OpenWindows Developer's Guide: OLIT Code Generator Programmer's Guide

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

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Preface. xiii

1. Introduction. 1

Devguide and Golit 1

Names and Terminology. 2

Installing Devguide and Golit 2

How you interact with Devguide and Golit. 2

How Golit and libgolit Work 3

2. Getting Started with Golit 5

Summary of How to Create an Application with Golit 5

Designing an Interface in Devguide 6

Configuring Devguide to Generate GIL files for Golit 6

Using Connections to Handle Events 7

Generating User Interface Code from GIL Files. 8

Generating Code from the Command Line 8

Generating Code with the Code Generator Tool 8

Golit Generated Files. 11
Compiling and Testing Interface Code 12
Integrating Application Code with Golit Interface Code 13
Setting Widget Resources 13
Regenerating Code for a Modified Interface 14
A Simple Example 15
3. Tutorial 21
Building an Interface 23
Creating a Directory for GIL and Project Files. 23
Creating the Base Window Interface 24
Creating the Base Window 24
Adding a Control Area to the Base Window 25
Adding Buttons to the top_controls Control Area. 25
Adding a Text Pane to the Tutorial Base Window 27
Creating Layered Control Areas 28
Adding Controls to the Layered Control Areas. 29
Saving the Tutorial Base Window Interface 31
Creating a Menu 31
Creating the Pop-up Window Interface 32
Creating the Pop-up Window 33
Adding a Control Area to the Pop-up Window. 34
Adding a Slider to the Control Area 34
Saving the Pop-up Window Interface. 35
Creating and Saving a Project 35

Setting up Connections 35

Creating a Connection to Display the Format Menu. 36

Creating Connections to Switch between Layered Control Areas 37

Creating a Connection to Display the Text Size Pop-up Window 37

Creating a Connection to Set the Text Size with the Slider. 38

Creating Connections to Set the Font and Style. 39

Experimenting with Test Mode 39

Generating Interface Source Code 40

Modifying the Golit-Generated Code 41

Creating a Header File 41

Customizing the tutorial.c File. 41

Customizing the tutorial_main_stubs.c File. 43

Customizing the tutorial_popup_stubs.c File 45

Compiling and Running the Tutorial Application. 46

4. Golit Functionality in Detail. 47

Designing Golit Interfaces in Devguide 47

Handling Events With Connections 47

Setting up Create Procs to Conduct Your Own Widget Creation 50

Using Golit to Generate Code 50

GolitCommand-Line Options 51

Files Generated for a Single GIL File. 52

_ui.c File 54

_ui.h File 56
_stubs.c. 57
Makefile 58
Files Generated for a Project. 59
<project_name>.c File 59
<project_name>.h File. 59
<project_name>.make File 60
Regenerating Code for a Modified Interface 60
Regenerating Code for an Individual GIL File. 60
Regenerating Code for a Project 60
Removing Obsolete Callbacks 61
Integrating Your Application Code with the Interface Code . . 61
Changing Widget Resources 62
Instantiating Parts of the Widget Tree with libgolit 66
Using Create Procs to Provide Your Own Widget Creation 70
Compiling. 77
5. Advanced Topics 79
Specifying an Icon for an Application. 79
Making a Text Field Read-Only 81
Specifying a Label Image for a Button 81
6. Internationalization 83
Overview of Internationalization Concepts 83
Levels of Internationalization 83
Golit Support for Internationalization 85

Generating Code for an Internationalized Application 85
Using Golit Command Line Options 85
Generated Resource Files 85
Level-3 Resources 86
Common Resources 86
PopupWindowShell Resources 86
TextEdit Resources. 87
Scrollbar Resources 87
Using XFILESEARCHPATH 88
A. Files Shipped with Golit 89
The Bin Subdirectory. 89
The Demo Subdirectory 89
The Include Subdirectory 89
The Lib Subdirectory. 90
The Man Subdirectory. 90
The Src Subdirectory 90
The Doc Subdirectory 90
B. Unsupported Devguide Features 91
Bold Labels 91
Help 92
Connections Between Base Windows 92
Miscellaneous Unsupported GUI Elements 92
C. libgolit Library Function Reference 95
GolitNameToWidget() 95

GolitFetchLastWidgetID() 95
GolitMergeArgLists() 96
GolitFetchWidget() 96
GolitFetchWidgetUnmanaged() 97
GolitFetchChildren() 98
GolitFetchChildrenUnmanaged() 99
GolitFetchPopups() 100
GolitFetchPopupsUnmanaged() 100
GolitFetchWidgetHier() 101
GolitFetchWidgetHierUnmanaged() 102
GolitFetchShellHier() 103

Index 105

Figures

Figure 1-1 Devguide Tools and Files 1

Figure 1-2 GolitGenerated Source Files. 3

Figure 2-1 The Devguide Defaults Popup Window 7

Figure 2-2 Code Generator Properties Window 10

Figure 2-3 Example Interface as it Appears in Devguide. 15

Figure 2-4 Connections Manager Window. 16

Figure 2-5 Generated stubs.c File for ex1 17

Figure 2-6 Generated _stubs.c File for ex1 (main) 18

Figure 2-7 Callback Modified to Increment a Gauge Each Time a Button is Pressed 19

Figure 3-1 The Tutorial Application 22

Figure 3-2 The Completed Base Window Interface Displayed in the Browser 24

Figure 3-3 Example Interface with top_controls and Buttons 27

Figure 3-4 Example Interface with Text Pane 28

Figure 3-5 Example Interface with Layered Control Areas 29

Figure 3-6 Example Interface with Style Controls Displayed 31

Figure 3-7 The Pop-up Window Interface Displayed in the Browser . . . 33

Figure 4-1 Overview of Golit Files for a Single GIL File. 53

Figure 4-2 Example Widget Tree for a Base Window 55

Figure 4-3 Example Widget Tree for a Base Window with a Footer Panel 56

Figure 4-4 Generated Callback that Displays a Popup Window 68

Figure 4-5 Callback that Displays a New Popup Window for each Button Press. 69

Figure 4-6 Template for the Create Proc, `create_textfield`() 71

Figure 4-7 Create Proc Modified to Register Help for a TextField Widget 72

Figure 4-8 Create Proc Modified to Register Help for both a Text Field and its Caption 73

Figure 4-9 Create Proc Templates for a Checkbox and a Setting 74

Figure 4-10 Checkbox and Setting Create Procs Modified to Change Fonts 76

Figure 5-1 Base Window Create Proc Modified to Display Icon. 80

Figure 5-2 Text Field Create Proc Modified to Make the textEdit Widget Read-Only 81

Figure 5-3 Button Create Proc Modified to Display Icon Label. 82

Tables

Table 4-1 When Items and the Resources Generated for Them. 49

Table 4-2 Devguide Panes and Windows and their Golit Widget Hierarchies 64

Table 4-3 Devguide Control Objects and their Golit Widget Hierarchies 65

Preface

This manual describes how to generate OPEN LOOK™ Intrinsics Toolkit (OLIT) user interfaces from Devguide GIL files. It also provides instructions on how to use Devguide to take advantage of Golit's special features.

Who Should Use This Book

This manual is for applications programmers who want to use Golit to generate user interface (UI) code and integrate this code with their own application code.

Before You Read This Book

Before you read this manual, you should be familiar with the OPEN LOOK User Interface, Devguide, and the C programming language. To use some of the advanced features described in this manual, you should also understand the how to create user interfaces with the OLIT toolkit.

Before you read this manual, you should read the following documents:

- *Software Developer Kit Installation Guide* (which includes instructions for installing Devguide)
- *OpenWindows Developer's Guide: User's Guide*, including especially Appendix D, "Devguide 3.0.1 Release Notes," which contains the latest information about changes to and problems with Devguide and GXV

You may also want to consult the following documents before going further:

- *Solaris Roadmap*
- *Solaris 2.4 Introduction*
- *X Window System Programming and Applications with Xt - OPEN LOOK edition*, by John Pew, published by Prentice Hall, 1992

How This Book Is Organized

The following is a brief description of each chapter of this manual.

Chapter 1, "Introduction" provides an overview of Golit and how to use it.

Chapter 2, "Getting Started with Golit" gets you acquainted with how Golit works. It shows you how to generate user interface code and how to compile this code with your application code. It provides a very simple example.

Chapter 3, "Tutorial" describes how to use Golit and Devguide to build a complete OpenWindows application.

Chapter 4, "Golit Functionality in Detail" describes Golit, the files it generates, and the libgolit library routines. If you want to write your own widget creation routines, you should read this chapter.

Chapter 5, "Advanced Topics" provides examples of callbacks that perform advanced functions, such as using icons and glyphs.

Chapter 6, "Internationalization" describes how to internationalize your Golit application.

Appendix A, "Files Shipped with Golit" lists the directory structure and the files on the Golit distribution medium.

Appendix B, "Unsupported Devguide Features" lists features available in Devguide that are not supported by Golit.

Appendix C, "libgolit Library Function Reference" lists the syntax and descriptions of libgolit runtime library functions.

