to start up before SPARCworks/Visual attempts to talk to it. Default *20*.

A.12 Configuration

See also the Configuration chapter for details on configuring the palette and toolbar.

stopList

A comma separated list of widgets that are not to be on the palette. The complete list for the Motif widgets is:

```
XmDialogShell, XmMainWindow, XmMenuBar, XmPulldownMenu, XmRadioBox,
       XmRowColumn, XmFrame, XmDrawingArea, XmBulletinBoard, XmForm,
       XmPanedWindow, XmScrolledWindow, XmMessageBox,
       XmMessageTemplate, XmCommand, XmSelectionPrompt,
       XmSelectionBox, XmFileSelectionBox, XmLabel, XmPushButton,
       XmToggleButton, XmDrawnButton, XmArrowButton,
       XmCascadeButton, XmOptionMenu, XmSeparator, XmScale,
       XmScrollBar, XmTextField, XmText, XmScrolledText, XmList,
       XmScrolledList
```

Use class names for user-defined widgets, e.g:

```
boxWidgetClass, formWidgetClass
```

Default empty.

pm_icons, pm_labels, pm_both

These are the three toggle buttons in the Palette Layout menu. Set one of them for the default layout. Default visu**pm_icons.set:true*

A

Motif MFC Reference B

B.1 Introduction

The Motif MFC library allows you to share code between Motif and Windows by providing a mapping of most of the MFC classes and methods to Motif widgets and X or UNIX calls. This chapter documents the library.

B.1.1 Using the Motif MFC

To make full use of the following information, start by checking which MFC class you are dealing with. The *Mapping Motif Widgets to Windows* section of the *Widget Reference* chapter on page 536 will give you this information. You can then look up the class in the following pages and find which methods are available.

Note that all variables and methods beginning *xd_* are specific to the Motif MFC - you can use them on Motif but not on Windows.

For information on the MFC and the methods, you should consult your MFC documentation supplied with the environment you are using on Windows.

B.1.2 Enhancing the Motif MFC

The source code for the Motif MFC is provided with SPARCworks/Visual, allowing you to add to it if you wish.

It is located in $VISUROOT/motifmfc/lib (where $VISUROOT is the path to the root of the SPARCworks/Visual installation directory). Each class has a separate source file and is commented to help you find your way around. The public headers are in *xdclass.h* which can be found in the $VISUROOT/motifmfc/h directory.

B.2 The Motif MFC Library

B.2.1 class CObject

The class *CObject* is the principal base class for the MFC library. All other classes are derived from this one.

virtual~CObject();

Destroys a CObject object.

protected CObject();

Constructs a CObject object.

virtual Widget xd_rootwidget();

virtual void xd_rootwidget(Widget xd_rootwidget);

The first version of xd_rootwidget() returns the widget pointer of the widget at the root of the hierarchy which is represented by the CObject object. The second version sets the root widget.

B.2.2 class CFrameWnd : public CWnd

The class *CFrameWnd* provides the functionality of a Windows single document interface overlapped or pop-up frame window. It is used by SPARCworks/Visual to support the ApplicationShell widget.

protected virtual
int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of XmNtitle for the Shell widget. Returns 0 if the widget has not yet been created, otherwise returns the length of the text.

protected virtual
int xd_get_window_text_length() const;

Returns the length of the widget's XmNtitle resource. Returns 0 if the widget has not yet been created.

protected virtual
void xd_set_window_text(LPCSTR lpszString);

Sets the XmNtitle and XmNiconName for the Shell widget to *lpszString*.

protected virtual
BOOL xd_show_window(int nCmdShow);

Used to implement ShowWindow for ApplicationShell. Supports SW_SHOWMINIMIZED, SW_HIDE and SW_RESTORE only.

B.2.3 class CCmdTarget : public CObject

The class *CCmdTarget* is the base class for the MFC library message-map architecture. A message map routes commands or messages to the member functions you write to handle them. This Motif version includes no functionality; the class is included only for compatibility with the Windows code.

B.2.4 class CWnd : public CCmdTarget

The class *CWnd* provides the base functionality of all window classes in the MFC library. The following MFC methods have been implemented:

CWnd();

virtual
~CWnd();

int GetWindowText(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the window text for the widget. This is implemented by calling the virtual member function *xd_get_window_text()*.

int GetWindowTextLength() const;

Gets the length of the window text for the widget. This is implemented by calling the virtual member function *xd_get_window_text_length()*.

BOOL EnableWindow(BOOL bEnable=TRUE);

Enables or disables a window. Returns 0 if the widget has not yet been created, 0 if the widget was previously enabled or non-zero if the widget was previously disabled.

void SetWindowText(LPCSTR lpszString);

Sets the window text for the widget. This is implemented by calling the virtual member function *xd_set_window_text()*.