Related Books

For more reference information on the OLIT toolkit and the Xt Intrinsics, consult:

- The *OPEN LOOK Intrinsics Toolkit (OLIT) Widget Set Reference Manual*
- *Xt Intrinsics Reference Manual*, O'Reilly & Associates, Inc., 1991

For background information on knowledge assumed by this manual, consult:

- *Solaris 2.4 Introduction*, *Solaris User's Guide*, and the *Solaris Advanced User's Guide* for information about the UNIX® and SunOS operating systems
- The *OpenWindows Reference Manual* for information on working in the OpenWindows environment
- *The OPEN LOOK Graphical User Interface Application Style Guidelines* by Sun Microsystems, Inc. (published by Addison Wesley, Inc.), for information about the elements that constitute user interfaces in the OpenWindows environment and the rules and suggestions for creating OPEN LOOK® user interfaces
- *The C Programming Language* by Kernighan and Ritchie (or other reputable C books) for the rules of programming in C
- *AT&T C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup (ISBN 0-201-51459-1)
- *The C++ Programming Language* by Bjarne Stroustrup (published by Addison Wesley, Inc., ISBN 0-201-12078-X)
- *XView Programming Manual*, *XLib Reference Manual*, *XLib Programming Manual*, and *X Protocol Reference Manual* (published by O'Reilly and Associates, Inc.) for information about programming using the OpenWindows XView toolkit

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Examples
`Courier`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. .
`Courier Bold`	What you type, contrasted with on-screen computer output	% **`su`** password:
Palatino Italic	Command-line placeholder: replace with a real name or value	To delete a file, type the following: `rm` *filename*.
	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
%	UNIX C shell prompt	% *or* system%
$	UNIX Bourne shell prompt	$ *or* system$
#	Superuser prompt, either shell	# *or* system#

Introduction 1

This chapter provides an overview of Guide to the OPEN LOOK Intrinsics Toolkit (Golit).

Devguide and Golit

Devguide is a development tool that enables you to create and test user interfaces without writing any code.

Devguide produces GIL (Guide Interface Language) files that you can convert into interface toolkit code with a code generator.

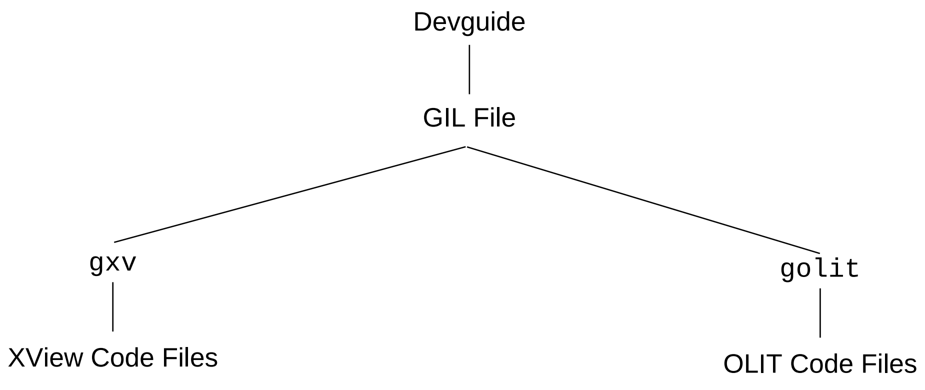


Figure 1-1 Devguide Tools and Files

There are two code generators described in the current manual set (see Figure 1-1). This manual describes the Golit code generator for the OLIT toolkit.

Names and Terminology

The word *Guide* in the full name, *OpenWindows Developer's Guide*, is an acronym for Graphical User Interface Design Editor. The abbreviated name, *Devguide*, refers to the graphical interface tool you use to develop the user interface.

The term *Golit* refers to the code generator that takes the GIL file produced by Devguide and produces the executable OLIT code for your user interface. This code is referred to as the *UI* (user interface) code, which contrasts with the *application code.*

Installing Devguide and Golit

To install Devguide and the code generators, follow the instructions in the *Software Developer Kit Installation Guide.*

Before you use Devguide and Golit, make sure the environment variables `GUIDEHOME` and `OPENWINHOME` point to Devguide's and OpenWindows' home directories respectively. These environment variables must be set correctly for Devguide, Golit, and the generated Makefile to work.

How you interact with Devguide and Golit

The basic interaction with Devguide is very simple. You use Devguide to develop your user interface with the OPEN LOOK look and feel by dragging and positioning glyphs on interface windows. Devguide saves a description of the interface in a GIL file. This process is explained in detail in the *OpenWindows Developer's Guide: User's Guide.*

Once you have built a working prototype of your user interface and generated the GIL file describing your prototype, you run the Golit program on the GIL file, shown below as `<fn>.G`. The `.G` suffix is the standard suffix for GIL files. Golit produces the three files shown in Figure 1-2.

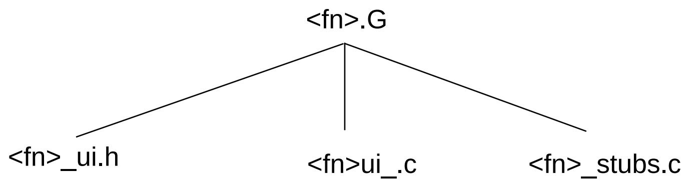


Figure 1-2 GolitGenerated Source Files

Golit also automatically generates a program makefile.

Devguide and Golit allow you to organize several GIL files into projects. Devguide saves projects in files that have a `.P` extension. When you run Golit on a project file, it generates the files shown in Figure 1-2 for each `.G` file in the project. It also generates `<projectname>.c` and `<projectname>.h` files.

How Golit and libgolit Work

Golit-generated code uses the OLIT toolkit and a runtime library (called libgolit.a) to create user interfaces. Golit-generated code is different from OLIT user interface code that you might write from scratch. It does not directly call widget creation routines. Instead, it initializes data structures to describe the user interface and form a *widget tree*. It then calls libgolit routines to traverse this tree and perform the actual widget creation.

Golit's design isolates code that describes interface objects from code that instantiates and manipulates the objects. Golit generates separate files for each type of code. This makes it easy to modify an application. It's easy to change the application code, since it's all in one place; you don't have to wade through detailed UI code to find it. It's also easy to change an interface and regenerate UI code, since you don't have to worry about overwriting application code. Golit only overwrites the files that contain the interface description.

Golits default widget creation does not allow you to change widget resources when a widget is created. However, Golit provides a way around this. You can use a special type of callback, called a *Create Proc*, to intercept the default widget creation and provide your own widget creation code. The Create Proc is invoked whenever the widget is about to be created. It can change resources at creation time as needed.

The libgolit library provides utility functions that enable you to selectively instantiate the widget tree. You can instantiate an individual widget or entire portions of that widget tree once, or many times. This allows you to reuse parts of the interface description that you created in Devguide, making your code more compact and efficient.

Getting Started with Golit 2

This chapter describes how to generate simple user interfaces and integrate them with your applications. It provides a very simple example of an application that uses Golit. This chapter is intended to be a “quick start guide” that provides you with the minimal instructions necessary to start using Golit. The topics discussed here are covered in greater detail in Chapter 3, “Tutorial” and Chapter 4, “Golit Functionality in Detail.”

Summary of How to Create an Application with Golit

To create an application with Devguide and Golit, follow these steps:

1. **Design an interface with Devguide and save it in a GIL file.**
   You can save an interface in several GIL files and organize these files into a project (`.P`) file. See the *OpenWindows Developer’s Guide: User’s Guide* for more information on designing interfaces.

2. **Generate UI code by running Golit on the GIL or project file.**
   Golit generates three C files and a makefile for each GIL file. The C files contain the interface source code. The makefile contains the Make utility commands to build the interface. If you run Golit on a project file, it generates two additional C files and a supplement to the makefile.

3. **Use `make` to build a test copy of the interface.**
   Golit generates all the code necessary to create an executable, so you can compile the code by itself immediately after you generate it. It is recommended that you test the interface before you add your application code.

4. **Insert your application code in the callback function templates generated by Golit in the `_stubs.c` file.**
   Your application code can change the resources of widgets in the interface or it can instantiate additional widgets.

5. **Run `make` again to build the application.**
   The Make utility compiles the interface and your application code into a complete executable.

Each of these steps is discussed in greater detail in the sections below.

Designing an Interface in Devguide

When you design an interface in Devguide, you must keep Golit in mind, since Golit's requirements are different than those of the other Devguide code generators.

Note – Golit does not support some of the user interface features available in Devguide. A list of these features is provided in Appendix B, "Unsupported Devguide Features." Also note that, because of differences between toolikits, some objects and labels won't appear exactly the same in your compiled application and Devguide.

Configuring Devguide to Generate GIL files for Golit

To create an interface that Golit can generate code for, you must make sure that Devguide is set up for Golit. Do the following before you start designing an interface:

1. **In Devguide, choose "Devguide ..." from the Properties menu.**
   The Devguide Defaults popup window appears (see Figure 2-1).

2. **Set the Toolkit setting to OLIT and click on the Apply button.**
   If the toolkit setting is already set to OLIT, don't change anything.

3. Dismiss the Devguide Defaults popup window.

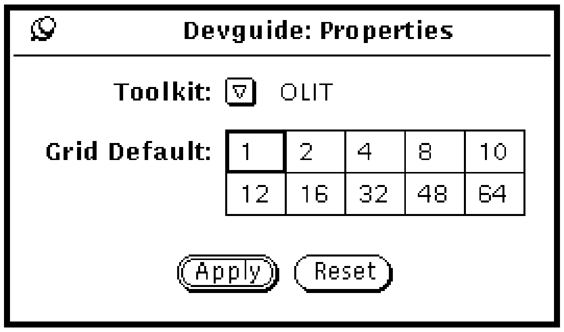


Figure 2-1 The Devguide Defaults Popup Window

Using Connections to Handle Events

To make your interface handle events, you specify *connections* in Devguide. When an event occurs on one object (called the *Source*) a connection triggers a specified action on a second object (called the *Target*). For example, you might want to specify that when SELECT is clicked (the Event) on a button (the Source), a popup (the Target) appears (the Action).

To set up a connection, you use the Connections Manager window. See the *OpenWindows Developer's Guide: User's Guide* for instructions on using this window.

Devguide and Golit provide a variety of predefined actions. "Action Menu Items" on page 49 furnishes a complete list of these actions. For each connection you set up in Devguide, Golit generates a callback function in the `_stubs.c` file. The function includes all the code necessary to execute the action you specified for the connection. It also includes a line of code that prints the connection name to the shell tool that you started Devguide from. You can supplement or modify this code with your own application code.

To set up a callback that consists entirely of your own code, create a connection in Devguide and specify CallFunction as the Action. Then enter a function name. Golit will generate a callback under the specified name. This callback contains only a `printf()` statement, which you can replace with your code.

If you want to provide some of your own callback code while you are in Devguide, choose ExecuteCode for the action when you create a connection. Devguide will display a popup which you can add your code into. Golit includes this code in the generated callback.

Note – If you are using a project file and more than one of its GIL files invokes the same CallFunction callback, the callback appears in the `<project_name>.c` file.

Generating User Interface Code from GIL Files

There are two ways to generate code from a GIL file: from the command line and with Devguide's code generator tool.

Generating Code from the Command Line

To generate user interface code from the command line, you enter the following command:

```
% golit [<GIL_filename>]
```

If you are generating code for a project, run Golit on the project file by entering the following command:

```
% golit -p [<project_filename>]
```

It is not necessary to type the `.G` or the `.P` filename extension after the GIL or project file filenames, since Golit adds the appropriate extension automatically.

Generating Code with the Code Generator Tool

Devguide provides a code generator tool which enables you to generate, compile, and run your applications within Devguide. The code generator features a menu-driven interface, so you don't have to type anything at the command line.

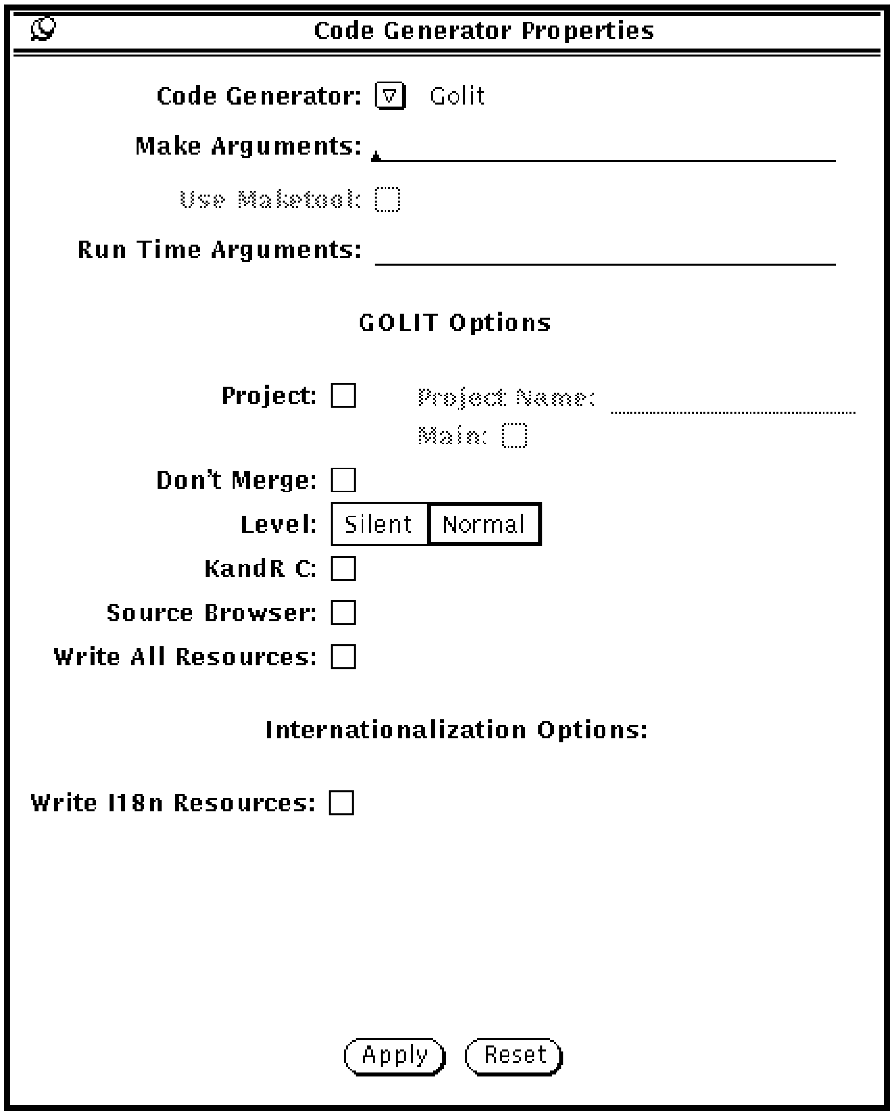
Setting Up The Code Generator Tool For Golit

To use the code generator tool, you must configure it for Golit. To do this, follow these steps:

1. **Click SELECT on the Properties... button.**
   The Code Generator Properties pop-up window appears (see Figure 2-2).

2. **Use the Code Generator abbreviated menu button to select Golit.**
   The options for Golit appear.

Note – Be sure to use only one code generator on each GIL file. If you use two different code generators, it will corrupt the generated files. For example, if you use GXV to generate code from a GIL file and subsequently use Golit on the same file, the generated files will be corrupted.



Code Generator Properties

Code Generator: ▽ Golit

Make Arguments:

Use Maketool: ☐

Run Time Arguments:

GOLIT Options

Project: ☐ Project Name:

Main: ☐

Don't Merge: ☐

Level: Silent [✓ Normal]

KandR C: ☐

Source Browser: ☐

Write All Resources: ☐

Internationalization Options:

Write I18n Resources: ☐

Figure 2-2 Code Generator Properties Window

Choosing Golit Options for Code Generator Tool

The code generator tool provides a variety of options for generating Golit code. In addition to the command line options, which are described on page 51, you can also select the following options in the Code Generator Properties pop-up window.

- **Make Arguments**
  You can enter `make` arguments, such as `clear` in this field. The `Makefile` contains all the property options that you used the last time you generated code. If you load in an application with a `Makefile`, all of the code generator options contained in the `Makefile` are loaded into the tool. When you click SELECT on the Apply button on the property sheet, the `Makefile` is updated.
- **Maketool**
  Check Use Maketool if you want to use the SPARCworks™ or ProWorks Maketool. Type in make or runtime arguments in the appropriate text fields. The Maketool option works only if you have installed the SPARCworks or ProWorks software and updated your path accordingly.
- **Runtime Arguments**
  You can specify any runtime arguments, such as `-scale large` in this field.

For more information on code generator tool, see the *OpenWindows Developer's Guide: User's Guide*.

Golit Generated Files

Golitgenerated files contain all the source code necessary to create an executable that uses the OLIT toolkit and the libgolit runtime library.

When you run Golit on a single GIL file, it generates the following files in the current working directory:

- `<GIL_filename>_stubs.c` - skeleton main program for the executable. It includes the callback templates that you insert your application code into.
- `<GIL_filename>_ui.c` - code that defines the user interface objects. Unless you are writing advanced applications, you don't need to look at this file.

- `<GIL_filename>_ui.h` - header file that declares the user interface objects, external callbacks, and external creation procedures. Normally, you won't need to look at this file.
- `Makefile` - a template makefile to build the executable. `Makefile` contains `make` utility commands to compile your interface and any application code you include in the `_stubs.c` file

The filenames of these files are based on the name of the GIL file. For example, if you use Golit to generate source code for a GIL file named `display.G`, Golit creates the files, `display_ui.c`, `display_ui.h`, and `display_stubs.c`.

When you run Golit on a project file, it generates the files listed above (except for the makefile) for each GIL file in the project. It also generates the following files:

- `<project_name>.c` - includes the main program which is left out of the individual `_stubs.c` files. It also includes templates for callbacks shared by the GIL files in the project.
- `<project_filename>.h` - external declarations for callbacks shared by the GIL files.
- `<project_filename>.make` - addition to the makefile that is automatically included.

The filenames of these files are based on the name of the project file. For example, if you use Golit to generate source code for a project file named `myproject.P`, Golit creates the files `myproject.c`, `myproject.h`, and `myproject.make`.

Compiling and Testing Interface Code

To compile the interface code generated by Golit, use the code generator tool or type the following after you have generated the code:

```
% make
```

The `make` utility compiles and links the Golit-generated code. The resulting executable has the same name as the GIL file, without the `.G` filename extension. If you have generated code for a project, it will have the same name as the project file without the `.P` extension.

Note – Devguide provides a test mode that allows you to test an interface while you are designing it. However, some connections that specify events or actions specific to the Golit code generator do not work in the Devguide test mode. Also, some objects and labels appear differently in the test mode than they do in the compiled applicaton. Always test your interface by compiling the generated interface code once and running it before you add your application code.

The executable you create when you compile the generated code provides a complete working model of the user interface. When you perform an action in the interface (for example, a button press) the program prints the name of any connections associated with that action to the console.

Integrating Application Code with Golit Interface Code

To add your application code to the Golit interface code, you modify the callback templates in the `_stubs.c` file. If you call functions that you keep in separate files, you also need to modify the `Makefile` to compile these files.

Caution – Do not alter the `_ui.c` or `_ui.h` files. If you add any code to these files, you will lose it the next time you run Golit.

Setting Widget Resources

To set a widget's resources in your application code, you must retrieve its id. The libgolit library provides a function, `GolitNameToWidget()`, that does this for you. You use it as follows:

```
Widget w;

w = GolitNametoWidget(root, “Ui_object_name”);
```

where:

`Ui_object_name`	is the name of the widget for which you want to get an id. Golit constructs widget names by prepending the name of the interface to the name you gave to the

	object in Devguide. For example, if you create a button called `button1` in Devguide and save it in a GIL file called `mygilfile`, its widget name will be `Mygilfile_button1`. Note that Golit automatically capitalizes the first character of the name.
`root`	is the widget in the widget tree where you would like to begin searching for the widget designated by `Ui_object_name`. Normally, you can just use the widget that is passed to the callback.

Regenerating Code for a Modified Interface

Golitand Devguide make it easy to change an interface, even after you have integrated your application code with the interface code. To change an interface, you load the interface's GIL file in Devguide, alter the interface, and save the GIL file. You then run Golit on the altered GIL file to regenerate the code.