BOOL ShowWindow(int nCmdShow);

Show, iconize (ApplicationShell or TopLevelShell only) or hide a window. Returns 0 if the widget has not yet been created; 0 if the window was previously hidden or non-zero if the window was previously visible. It is implemented by calling the virtual member function *xd_show_window()*.

void xd_call_data (XmAnyCallbackStruct *call_data);

XmAnyCallbackStruct *xd_call_data () { return _xd_call_data; }

The first version of *xd_call_data()* is used by the SPARCworks/Visual generated code to store a callback's call_data in the class. It can be retrieved in the callback method using the second version.

protected virtual
int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Used by the sub-classes to implement *GetWindowText()*.

protected virtual
int xd_get_window_text_length() const;

Used by the sub-classes to implement *GetWindowTextLength()*.

protected virtual
void xd_set_window_text(LPCSTR lpszString);

Used by the sub-classes to implement *SetWindowText()*.

protected virtual
BOOL xd_show_window(int nCmdShow);

Implements default show and hide behavior for ShowWindow. For gadgets it manages and unmanages the gadget, for widgets it sets mappedWhenManaged appropriately.

B.2.5 class CDialog : public CWnd

The class *CDialog* is the base class used for displaying dialog boxes on the screen. To make a useful class, you would normally derive another class from CDialog.

protected virtual
int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of XmNtitle for the Shell widget. Returns 0 if the widget has not yet been created, otherwise returns the length of the text and the text is placed into lpszStringBuf.

protected virtual
int xd_get_window_text_length() const;

Returns the length of the widget's XmNtitle resource. Returns 0 if the widget has not yet been created.

protected virtual
void xd_set_window_text(LPCSTR lpszString);

Sets the XmNtitle and XmNiconName for the Shell widget to *lpszString*.

protected virtual
BOOL xd_show_window(int nCmdShow);

Implements ShowWindow for TopLevelShell or DialogShell. Supports SW_SHOWMINIMIZED (TopLevelShell only), SW_HIDE and SW_RESTORE.

B.2.6 class CScrollBar : public CWnd

The class *CScrollBar* provides the functionality of a Windows scroll-bar control.

int GetScrollPos() const;

Returns 0 if the widget has not yet been created, otherwise returns XmNvalue.

void GetScrollRange(LPINT lpMinPos, LPINT lpMaxPos) const;

If the widget has been created sets *lpMinPos* and *lpMaxPos* to XmNminimum and XmNmaximum respectively.

int SetScrollPos(int nPos, BOOL bRedraw = TRUE);

If the widget has been created sets XmNvalue to nPos and returns the previous XmNvalue, otherwise returns 0.

void SetScrollRange(int nMinPos, int nMaxPos, BOOL bRedraw = TRUE);

If the widget has been created sets XmNminimum and XmNmaximum to *nMinPos* and *nMaxPos* respectively.

void ShowScrollBar(BOOL bShow = TRUE);

If the widget has been created manages or unmanages it as determined by *bShow*.

B.2.7 class CFileDialog : public CDialog

The *CFileDialog* class encapsulates the Windows common file dialog box, providing an easy way to implement File Open and File Save As dialog boxes (as well as other file selection dialog boxes) in a manner consistent with Windows standards.

```
CFileDialog (BOOL bOpenFileDialog,
    LPCSTR lpszDefExt = NULL,
    LPCSTR lpszFileName = NULL,
    DWORD dwFlags =
    OFN_HIDEREADONLY |OFN_OVERWRITEPROMPT,
    LPCSTR lpszFilter = NULL,
    CWnd* pParentWnd = NULL);
```

The constructor simply builds a CFileDialogObject. The *lpszFileName* and *lpszFilter* parameters are used to set the XmNdirSpec and XmNpattern resources of the file selection box in the *DoModal()* method. The *pParentWnd* resource should point to a CFrameWnd object.

virtual
~CFileDialog();

Destroys the CFileDialog object, freeing private class variables.

virtual
int DoModal();

Sets the XmNdirSpec and XmNpattern resources as specified in the constructor then executes a private event loop until the OK, Cancel, or Popdown callback is processed.

CString GetPathName() const;

Returns the value of the file selection box's XmNdirSpec resource.

protected virtual
void OnCancel();

Called when the user presses the Cancel button or pops down the dialog from the window menu. Sub-classes should call this method when overriding *OnCancel()* if they want the file selection to complete.

protected virtual
void OnOK();

Called when the user presses the OK button. Sub-classes should call this method when overriding *OnOk()* if they want the file selection to complete.

virtual
BOOL OnInitDialog();

Returns TRUE by default. This is overridden in SPARCworks/Visual generated code to call the create method which will create the widgets.

B.2.8 class CSplitterWnd : public CWnd

Used to implement PanedWindows.

B.2.9 class CMenu : public CObject

The class *CMenu* is a class for handling the Windows menu control.

CMenu();

Creates a CMenu object.

~CMenu();

Destroys a CMenu object.

UINT CheckMenuItem(UINT nIDCheckItem, UINT nCheck);

Sets the check state for a menu item which corresponds to a toggle button. The *nCheck* parameter specifies both the required state of the item (MF_CHECKED or MF_UNCHECKED) and the interpretation of *nIDCheckItem* (MF_BYCOMMAND and MF_BYPOSITION). These two values should be specified using bitwise OR (e.g. *menu->CheckMenuItem (ID_toggle_b, MF_BYCOMMAND | MF_CHECKED)*). The function returns -1 if the menu item is not found or is not a toggle button (note that Windows MFC will allow any menu item to be check - even a separator). The previous state (MF_CHECK or MF_UNCHECKED) is returned otherwise. If *nCheck* includes MF_BYCOMMAND any submenus are also searched.