Each time you run it, Golit does the following:

1. Overwrites the existing `_ui.c` and `_ui.h` files.
2. Backs up the current `_stubs.c file` to `_stubs.c.BAK`.
3. Generates a new `_stubs.c` file that contains templates for all the connections specified in the GIL file.
4. Merges any code that you inserted in the original `_stubs.c` into the new `_stubs.c`.
5. Creates a `_stubs.c.delta` file that tells you how it has changed `_stubs.c`. This file lists the added text and the affected line numbers.

If you are regenerating code for a project, Golit performs the steps above for each of the GIL files in the project. It also does the following:

1. Backs up the `<projectname>.c` and `<project_name>.h` files to `.BAK` files.
2. Generates a new `<projectname>.c` file and merges it with the old one, listing the changes in a `.delta` file.
3. Overwrites the original `<project_name>.h` file.

A Simple Example

Let's say that you want to create an application that displays a button and a gauge. Each time the user presses the button, you want the gauge value to increase by one. To create this application, you follow these steps:

1. **Create the interface in Devguide.**
   To do this, drag a Base Window glyph onto the workspace. Then place a control area in it and resize it to fit. Drag a button and a gauge glyph onto the control area. For this example, use the default names and values provided by Devguide for the objects. The interface should appear like the one displayed in Figure 2-3.

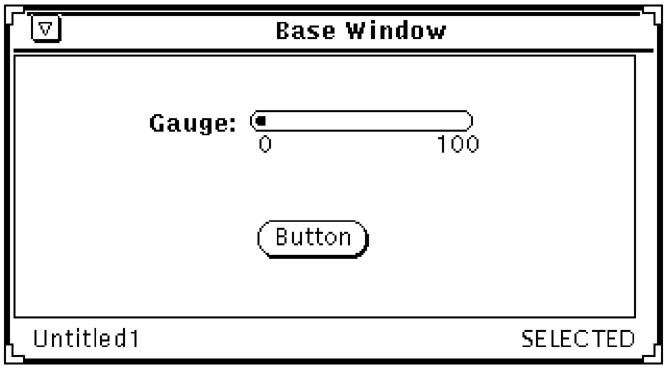


Figure 2-3 Example Interface as it Appears in Devguide

2. **Create a connection for the button in the Connections Manager.**
   Make sure that the button(button1) is the Source. Select CallFunction for the Action and specify `incr_gauge` as the function name. Since you are defining the Action, it does not matter what the Target is. After you have created the connection, the Connections Manager window should look something like the one shown in Figure 2-4.

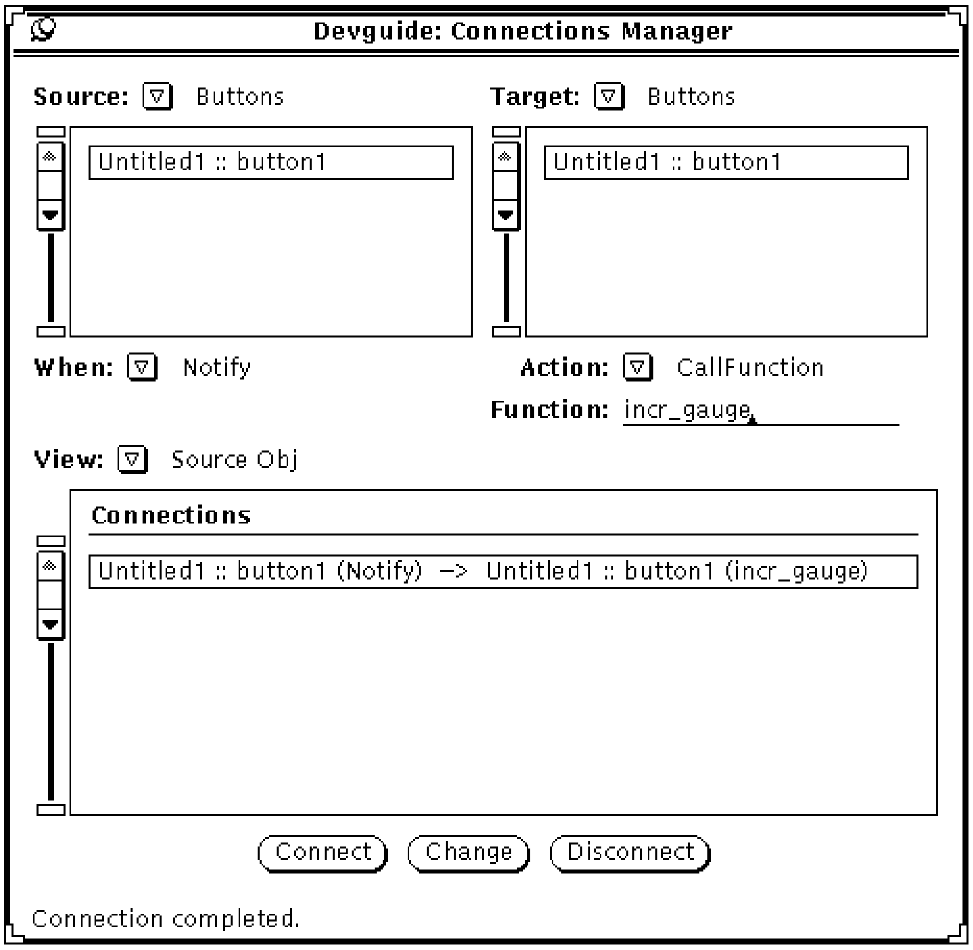


Figure 2-4 Connections Manager Window

3. **Save the interface to a GIL file.**
   For this example, save it to `ex1`.

4. **Type** `golit ex1` **to generate the code.**
   Golit generates the interface code and the `Makefile`.

5. **Type** `make` **to build a test copy of the interface.**
   Make builds an executable named `ex1`.

6. **Type** `ex1` **to run the executable.**
   Each time you press the button displayed in the base window, the application prints the connection name to the console.

7. **Modify the** `ex1_stubs.c` **file.**
   The file generated by Golit should look like the one in Figure 2-7. You will only need to change the `incr_gauge()` function template. Add code that uses `GolitNameToWidget()` to retrieve the gauge widget id. Then use Intrinsics functions to increment the gauge. When you are done, the `incr_gauge()` callback should look something like the code in Figure 2-7.

8. **Type** `make` **to build the complete application.**
   Golit builds an executable called `ex1`.

```
/*
 * ex1_stubs.c - Main routine, callbacks, and connections stubs.
 * This file was generated by ‘golit’ from ‘ex1.G’.
 */

#include “ex1_ui.h”


String fallback_resources [] = {
          “*background: gray”,
          NULL
};
```

Figure 2-5 Generated stubs.c File for ex1

```
/*
 * main for application ex1
 */
int
main(argc, argv)
      int       argc;
      char      **argv;
{
      Widget widget1;
      Display *dpy;
      XtAppContext app;

      /* Initialize Toolkit */
      OlToolkitInitialize(NULL);
      XtToolkitInitialize();

      / * Create Application Context and set fallback resources
      app = XtCreateApplicationContext();
      XtAppSetFallbackResources(app, fallback_resources);

      /* Open Display */
      dpy = XtOpenDisplay(app, NULL, “ex1”, “Ex1”,
                (XrmOptionDescList) NULL, 0, &argc, argv);
      widget1 = GolitFetchShellHier(“Ex1_window1”, “Ex1”,
                Ex1_window1, dpy, NULL, NULL, NULL);
      XtRealizeWidget(widget1);

       /* Enter event loop */
      XtAppMainLoop(app);
      exit(0);
}
void
incr_gauge(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
      printf(“connection: incr_gauge\n”);
}
```

Figure 2-6 Generated _stubs.c File for ex1 (main)

```
void
incr_gauge(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
Arg arg[1];
static int gauge_value;

/* Get the gauge’s widget id */
      Widget target = GolitNameToWidget(widget, “Ex1_gauge1”);

/* Increment gauge unless limit has been reached */
      if (gauge_value < 100)
      {
      XtSetArg(arg[0], XtNsliderValue, ++gauge_value);
      XtSetValues(target, arg, 1);
      }
}
```

Figure 2-7 Callback Modified to Increment a Gauge Each Time a Button is Pressed

Tutorial 3

This chapter demonstrates how to create a complete application with Devguide and Golit. In particular, it shows how to create projects and layered panes and how to use connections to handle events. It also shows how to modify generated callback functions, compile code, and run the completed application.

The sample application in this chapter consists of a text pane into which you can enter text and controls that set the appearance of the text. More specifically, it consists of a base window with:

- A text pane
- A control area with two buttons:
  - A menu button that switches between the two layered control areas
  - A button that displays a pop-up window
- Two layered control areas:
  - A control area with an exclusive setting that sets the font to Lucida or Rockwell
  - A control area with a checkbox that sets the font style to bold, italic, or both

The application provides a pop-up window, with:

- A control area with a slider that sets the text size.

The interface is shown in Figure 3-1 on page 22. Note that the control area that provides the font style checkbox is not visible.

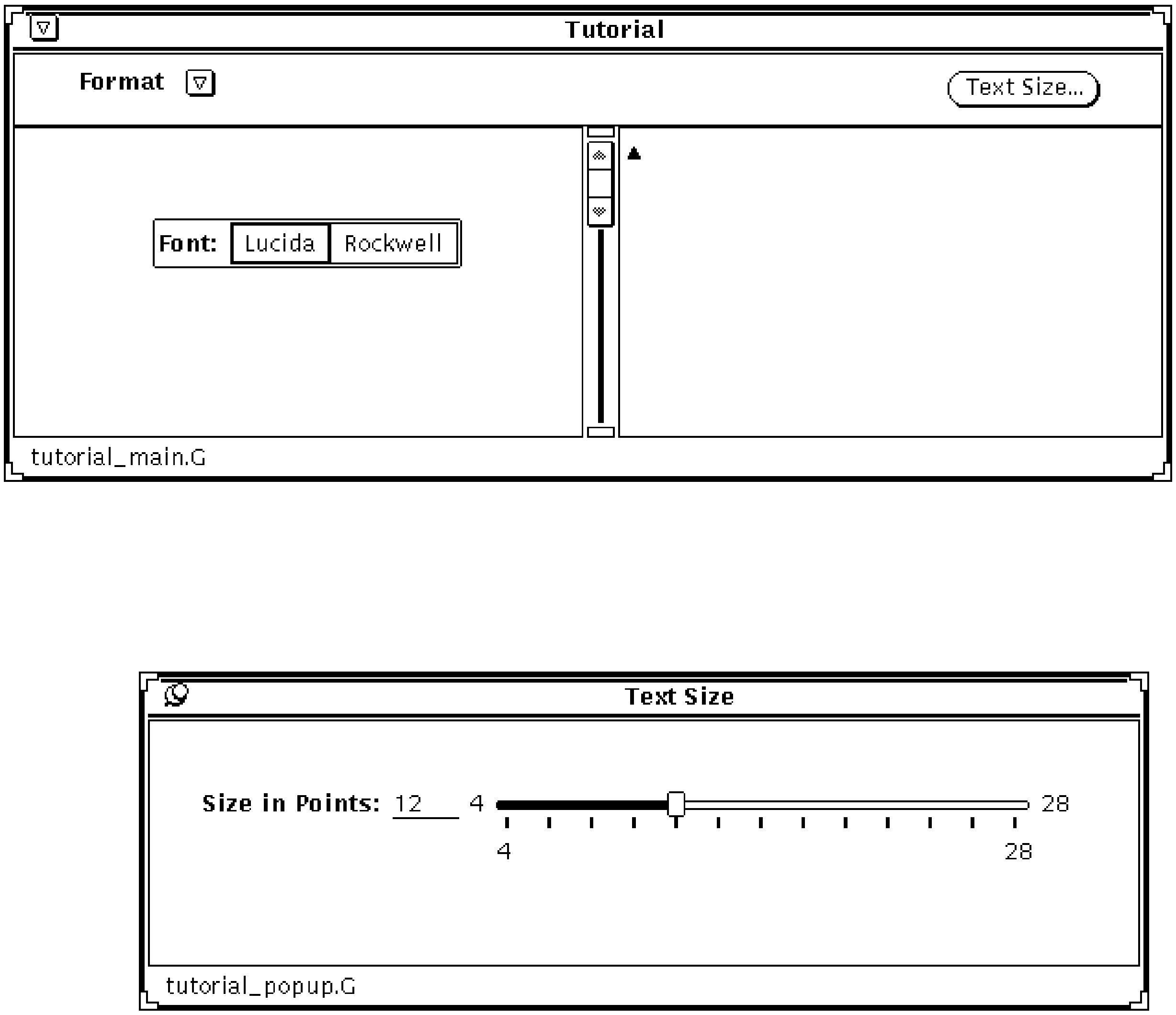


Figure 3-1 The Tutorial Application

Note – The example in this chapter occasionally diverges from the OPEN LOOK specification as it is outlined in the *OPEN LOOK Graphical UI Style Guidelines*. This is necessary to demonstrate a wide variety of Devguide features in a simple interface. When you develop your own applications, you should attempt to adhere to the specification.

Building an Interface

To create an interface with Devguide, you drag glyphs from Devguide's palette onto the workspace. Each glyph turns into an interface element. To customize an element's properties, open its property window, set property values, and click SELECT on the Apply button.

In this example, you will create a project with two interface (`.G`) files: one for the base window and one for the pop-up window. Note that the term, *interface*, refers to both the portion of your application stored in each `.G` file and to the entire application.

Creating a Directory for GIL and Project Files

When you create an application with Devguide, store all GIL (`.G`) and project (`.P`) files in one directory. Start Devguide and run Golit in that directory.

1. **Create a new subdirectory in your home directory and name it `tutorial`.**

```
% mkdir tutorial
```

2. **Change directories from your home directory to `tutorial`.**

```
% cd tutorial
```

3. **Start Devguide.**

```
% devguide &
```

Creating the Base Window Interface

All applications have one main base window which is visible (or iconic) when you first run an application. Figure 3-2 shows the completed base window interface for the sample application as it appears in the Browser.

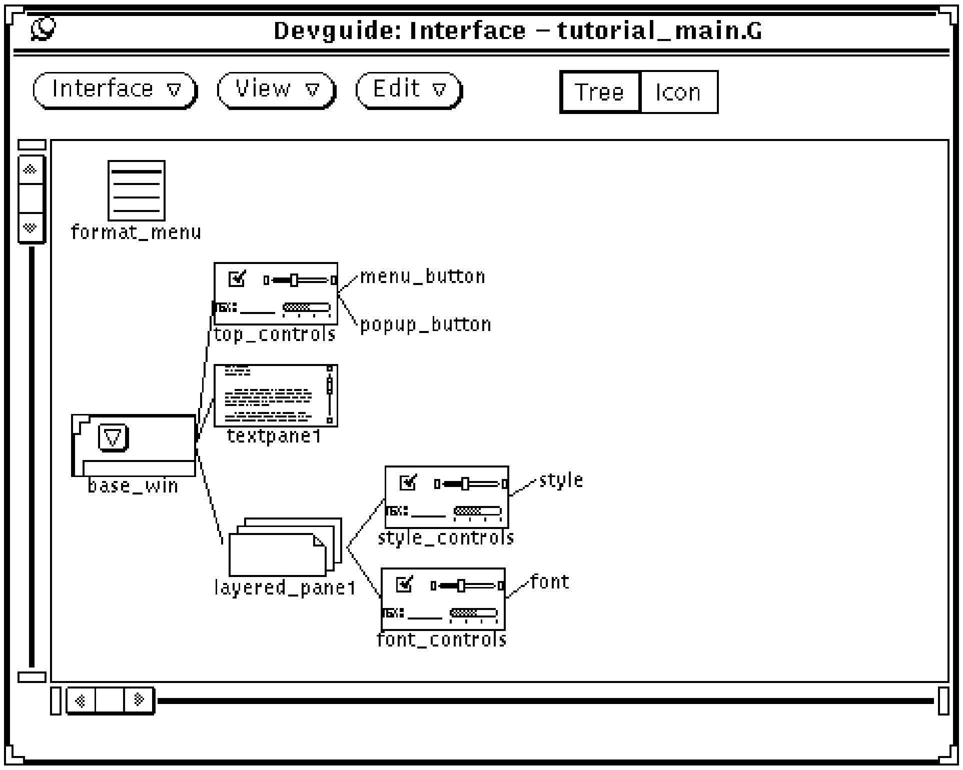


Figure 3-2 The Completed Base Window Interface Displayed in the Browser

Creating the Base Window

The first step in building an interface is to create a base window on the workspace. In this section you create a base window with layered control areas, a text pane, and various controls. You then save it in a GIL file.

1. **Drag the Base Window glyph from Devguide's palette onto the workspace and resize the base window.**
   Resize the base window so it is about three inches high and seven inches long.

2. **Customize the base window's properties.**
   Double-click SELECT on the base window's interior or footer to open the property window. Then, change the values in the fields:

Object	Property Field	Value
Base Window	Object Name	`base_win`
Base Window	Label	`Tutorial`

Note that the changes you make on the Properties window do not take effect until you click SELECT on the Apply button.

Adding a Control Area to the Base Window

You can place panes and control areas directly on a base or pop-up window. However, you *must* place control elements, such as buttons and sliders on a control area. A window can have more than one control area.

1. **Create a control area by dragging the Control Area glyph onto the top of the base window.**
   Place and resize the control area so it is about half an inch high and extends all the way across the top of the base window. This control area will contain the application's Format and Text Size buttons (see Figure 3-3).

2. **Customize the control area's properties as follows:**

Object	Property Field	Value
Control area	Object Name	`top_controls`

Adding Buttons to the top_controls Control Area

You create two different kinds of buttons for this application: an abbreviated menu button and a button to display a pop-up window.

1. **Place buttons in the top control area.**
   Drag the Button glyph and drop it at the left end of the control area. This button becomes the Format menu button. Later on, you will create a menu and attach it to this button.

   Drag the Button glyph again and drop it at the right end of the control area. This button becomes the Text Size... button, which displays the pop-up window.

2. **Customize the buttons' properties.**

Object	**Property Field**	**Value**
Left button	Object Name	`menu_button`
Left button	Label	`Format`
Left button	Type	Abbreviated Menu
Right button	Object Name	`popup_button`
Right button	Label	`Text Size...`

Figure 3-3 shows how the interface should appear when you have completed these steps.

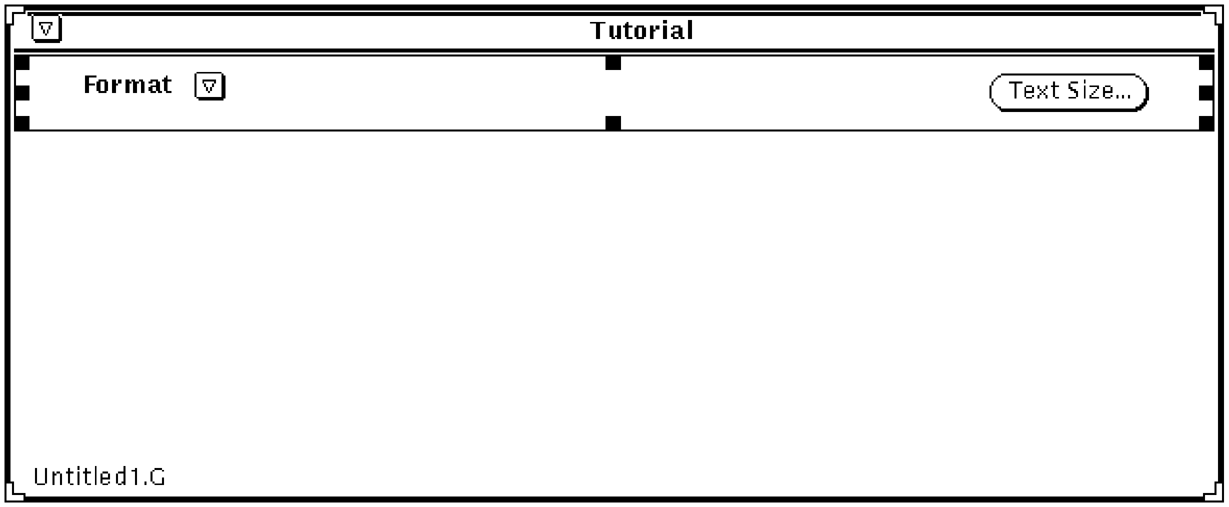


Figure 3-3 Example Interface with top_controls and Buttons

Adding a Text Pane to the Tutorial Base Window

To add a text pane to the base window, drag the Text Pane glyph onto the right side of the base window and resize it (see Figure 3-4). Use the default name, `textpane1`, for the text pane's Object Name.