UINT EnableMenuItem(UINT nIDEnableItem, UINT nEnable);

Enables or Disables a menu item. The *nEnable* parameter specifies both the required state of the item (MF_ENABLED or MF_GRAYED) and the interpretation of *nIDEnableItem* (MF_BYCOMMAND and MF_BYPOSITION). These two values should be specified using bitwise OR (e.g. *menu->EnableMenuItem (ID_toggle_b, MF_BYCOMMAND | MF_GRAYED)*). The function returns -1 if the menu item is not found or is a menubar, menu, separator or a cascade button and MF_BYCOMMAND is specified. The previous state (MF_ENABLED or MF_GRAYED) is returned otherwise. If *nEnable* includes MF_BYCOMMAND any submenus are also searched. Note that the state MF_DISABLED (insensitive but not grayed out) is not supported.

UINT GetMenuState(UINT nID, UINT nFlags)const;

Enables or Disables a menu item. The *nFlags* parameter specifies the interpretation of *nID* (MF_BYCOMMAND or MF_BYPOSITION). The function returns -1 if the menu item is not found or is a separator and MF_BYCOMMAND is specified. A bitwise OR of the states (MF_CHECKED, MF_UNCHECKED, MF_SEPARATOR, MF_ENABLED or MF_GRAYED) is returned otherwise. If *nEnable* is MF_BYCOMMAND any submenus are also searched. Note that in Windows MFC GetMenuState for a popup menu also returns the number of items in the high order byte. This is not supported by the Motif MFC.

BOOL TrackPopupMenu (UINT nFlags,
int x, int y,
CWnd *pWnd,
LPCRECT lpRect = 0);

This function simulates the behavior of the Windows MFC TrackPopupMenu function. The function retrieves the call_data from the window specified by *pWnd* (this will have been saved by the callback function). If the event in the call_data is a ButtonPress event the popup menu is positioned using the call_data's event (not the function parameters), the menu is managed and TRUE is returned. FALSE is returned otherwise.

void xd_register_menu(CMenu *menu);

Used by the toolkit to map IDs to menu items.

void xd_register_menu_item(UINT nIDItem, Widget item);

Used by the toolkit to map IDs to menu items.

protected
Widget xd_get_menu_item_by_position(UINT nPos);

Used by the toolkit to map IDs to menu items.

protected
Widget xd_get_menu_item_by_id(UINT nIDItem);

Used by the toolkit to map IDs to menu items.

B.2.10 class CComboBox : public CWnd

The class *CComboBox* is used to wrap an OptionMenu to provide an interface equivalent to the Windows ComboBox.

int GetCurSel() const;

Returns the (zero based) index of the currently selected item. Returns 0 if the widget has not yet been created.

int GetLBText(int nIndex, LPSTR lpszText) const;

Gets a copy of the text of the item into lpszText identified by *nIndex* and returns its length. Returns 0 if the widget has not yet been created and LB_ERR if the index is out of range.

int GetLBTextLen(int nIndex) const;

Returns the length of the text of the item identified by *nIndex*. Returns 0 if the widget has not yet been created and LB_ERR if the index is out of range.

int SetCurSel(int nSelect);

Sets the current selection to be the item identified by *nSelect*. Returns 0 if the widget has not yet been created and LB_ERR if the index is out of range. Otherwise returns the index of the selected item. Note unlike Windows MFC passing *nSelect* as -1 to clear the selection is not supported.

protected virtual
int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Returns the text of the selected item in lpszStringBuf. Returns 0 if the widget has not been created, LB_ERR if there is no selected item, the length of the text otherwise.

protected virtual
int xd_get_window_text_length() const;

Returns -1 if the widget has not yet been created, and 0 if it has. This corresponds to MFC behavior.

protected virtual
void xd_set_window_text(LPCSTR);

This is a noop for CComboBox.

B.2.11 class CStatic : public CWnd

The class *CStatic* implements a Windows static control which is a simple text field, implemented with a Label widget.

protected virtual
void xd_set_window_text(LPCSTR lpszString);

Sets the XmNlabelString resource for the widget to an XmString created with *XmStringCreateLocalized()* using *lpszString*.

protected virtual
int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of XmNlabelString for the widget into lpszStringBuf. If the widget has not yet been created 0 is returned, otherwise the length of the string is returned.

protected virtual
int xd_get_window_text_length() const;

Returns the length of the XmNlabelString resource for the widget. If the widget has not yet been created 0 is returned.

B.2.12 class CButton : public CWnd

The *CButton* class provides the functionality of Windows button controls and is implemented with either a PushButton or a ToggleButton.

int GetCheck() const;

Gets the check state of a button. Returns 0 if the widget has not yet been created, is not a toggle button or is a toggle button and is not set. Returns 1 if the toggle button is set. Note that the Windows MFC can return a value 2 (indeterminate state) which is not supported by the Motif MFC.

void SetCheck(int nCheck);

Sets the state of a toggle button according to *nCheck*. This is a noop for push buttons.

protected virtual void xd_set_window_text(LPCSTR lpszString);

Sets the XmNlabelString resource for the widget to an XmString created with *XmStringCreateLocalized()* using *lpszString*.

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of XmNlabelString for the widget into lpszStringBuf. If the widget has not yet been created 0 is returned, otherwise the length of the string is returned.

protected virtual int xd_get_window_text_length() const;

Returns the length of the XmNlabelString resource for the widget. If the widget has not yet been created 0 is returned.

B.2.13 class CBitmapButton : public CButton

The class *CBitmapButton* implements a button with a bitmap instead of text.

B.2.14 class CListBox : public CWnd

The class *CListBox* provides the functionality of a list box which can display a list of items that the user can view and select.

CListBox();

Initializes the private data.

virtual Widget xd_rootwidget();

virtual void xd_rootwidget(Widget xd_rootwidget);

Overrides the methods in CObject so that the class can distinguish between List and ScrolledList.

virtual Widget xd_listwidget();

Returns the list widget for the object. For an ordinary List this is the same as the root widget, however, it is different for a ScrolledList.

int DeleteString(UINT nIndex);

Deletes the list item identified by *nIndex* (zero based). Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range or the number of items remaining in the list.

int GetCount() const;

Returns 0 if the widget has not yet been created, otherwise returns the number of items in the list (XmNitemCount).

int GetCurSel() const;

Gets the index of the currently selected item in a single select list (XmNselectionPolicy is XmSINGLE_SELECT or XmBROWSE_SELECT). Returns 0 if the widget has not yet been created, LB_ERR if the list is a multiple selection list or it has not selected item. Otherwise the index of the selected item is returned. Note that the Windows MFC returns an arbitrary positive value if the list is a multiple selection list, Motif MFC always returns LB_ERR.

int GetSel(int nIndex) const;

Returns the selection state of the item indicated by *nIndex*. Returns 0 if the widget has not yet been created or if the item is not selected. Returns LB_ERR if the index is out of range, or a positive value if the item is selected.

int GetSelCount() const;

Returns the number of selected items in a multiple selection list. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list, or the number of indices copied otherwise.

int GetSelItems(int nMaxItems, LPINT rgIndex) const;

Gets the indices of the selected items in a multiple selection list and copies them into the array *rgIndex*. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list, or the number of selected items otherwise.

int GetText(int nIndex, LPSTR lpszBuffer) const;

Gets the text of an item identified by *nIndex* into lpszBuffer. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, the length of the text otherwise.

int GetTextLen(int nIndex) const;

Gets the length of the text of an item identified by *nIndex*. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, the length of the text otherwise.

int GetTopIndex() const;

Returns the index of the item that is visible at the top of the list. Returns 0 if the widget has not yet been created.

int InsertString(int nIndex, LPCSTR lpszItem);

Inserts an item into the list at the position given by *nIndex*, If nIndex is -1 the item is appended at the end of the list. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, the position at which the item was inserted otherwise.

void ResetContent();

Removes all the items from a list.

int SelItemRange(BOOL bSelect, int nFirstItem, int nLastItem);

Selects or deselects, according to *bSelect*, a range of items in a multiple selection list. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list, a value other than LB_ERR otherwise.

int SetCurSel(int nSelect);

Select an item, identified by *nSelect*, in a single selection list and scroll it into view. If *nSelect* is -1, the selection is cleared. Returns 0 if the widget has not yet been created, LB_ERR if the list is a multiple selection list or the index is out of range, a value other than LB_ERR otherwise.

int SetSel(int nIndex, BOOL bSelect = TRUE);

Selects or deselects, according to *bSelect*, an item in a multiple selection list. If nIndex is -1 all items are selected or deselected. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list or the index is out of range, a value other than LB_ERR otherwise.

int SetTopIndex(int nIndex);

Scroll the list to make the item identified by *nIndex* visible. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, a value other than LB_ERR otherwise.

B.2.15 class CEdit : public CWnd

The class *CEdit* provides the functionality of a Windows edit control which is a rectangular window in which the user can enter text. Implemented with either a Text or TextField widget.

CEdit();

Initializes the private data.

virtual Widget xd_rootwidget();

virtual void xd_rootwidget(Widget xd_rootwidget);

Overrides the methods in CObject so that the class can distinguish between Text and ScrolledText.

virtual Widget xd_textwidget();

Returns the text widget for the object. For ordinary Text this is the same as the root widget, however, it is different for ScrolledText.

void Clear();

Deletes the currently selected text (*XmTextRemove()*).

void Copy();

Copies the currently selected text to the clipboard (*XmTextCopy()*).

void Cut();

Deletes the currently selected text and copies it to the clipboard. (*XmTextCut()*).

void GetSel(int& nStartChar, int& nEndChar) const;

Gets the start and end of the selected text. Returns start and end as 0 if there is no selected text.

void LimitText(int nChars = 0);

Limit the number of characters that can be typed. If *nChars* is 0 the limit is set to maximum.

void Paste();

Insert data from the clipboard into the text widget (*XmTextPaste()*).

void ReplaceSel(LPCSTR lpszNewText);

Replaces the current selection with the text supplied in *lpszNewText*. If there is no selection the text is inserted at the insert cursor position.