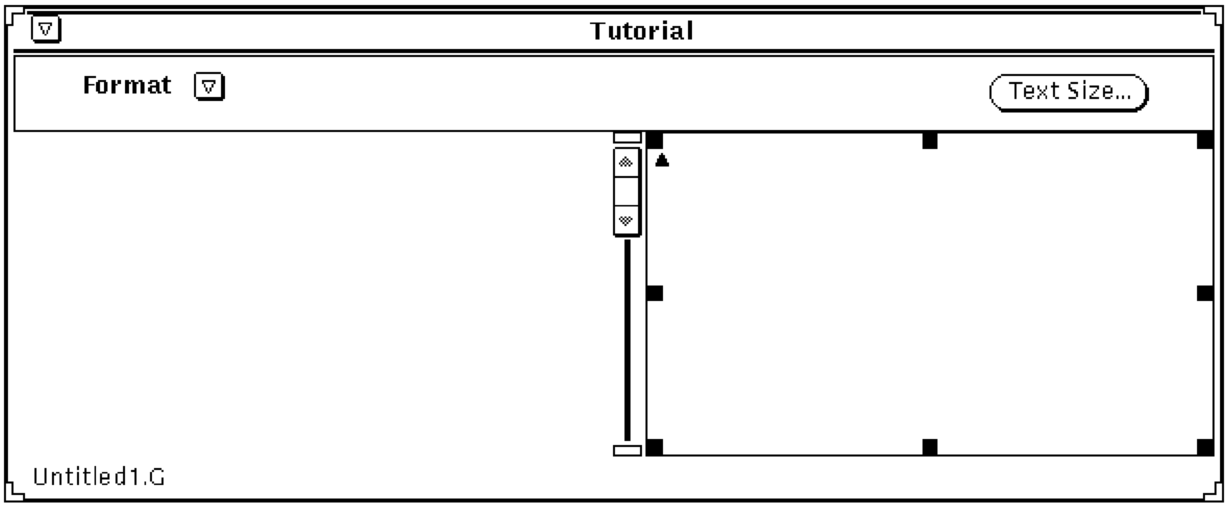


Figure 3-4 Example Interface with Text Pane

Creating Layered Control Areas

Devguide allows you to add layered panes to your application. You can layer control areas, text panes, or any other panes. In the completed application, only one pane is visible at a time. The application must include connections that make the desired pane visible and the other pane invisible. In this application, you will create connections to the menu, `format_menu`, to switch between the two control areas.

To create layered control areas:

1. **Drag the Control Area glyph onto the lower left control area of your base window.**
   Resize the control area to look like the one in Figure 3-5 on page 29.

2. **Drag the Control Area glyph onto the control area you just created**
   The new control area assumes the position and size of the original control area underneath it.

3. Customize the control areas' properties:

Object	Property Field	Value
controls1	Object Name	`style_controls`
controls1	Initial State	Invisible
controls2	Object Name	`font_controls`

4.

The font_controls control area will be visible first when you run the completed application.

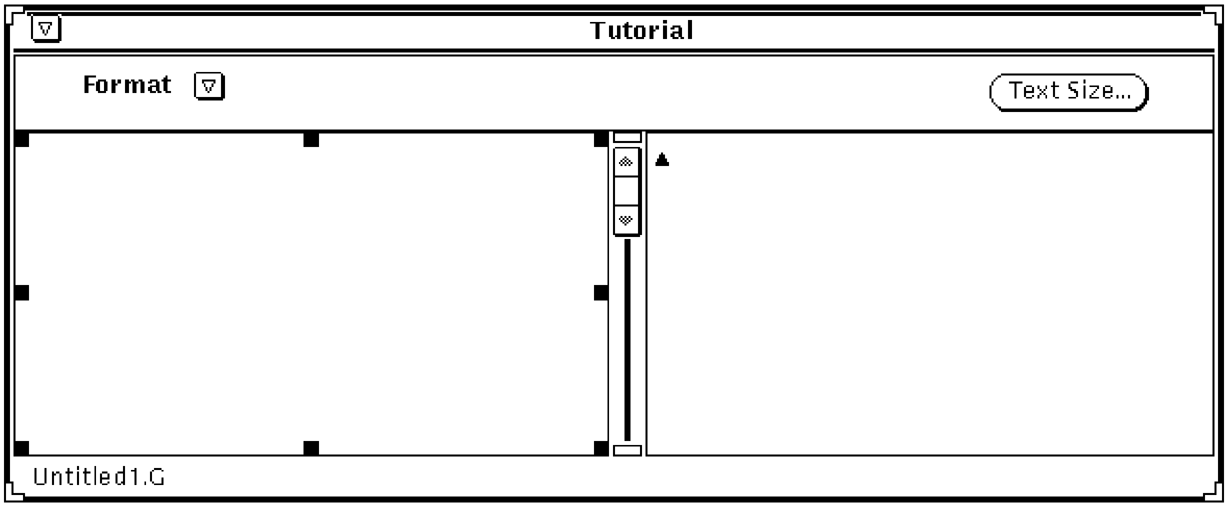


Figure 3-5 Example Interface with Layered Control Areas

Adding Controls to the Layered Control Areas

In this section, you will add controls that set the text font and style to the layered control areas. You will add an exclusive setting to font_controls and a checkbox setting to style_controls.

To add an exclusive setting to the font_controls control area:

1. **Drag the Exclusive Setting glyph onto the font_controls control area.**
   Settings can contain any number of choices; you will provide two, Rockwell and Lucida.

2. **Customize the setting's properties as follows:**

Object	Property Field	Value
Setting	Object Name	`font`
Setting	Label	`Font:`

3. **Customize the setting items' (Choices) properties.**
   The Choices scrolling list contains two choices. Set them as follows:

Object	Property Field	Value
Top Choice	Label	`Lucida`
Bottom Choice	Label	`Rockwell`

To add a checkbox setting to the style_controls area:

1. **Make the style_controls control area visible.**
   Select the font_controls control area. Choose Next Layer from the View menu. The style_controls control area will come to the front.

2. **Drag a Checkbox Setting glyph onto the control area.**
   Customize the checkbox setting's properties as follows:

Object	Property Field	Value
Checkbox Setting	Object Name	`style`
Checkbox Setting	Label	`Style:`

3. **Customize the individual checkboxes' (Choices) properties.**
   The Choices scrolling list contains two choices. Set them as follows:

Object	Property Field	Value
First Choice	Label	`Bold`
Second Choice	Label	`Italic`

Figure 3-6 shows the interface with style_controls in front; Figure 3-2 on page 24 shows it with font_controls in front.

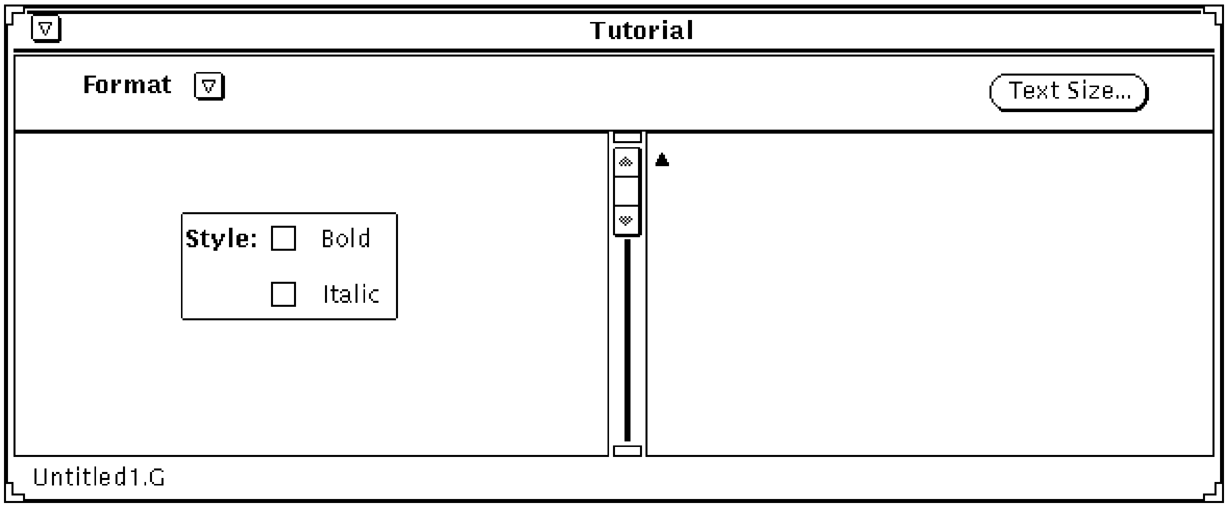


Figure 3-6 Example Interface with Style Controls Displayed

Saving the Tutorial Base Window Interface

To save the base window interface:

1. **Click MENU on Devguide's File button.**
2. **Choose Save As from the File menu to display the Save As file chooser.**
3. **Enter** `tutorial_main` **in the file chooser's Name field.**
4. **Click SELECT on the Save button**

Creating a Menu

You can attach menus to buttons, canvas panes, control areas, and scrolling lists. In this section, you create a menu and insert menu items using the Menu Editor window. In a later section, you will create a connection to make the menu appear when the Format menu button is pressed.