BOOL SetReadOnly(BOOL bReadOnly = TRUE);

Sets the XmNeditable resource of the widget to be !*bReadOnly*. Returns 0 if the widget has not yet been created, otherwise returns 1.

void SetSel(int nStartChar, int nEndChar, BOOL bNoScroll = FALSE);

Sets the current selection to the text specified by *nStartChar* and *nEndChar*. Will also set XmNautoShowCursorPosition to !*bNoScroll*.

protected virtual
void xd_set_window_text(LPCSTR lpszString);

Sets the value for the widget to *lpszString* (*XmTextSetString()*).

protected virtual
int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the text from the widget (*XmTextGetString()*) into lpszStringBuf. If the widget has not yet been created 0 is returned, otherwise the length of the string is returned.

protected virtual
int xd_get_window_text_length() const;

Returns the length of the widget's text. If the widget has not yet been created 0 is returned.

B.2.16 class CWinApp : public CCmdTarget

The class *CWinApp* is the base class from which you derive a Windows application object for initializing your application and for running the application.

CWinApp(const char* pszAppName = NULL);

Constructs a CWinApp object.

const char* m_pszAppName;

The name of the application. This comes from the parameter passed to the CWinApp constructor.

int m_nCmdShow;

Defaults to SW_RESTOR.

CWnd *m_pMainWnd;

The main window (ApplicationShell) of the application.

Display *xd_display();

void xd_display(Display *display);

These two functions store and retrieve the applications Display connection.

char **xd_argv() const;

void xd_argv(char **argv);

These two functions store and retrieve the argv parameters passed into *main()*.

int xd_argc() const;

void xd_argc(int argc);

These two functions store and retrieve the argc parameter passed into *main()*.

char *xd_app_class() const;

void xd_app_class(char *app_class);

These two functions store and retrieve the application class name used in *XtOpenDisplay()*.

B.2.17 CWinApp* AfxGetApp()

Returns the one, and only, instance of a CWinApp object.

B.3 Linking Error with Some Compilers

Some C++ compilers will fail to link and produce the following sort of errors:

```
Undefined symbol
CWnd::xd_get_window_text_length(void) const
CFrameWnd::xd_get_window_text_length(void) const
CButton::xd_get_window_text_length(void) const
CButton::__vtbl
CMenu::xd_register_menu_item(unsigned int, _WidgetRec*)
CDialog::xd_show_window(int)
CDialog::xd_get_window_text_length(void) const
CDialog::xd_set_window_text(const char*)
CEdit::__vtbl
```

B.3.1 The problem

The compiler requires there to be an implementation of the copy constructor.

B.3.2 The remedy

If this occurs, do the following:

1. **Find the header file *xdclass.h* in $VISUROOT/motifmfc/h (where $VISUROOT is the path to the root of the SPARCworks/Visual installation directory).**

2. **Locate the following lines:**

```
private:
    // Certain C++ compilers (eg gcc 2.5) require there to be an
    // implementation of the copy constructor. If your application
    // fails to link try using the second version of the constructor
    CObject(const CObject& objectSrc);
    // no default copy
    //CObject(const CObject& objectSrc) { abort();}
    // no default copy
```

3. **Follow the instructions so that the first line beginning `CObject` is commented out and the second has the comment markers removed:**

```
    // CObject(const CObject& objectSrc);
    // no default copy
    CObject(const CObject& objectSrc) { abort();}
    // no default copy
```

4. **Add the following include line somewhere above the line containing the call to `abort`:**

```
    #include <stdlib.h>
```

5. **Recompile your generated code.**
   The application should now link successfully.

B

Further Reading C

C.1 Introduction

This section supplies further details on the books which are referred to in this manual and others which we recommend for additional reading.

We list ISBN numbers but suggest you consult your book supplier for the latest editions.

C.2 Books Mentioned In This Manual

Kernighan, B.W. and Ritchie, D.M., *The C Programming Language.* Prentice Hall, 1978

First edition 1978 ISBN 0-13-110163-3

Second edition 1988 ISBN 0-13-110362-8

Open Software Foundation, *OSF/Motif* (5 vols). Prentice Hall, 1990, 1991, 1992.