1. **Choose Menus... from Devguide's Properties menu.**
   This displays the Menu Editor window.

2. **Create a new menu.**
   Click SELECT on the Create button. This creates a new menu in the current interface, `tutorial_main.G`. Set the menu's properties as follows:

Object	**Property Field**	**Value**
Menu	Object Name	`format_menu`
Menu	Title	`Format`
Menu	Type	Exclusive

3. **Insert items into the exclusive menu.**
   Click SELECT on the Insert button to create a new menu item. Create two items with the following properties:

Object	**Property Field**	**Value**
First Menu Item	Label	`Font`
Second Menu Item	Label	`Style`

4. **Save the base window interface again.**
   Choose Save from Devguide's File menu. The menu you created is saved as part of `tutorial_main.G`.

Creating the Pop-up Window Interface

In this section, you create an additional interface and add a pop-up window, a control area, and a slider to it. Figure 3-7 shows the complete interface as it appears in the Browser.

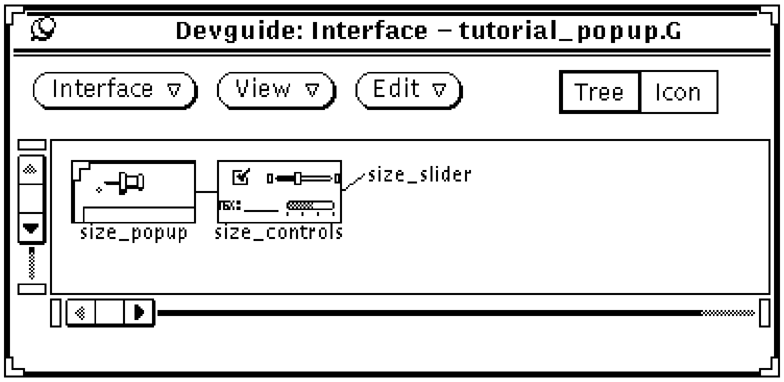


Figure 3-7 The Pop-up Window Interface Displayed in the Browser

Creating the Pop-up Window

1. **Choose New Interface... from Devguide's File menu.**
   The Interface Browser is displayed on the workspace. Note that anything you drag onto the workspace is part of the new interface until you create another interface or modify a previously made interface. Icons for new interface elements appear in the Browser's window.

2. **Drag the Pop-up Window glyph onto the workspace and resize the pop-up window.**
   Use the resize corners to stretch the window so that it is about two inches high and six inches long.

   Customize the pop-up window's properties as follows:

Object	**Property Field**	**Value**
Pop-up window	Object Name	`size_popup`
Pop-up window	Label	`Text Size`
Pop-up window	Window Parent	`base_win`

Adding a Control Area to the Pop-up Window

1. **Drag the Control Area glyph onto the pop-up window and resize the control area.**
   Resize the control area to fill the pop-up window.

2. **Customize the control area's properties as follows:**

Object	Property Field	Value
control area	Object Name	`size_controls`

Adding a Slider to the Control Area

To enable the user to set the text font size, add a slider to the control area. Modify the slider's properties so that the user can specify any font size between 4 and 28.

1. **Drag the Slider glyph onto the pop-up window's control area.**

2. **Customize the slider's properties as follows:**

Object	Property Field	Value
slider	Object Name	`size_slider`
slider	Label	`Size in Points:`
slider	Width	`240`
slider	Range Min	`4`
slider	Range Max	`28`
slider	Tick String Min	`4`
slider	Tick String Max	`28`
slider	Ticks	`13`
slider	Initial Value	`14`

Saving the Pop-up Window Interface

To save the pop-up window interface:

1. **Click MENU on Devguide's File button.**
2. **Choose Save As from the File menu to display the Save As file chooser.**
3. **Enter `tutorial_popup` in the file chooser's Name field.**
4. **Click SELECT on Save.**

Creating and Saving a Project

If you have more than one interface file in your application, as in this tutorial, create a project. To create a project consisting of the base and pop-up window interfaces:

1. **Choose Project... from Devguide's File menu to display the Project Organizer window.**
   The window displays a file icon for each GIL file currently loaded in Devguide. Icons should appear for `tutorial_main.G` and `tutorial_popup.G`.
2. **Choose Save As from the Project menu.**
3. **Enter `tutorial` in the Save As file chooser's Name field.**
4. **Click SELECT on Save.**
   This produces a project file named `tutorial.P`.

Setting up Connections

Devguide provides two kinds of connections that handle events in your application:

- Predefined connections, which generate complete callback code
- CallFunction connections, which generate callbacks that you insert custom code into

In this section you establish predefined connections to:

- Display the Format menu when the Format menu button is pressed

- Switch between the layered control areas when an item is chosen from the Format menu
- Display the Text Size pop-up window when popup_button button is pressed

You also establish CallFunction connections to:

- Change the text font size whenever the value of size_slider is changed
- Set the text font type and style whenever the controls in font_controls or style controls are changed

You must write the code to execute the CallFunction connections.

To set up connections, use the Connections Manager window. Choose the appropriate Source and Target menu items and select the Source and Target you want from the scrolling lists. Choose the When and Action menu items for the connection. In some cases, you must enter an argument for the Action. Click SELECT on the Connect button to establish the connection.

You can open the Connections Manager window from Devguide's Properties menu, from an object's property window (using the Connections... button), or by using drag-and-link.

Note – You can resize the Connections Manager window to display the full item names in the scrolling lists.

Creating a Connection to Display the Format Menu

To create the connection that makes the Format Menu appear when menu_button is pressed:

1. **Open the Menu Editor window.**
   Choose Menus... from the Properties Menu

2. **Click SELECT on the Connections... button in the top section of the Property sheet.**

3. **Choose Buttons from the Connections Manager Source menu and Menus from the Target menu.**
   Set up the following connection:

Source	**Target**	**When**	**Action**
menu_button	format_menu	Menu	Show

Be sure to click SELECT on the Connect button to create the connection.

Creating Connections to Switch between Layered Control Areas

To switch between the layered control areas, you need to create four separate connections to do the following:

- Show style_controls when Style is chosen
- Hide font_controls when Style is chosen
- Show font_controls when Font is chosen
- Hide style_controls when Font is chosen

To create the connections (in the Connection Manager), choose Menu items from the Source menu and Control Areas from the Target menu. The item names that appear in the scrolling lists are prefixed with the menu name (for example, format_menu.Font). Note that you may have to resize the Connections Manager Window so that you can see the entire names.

Create the following connections:

Source	**Target**	**When**	**Action**
Font	font_controls	Notify	Show
Font	style_controls	Notify	Hide
Style	style_controls	Notify	Show
Style	font_controls	Notify	Hide

Creating a Connection to Display the Text Size Pop-up Window

You can use drag and link to create a connection that displays the pop-up window when you click SELECT on the Text Size menu button:

1. **Click SELECT on the Text Size button.**

2. **Press the Meta key and drag the pointer from the button to the pop-up window's header.**
   The pointer turns into a cord with a plug at the end.

3. **Release the SELECT button.**
   The Connection window will appear if it is not already on the screen. Drag and link sets the Text Size button and pop-up as the default Source and Target objects. Check that these are set correctly.

Source	**Target**	**When**	**Action**
popup_button	size_popup	Notify	Show

4. **Click SELECT on the Connect button to create the connection.**

Creating a Connection to Set the Text Size with the Slider

To create a connection that enables the slider to set the text size, choose Sliders from the Source and Target menus and specify the following:

Source	**Target**	**When**	**Action**	**Arg**
size_slider	size_slider	Notify	CallFunction	`set_textsize`

Creating Connections to Set the Font and Style

These connections will all be CallFunction Connections. Note that you must provide two connections for each checkbox: one for when it is selected (to set the font to bold or italic) and one for when it is deselected (to set the font back to normal). To set up the connections, choose Setting Items from the Source and Target menus (in the Connections Manager). The item (checkbox) names displayed in the scrolling lists are prefixed with the setting name (for example, font.Lucida). Specify the connections as follows:

Source	**Target**	**When**	**Action**	**Arg**
Lucida	Lucida	Notify	CallFunction	`set_lucida`
Rockwell	Rockwell	Notify	CallFunction	`set_rockwell`
Bold	Bold	Notify	CallFunction	`set_bold`
Italic	Italic	Notify	CallFunction	`set_italic`
Bold	Bold	Unselect	CallFunction	`set_unbold`
Italic	Italic	Unselect	CallFunction	`set_unitalic`

Experimenting with Test Mode

You have created several predefined connections that work in Test mode. Try them out to see how parts of the tutorial user interface work. First enter Test mode. Then display the Format menu, scroll through the layered panes, and display the Text Size pop-up window.

1. **Enter Test mode by clicking SELECT on Test.**
   The Test choice is highlighted. All the predefined connections in your application will work. The only connections that will not work are CallFunction connections.

2. **Put the pointer over the text pane you created and enter some sample text.**

3. **Press MENU on the Format menu button.**
   This displays the Format menu.

4. **Choose Style from the Format menu.**
   You now see the checkbox setting that allows you to choose Bold, Italic, or both. When you click SELECT on a choice, that choice highlights. You cannot change the text style in Test mode, because the underlying connections are CallFunction connections. See "Setting up Connections" on page 35 for more information about connections.

5. **Choose Font from the Format menu.**
   You now see the Font exclusive setting. You cannot change the text font in Test mode.

6. **Click SELECT on the Text Size button to display the Text Size pop-up window.**
   This displays the slider you created. You cannot change the font size in Test mode.