OSF/Motif Style Guide 1993 ISBN 0-13-643123-2

OSF/Motif Programmer's Guide 1993 ISBN 0-13-643107-0

OSF/Motif Programmer's Reference 1993 ISBN 0-13-643115-1

OSF/Motif User's Guide 1993 ISBN 0-13-643131-3

Application Environment/Specification (AES) User Environment, Revision C 1992 ISBN 0-13-043621-6

O'Reilly and Associates, *The X Window System Series* (8 vols). O'Reilly and Associates, Inc., 1988, 1989, 1990, 1991, 1992, 1993

Volume 0:1992 ISBN 1-56592-008-2

Volume 1:1992 ISBN 1-56592-002-3

Volume 2:1992 ISBN 1-56592-006-6

Volume 3M:1993 ISBN 1-56592-015-5

Volume 4M:1992 ISBN 1-56592-013-9

Volume 5:1992 ISBN 1-56592-007-4

Volume 6A:1994 ISBN 1-56592-016-3

Volume 6B:1993 ISBN 1-56592-038-4

Volume 7:1993 ISBN 0-937175-87-0

Volume 8:1992 ISBN 0-937175-83-8

C.3 Books on X and Motif

The following books are useful books on the X Window System and OSF/Motif.

C.3.1 Beginner/ Intermediate

Berlage, Thomas, *OSF/Motif: Concepts and Programming.* Addison-Wesley, 1991. ISBN 0-201-55792-4

Jones, Oliver, *Introduction to the X Window System*. Prentice Hall, 1989. ISBN 0-13-499997-5

Rost, Randi J., *X and Motif Quick Reference Guide*. Digital Press, 1993. ISBN 13-972746-9

Young, Douglas A., *X Window System: Programming and Applications with Xt, 2nd OSF/Motif Edition*. Prentice Hall, 1994. ISBN 0-13-123803-5

Young, Douglas A., *OSF/Motif Reference Manual*. Prentice Hall, 1990. ISBN 0-13-642786-3

C.3.2 Intermediate/ Advanced

Asente, Paul J. and Swick Ralph R., *X Window System Toolkit*. Digital Press, 1990. ISBN 1-55558-051-3

Scheifler, R.W. and Gettys, J., *X Window System, 3rd edition*. Digital Press, 1992. ISBN 13-971201-1

George, Alistair and Riches, Mark, *Advanced Motif Programming Techniques*, Prentice Hall, 1993. ISBN 0-13-219965-3

A more comprehensive listing of publications is posted monthly to the X newsgroup on *usenet* by Ken Lee of DEC.

C.4 Books on C++ and Object Oriented Programming

Stroustrup, Bjarne, *The C++ Programming Language, 2nd edition*. Addison-Wesley Publishing Company, 1991. ISBN 0-201-53992-6

Young, Douglas, *Object-Oriented Programming with C++ and OSF/Motif*. Prentice-Hall, 1992. ISBN 0-13-630252-1

C

Glossary

accelerator A key or key combination that immediately executes a command from a menu.

accelerator text Text that appears on the buttons of a menu to remind the user of an accelerator.

action The name (such as “Arm,” “Activate,” or “Help”) of a widget’s prescribed response to an event, as defined in the widget’s translations table. Actions are mapped to functions in the widget’s action table.

action routine A function that performs an action. Many action routines are supplied with Motif. SPARCworks/Visual users can also write their own action routines.

action table A table associated with a widget that maps actions to the action routines that perform them.

application class name The name given to the Application Shell of a generated application. This name is used as a title for that Shell’s window and to identify resources that belong to that application. In SPARCworks/Visual, the application class name is assigned at code generation time.

Application Shell The type of Shell widget that is used for the primary window in the application.

attachment A constraint fixing one side of a widget to one side of a sibling widget or to one side of its parent layout widget. Attachments can be made at a fixed distance or at a percentage of the layout widget’s dimension.

button box The area reserved for buttons at the bottom of a composite widget such as a MessageBox, DialogTemplate, or FileSelectionBox.

C++ class widget A widget in the design hierarchy that is designated as a C++ class. In the generated code, SPARCworks/Visual defines a C++ class for the widget, with named descendant widgets as members of the class.

callback A field of a widget structure that designates a list of callback functions and a user action. When the user action occurs on that widget, the functions on the callback list are executed.

callback function One of the functions on a callback list.

callback list A group of functions (callback functions) associated with a callback.

children The widgets that are managed by, and (if visible) contained within the boundaries of, a parent widget. In SPARCworks/Visual, children widgets appear below their parents in the design hierarchy.

Children Only widget A widget which SPARCworks/Visual treats as a place-holder when generating code. No code is generated for the Children Only widget itself, but code is generated for any of its descendants that are designated as data structures, function structures, or C++ classes.

circular attachment In Form layout, an attachment of Widget A to Widget B, when Widget B is attached to Widget A. Attachment of Widget A to B, B to C, and C to A, or any larger loop, is also considered circular. Circular attachments are not allowed in Motif.

class hierarchy The abstract hierarchy of widget classes in Motif. Class hierarchy is distinguished from design hierarchy.

client data The single parameter that can be passed to a callback function.

code prelude Lines of code supplied in a SPARCworks/Visual dialog and inserted at specific points in the generated code.

color object Association of a name with a color.

compound string A Motif data structure that combines a text string with font and direction information.

constraint resources 1) The resources displayed on the Constraints panel. These resources can be viewed for any child of a constraint widget. 2) In general, any resources of a widget that control its children's sizes or positions.

constraint widget One of the two types of widgets (the Form and PanedWindow) whose children have a Constraints panel.

construction area The drawing area on the main SPARCworks/Visual screen in which the design hierarchy is displayed.

container widget A widget whose main purpose is to contain and organize its children.

converter A function or set of functions used to convert text entries on the resource panel to numeric resource values. Needed for certain types of user-defined widgets.

Core resource panel The special resource panel that lets you set the resources of the Core, Primitive, and Manager superclasses.

Core widget The broad superclass from which all widget classes are derived.

creation procedure A function generated by SPARCworks/Visual that creates a Shell widget with its children.

data structure A widget for which SPARCworks/Visual generates a *typedef* for a data structure, and a creation procedure that sets up that type of structure and returns a pointer to it.

derived widget class A widget class that is below another class in the class hierarchy. The derived class possesses all attributes of the classes above it, plus specialized attributes of its own.

design hierarchy The hierarchy of individual widget instances that makes up the design for an interface. Distinguished from class hierarchy.

detail The last field in an event specification for a translation, normally used to specify which key must be pressed.

Dialog Shell The type of Shell widget used for subsidiary windows. Dialog Shells cannot be iconified independently of their parent Shells.

dynamic display The working version of the design that SPARCworks/Visual creates dynamically as you build and edit the design hierarchy.

editing area The drawing area on the Layout Editor screen in which an editable sketch of the layout is displayed.

editing modes The designated behavior of mouse button 1 in the Layout Editor, as controlled by the radio buttons on the left side of the screen. The modes include “Move,” “Attach,” “Resize,” “Self,” and “Position.”

enumeration resource A resource that has a limited set of possible values. SPARCworks/Visual displays enumeration resources on the “Settings” page of resource panels.

event An element of user input such as a key press or button press.

fold A display command in SPARCworks/Visual that makes the folded widget's children not appear on the screen, thus saving space.

font object Association of a name with a font or a list of fonts.

Form attachment An attachment of one side of a widget to one side of its parent Form at a fixed horizontal or vertical offset. The offset remains the same when the Form resizes.

function structure A widget for which SPARCworks/Visual produces a separate creation procedure in the generated code.

gadget An alternative version of certain widgets derived from the Primitive class. Unlike widgets, gadgets do not require the internal creation of a window for each instance. Use of gadgets instead of widgets may or may not be advantageous, depending on your system.

graying out Fuzzy display of an icon, pushbutton, or menu option. In SPARCworks/Visual, graying out denotes that the command is inactive.

hard-wired resource A resource value that is generated into the source code, not into the X resource file. Hard-wired resource values cannot be changed without remaking the application.

inherit To possess the attributes of a superclass. Derived widget classes are said to inherit from their superclasses.

input focus Indicates the window or component within a window that receives keyboard input. Sometimes called keyboard focus.

instance An individual widget data structure. Widget instances are specific examples of widget classes. To instantiate a widget means to create an instance of a widget class.

keysym A string used to identify a key in the detail field of a translation.

layout Geometric arrangement of widgets in an interface.

Layout Editor The interactive screen editor used for setting constraint resources for the Form, BulletinBoard, or DrawingArea widget.

layout widget Any of the widgets that can be used with the Layout Editor: a Form, BulletinBoard, or DrawingArea.

link A pre-defined callback provided by SPARCworks/Visual.

link function The function code executed by a link.

Manager widget A broad superclass in the Motif class hierarchy, from which most container widgets are derived. The Manager widget is derived from the Core widget.

masking toggle The unlabeled toggle next to each resource in the resource panels, used to designate which resources are generated into the X resource file and which are hard-wired into the code.

mnemonic A single character (often the initial character) of a menu or menu selection, which initiates the selection when the menu is displayed and the character is pressed on the keyboard.

module heading Lines of code inserted at the beginning of the generated primary module and the stubs file.

modifier list A field in the event specification of a translation that specifies whether modifier keys (such as <*Ctrl*> and <*Shift*>) must be pressed or not to cause the event.

module prelude 1) Lines of code inserted just after the SPARCworks/Visual generated header and *#include* directives. 2) A general term meaning either a module prelude or module heading.

offset The fixed distance, in pixels, between two attached widgets, or between a widget and the side of the layout widget to which it is attached.

originate To belong to a certain widget; said of attachments. Attachments control the behavior of the widget with which they originate.

page selector The options menu at the top of some resource panels that lets you move from one page of resources to another.

parent A widget that manages and determines the layout of its children. In SPARCworks/Visual, parent widgets are shown above their children in the design hierarchy.

pixmap object Association of a name with a pixmap.

position attachment Attachment of one side of a widget at a specified percentage of the width or height of its parent Form. This type of attachment adjusts to the current Form dimensions.

pre-create prelude A code prelude inserted just before the given widget is created.

pre-manage prelude A code prelude inserted after the given widget is created but before it is managed. Commonly used to set up client data for callbacks.

primary module The code module generated by SPARCworks/Visual that contains the creation procedures for your interface.

Primitive widget In the Motif class hierarchy, a broad superclass from which all the button-type widgets and several other classes are derived. The Primitive widget is derived from the Core widget.

radio buttons Toggle buttons grouped inside a RadioBox, or inside a Menu or RowColumn with the "Radio behavior" resource set to "Yes." Only one radio button in the group may be selected at a time.

resource A settable field in a widget data structure. Resources control many aspects of a widget's appearance and behavior. Resources can be set by the designer or the user, or both.

resource panel An interactive screen in SPARCworks/Visual that lets you specify resource values for a widget.

resource prelude A prelude inserted at the beginning of the generated X resource file. Commonly used to add loose resource bindings for the entire application rather than for individual widgets.

selected widget The widget whose icon is currently highlighted in the design hierarchy. A widget must be selected before anything can be done to it in SPARCworks/Visual.

simple font object Association of name with a single font.

source file One of the code files generated by SPARCworks/Visual. When contrasted to the "resource file," this term refers to the primary module.

stubs file A generated file containing *#include* statements, function declarations and empty braces for callbacks.

subclass A widget class that is derived from another class.

superclass A widget class from which another class is derived.

tag Hypertext help information, consisting of a document name and a hypertext marker within the document.

Top level Shell The type of Shell widget used for primary windows in a design other than the main application window.

translation A mapping of an event, such as a key or button press, or a sequence of events, to an action.

translations table The list of translations associated with a widget.

user action A predefined set of events, such as keystrokes or button presses, that triggers a callback.

variable name The name used to identify a widget's data structure in the generated code. This is a C variable name, so it must not be the same as the variable name of any other widget, any other variable name or function name in your application, or any C code word.

widget One of the predefined data structures in the Motif toolkit, or other toolkits, that are used as building blocks for graphical user interfaces.

widget attachment An attachment of one widget to another widget within the Form.

widget class A specific type of widget.

widget name The name used to distinguish a widget instance in the X resource file. This name does not have to be the same as the variable name and does not have to be unique.

widget palette The area on the main SPARCworks/Visual screen that shows icons representing the available widget classes.

window holding area The area at the upper right of the main SPARCworks/Visual screen that displays one Shell icon for each window in the design.

work area The central area of a composite widget such as a MessageBox, DialogTemplate, or FileSelectionBox, which can contain one child widget of any type.

X resource file An editable file generated by SPARCworks/Visual, containing some or all explicit resource values for the design.

XmString The Motif compound string structure.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Copyright 1995 Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100 U.S.A.

Tous droits réservés. Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, et la décompliation. Aucune partie de ce produit ou de sa documentation associée ne peuvent Être reproduits sous aucune forme, par quelque moyen que ce soit sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il en a.

Des parties de ce produit pourront etre derivees du système UNIX®, licencié par UNIX System Laboratories, Inc., filiale entierement detenue par Novell, Inc., ainsi que par le système 4.3. de Berkeley, licencié par l'Université de Californie. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fourmisseurs de Sun.

LEGENDE RELATIVE AUX DROITS RESTREINTS: l'utilisation, la duplication ou la divulgation par l'administration americaine sont soumises aux restrictions visées a l'alinéa (c)(1)(ii) de la clause relative aux droits des données techniques et aux logiciels informatiques du DFARS 252.227-7013 et FAR 52.227-19. Le produit décrit dans ce manuel peut Être protege par un ou plusieurs brevet(s) americain(s), etranger(s) ou par des demandes en cours d'enregistrement.

MARQUES

Sun, Sun Microsystems, le logo Sun, SunSoft, le logo SunSoft, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+ et NFS sont des marques deposées ou enregistrées par Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. UNIX est une marque enregistrée aux Etats- Unis et dans d'autres pays, et exclusivement licenciée par X/Open Company Ltd. OPEN LOOK est une marque enregistrée de Novell, Inc. PostScript et Display PostScript sont des marques d'Adobe Systems, Inc.

Toutes les marques SPARC sont des marques deposées ou enregitrées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. SPARCcenter, SPARCcluster, SPARCompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, et UltraSPARC sont exclusivement licenciées a Sun Microsystems, Inc. Les produits portant les marques sont basés sur une architecture développée par Sun Microsystems, Inc.

Les utilisateurs d'interfaces graphiques OPEN LOOK® et Sun™ ont été développés par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnait les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique, cette licence couvrant aussi les licenciés de Sun qui mettent en place OPEN LOOK GUIs et qui en outre se conforment aux licences écrites de Sun.

Le système X Window est un produit du X Consortium, Inc.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A REPONDRE A UNE UTILISATION PARTICULIERE OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

CETTE PUBLICATION PEUT CONTENIR DES MENTIONS TECHNIQUES ERRONEES OU DES ERREURS TYPOGRAPHIQUES. DES CHANGEMENTS SONT PERIODIQUEMENT APPORTES AUX INFORMATIONS CONTENUES AUX PRESENTES. CES CHANGEMENTS SERONT INCORPORES AUX NOUVELLES EDITIONS DE LA PUBLICATION. SUN MICROSYSTEMS INC. PEUT REALISER DES AMELIORATIONS ET/OU DES CHANGEMENTS DANS LE(S) PRODUIT(S) ET/OU LE(S) PROGRAMME(S) DECRITS DANS DETTE PUBLICATION A TOUS MOMENTS.