7. **Click SELECT on Build to reenter Build mode.**

8. **Unload the previously-saved base and pop-up window interface files.**

Generating Interface Source Code

To generate the interface source code, first make sure you have saved all your changes. Then run the `golit` command on your project file. Golit lists the files as it reads and writes them:

```
% golit -p tutorial
golit: reading tutorial.P
golit: reading tutorial_main.G
golit: writing tutorial_main_ui.h
golit: writing tutorial_main_ui.c
golit: writing tutorial_main_stubs.c
golit: reading tutorial_popup.G
golit: writing tutorial_popup_ui.h
golit: writing tutorial_popup_ui.c
golit: writing tutorial_popup_stubs.c
golit: writing tutorial.c
golit: writing tutorial.h
golit: writing Makefile
```

Modifying the Golit-Generated Code

In this section you will alter the Golit-generated files to make the CallFunction connections work. The callback functions Golit generates for these connections contain print statements that you will replace with custom code. You will also create a function called `set_font()` which the `set_rockwell()`, `set_lucida()`, `set_italic()`, `set_bold()`, `set_unitalic()`, `set_unbold()`, and `set_textsize()` callbacks will use to change the text pane's font, style, and size.

Creating a Header File

To register the user's choice of Bold and/or Italic, and Rockwell or Lucida and font size, you must create a global structure. To do this, create a separate header file, `myheader.h`, that contains the following declaration:

```
struct characterformat {
          String font;
          Boolean bold;
          Boolean italic;
          String size;
} characterformat;
```

Add the following line below the other `#include` lines in the `tutorial.c`, `tutorial_main_stubs.c`, and `tutorial_popup_stubs.c` files:

```
#include “myheader.h”
```

Customizing the tutorial.c File

To initialize the `characterformat` structure, add the following code after the beginning of the `main()` program in `tutorial.c`:

```
characterformat.font = “lucidasans”;
characterformat.bold = FALSE;
characterformat.italic = FALSE;
characterformat.size = “14”;
```

This sets the `characterformat` structure to the normal default font settings.

The CallFunction callbacks that you set up in Devguide must reset the textpane font to reflect the user's choice of font style and size. It is easiest to do this by calling a common function that sets the font to the specifications in the `characterformat` structure. Define this function, `setfont()`, at the end of the `tutorial.c` file as follows:

```
/* Sets font to that specified by characterformat struct */
void
setfont(widget)
Widget widget;
{
    Arg arg;
    char fontname[64]; /* array to store font name */
    struct characterformat *cf = &characterformat;

    /* Get Widget id for the text pane */
    Widget target = GolitNameToWidget(widget,
        “Tutorial_main_textpane1”);

    /* Concatenate font descriptions into font name */
    sprintf (fontname, “%s%s%s%s-%s”,
        cf->font,
        ((cf->bold || cf->italic) ? “-” : ““),
        (cf->bold ? “bold” : “”),
        (cf->italic ? “italic” : “”),
        cf->size);

    /* Set text pane font resource to font name */
    XtVaSetValues(target, XtVaTypedArg, XtNfont, XtRString,
         fontname,strlen(fontname) + 1, NULL);
}
```

The `setfont()` function uses `GolitNameToWidget()` to retrieve the widget id of the textpane. The widget name (`Tutorial_main_textpane1`) that you pass to `GolitNameToWidget()` is constructed by joining the textpane object's name to the interface name.

`setfont()` creates a fontname by compiling the description in the `characterformat` structure. It starts with the selected font. If the `bold` or `italic` variables are TRUE, it adds `bold` or `italic` to the name. Finally, it adds the font size. For example, if the user has selected Rockwell, checked the bold checkbox, and specified a point size of 20 with the slider, `setfont()` creates the font name, `rockwell-bold-20`. `setfont()` uses `XtVaSetValues()` to set the textpane font to this font name.

Customizing the tutorial_main_stubs.c File

You created CallFunction connections for both checkbox and exclusive settings earlier in this tutorial. Since these connections were only for the base window, Golit generates the callbacks for them in the `tutorial_main_stubs.c` file. These callbacks have the names that you specified in Devguide and appear along with callbacks for the predefined connections you specified in Devguide.

Each of the CallFunction callbacks must set a field of the `characterformat` structure to reflect user input. They must then invoke `setfont()` to set the textpane font to that specified by the changed `characterformat` structure.

Modify the checkbox callbacks to set the font characteristics as follows:

```
void
set_bold(widget, clientData, callData)
        Widget widget;
        XtPointer clientData, callData;
{
        characterformat.bold = TRUE;
        setfont(widget);
}

void
set_unbold(widget, clientData, callData)
        Widget widget;
        XtPointer clientData, callData;
{
        characterformat.bold = FALSE;
        setfont(widget);
}

void
set_italic(widget, clientData, callData)
        Widget widget;
        XtPointer clientData, callData;
{
        characterformat.italic = TRUE;
        setfont(widget);
}

void
set_unitalic(widget, clientData, callData)
        Widget widget;
        XtPointer clientData, callData;
{
        characterformat.italic = FALSE;
        setfont(widget);
}
```

Modify the callbacks for the exclusive settings to set the font to `rockwell` or `lucida` as follows:

```
void
set_rockwell(widget, clientData, callData)
         Widget widget;
         XtPointer clientData, callData;
{
         characterformat.font = “rockwell”;
         setfont(widget);
}

void
set_lucida(widget, clientData, callData)
         Widget widget;
         XtPointer clientData, callData;
{
         characterformat.font = “lucidasans”;
         setfont(widget);
}
```

Customizing the tutorial_popup_stubs.c File

In Devguide you set up a connection that is activated by a change in the slider position. Golit generates the callback, `set_textsize(),` for this connection in `tutorial_popup_stubs.c`. Modify this callback so that it retrieves the new slider value from the `callData` variable passed to the function. Add code to convert that value to a string, and set the `size` variable of `characterformat` structure to it. Add a call to the `setfont()` function to apply the new font size:

```
void
set_textsize(widget, clientData, callData)
        Widget widget;
        XtPointer clientData, callData;
{
 /* Get slider value from callData  */
 int size = *(int * )callData;

 /* Convert the slider value to string  */
 sprintf(characterformat.size, “%1d”, size);

 /* Apply characterformat to the textpane  */
 setfont(widget);
}
```

Compiling and Running the Tutorial Application

To compile code for your sample application, you enter `make` in the shell tool. `make` uses the Golit-generated `Makefile` to compile code.

1. **Run `make` to compile and link the application.**

```
% make
```

   An executable file named `tutorial` is produced, assuming you have not made any errors.

2. **Run `tutorial` and test it.**

```
% tutorial
```

Golit Functionality in Detail 4

This chapter explains in detail how to use Golit to generate source code and a makefile. It suggests ways to alter the Makefile and the generated callback templates. It also describes how to use libgolit functions to instantiate parts of the generated widget tree and how to use Create Procs to implement your own widget creation routines.

Designing Golit Interfaces in Devguide

Before you use Golit, you must first use Devguide to design an interface and save it as a GIL file. The GIL file stores the interface using a generic description language that identifies each element and lists its properties.

To help keep your files straight during interface editing and compiling, you should create a single directory to store all the source code files for a single program: the `.P` and `.G` files for the interface, the Golit-generated source code files for the interface, your own custom source code files for the software underlying the interface, and the makefile.

Handling Events With Connections

To make your interface handle events, you specify *connections* in Devguide. When an event occurs on one object (called the *Source*) a connection triggers a specified action on a second object (called the *Target*).

Use the Connections Manager window to define connections. Connections are discussed in detail in the *OpenWindows Developer's Guide: User's Guide*.

The When menu items that appear for a specified Source object represent the events that can occur on that object. The Action menu items that appear for a specified Target object represent actions that are valid for that object.

Golitgenerates all the code necessary to execute any action except ExecuteCode and CallFunction. If you select ExecuteCode, a pop-up window which you can enter code in appears. Golit generates a callback function that includes this code in the `_stubs.c` file. If you select CallFunction, you must also specify a function name. Golit generates a template callback under that name in the `_stubs.c` file.

When Menu Items

The When menu items you can choose from are listed below, along with a brief description of when the corresponding connection will be invoked.

- Adjust - the user clicks the OPEN LOOK mouse ADJUST
- AnyEvent - any keyboard or mouse event
- Create - the source object is created (create proc)
- Destroy - the source object is destroyed
- Done - (before) popup window is dismissed
- DoubleClick - SELECT is pressed twice quickly
- DroppedUpon - drop target receives drop
- Enter - the pointer enters a window or pane
- Exit - the pointer exits a window or pane
- Keypress - the user presses any key on the keyboard
- Menu - the user clicks the OPEN LOOK mouse MENU
- Notify - the user selects a control UI element
- Popdown - menu pops down
- Popup - menu pops up
- Repaint - the user repaints a window or pane
- Resize - the user resizes a window or pane
- Select - the user clicks the OPEN LOOK mouse SELECT
- Unselect - the user lets up the OPEN LOOK mouse SELECT

Note that Golit generates the same resource for many of the When items. It is up to the application to distinguish between them. Table 4-1 on page 49 lists the resource generated for each When item.

Table 4-1 When Items and the Resources Generated for Them

When Item	**Resource**
Adjust	XtNconsumeEvent
AnyEvent	XtNconsumeEvent
Destroy	XtNdestroyCallback
Done	XtNpopdownCallback
DoubleClick	XtNconsumeEvent
DroppedUpon	XtNdndTriggerCallback
Enter	XtNconsumeEvent
Exit	XtNconsumeEvent
KeyPress	XtNconsumeEvent
Menu	XtNconsumeEvent
Notify	XtNselect, XtNsliderMoved, or XtNverification
Popdown	XtNpopdownCallback
Popup	XtNpopupCallback
Repaint	XtNexposeCallback
Resize	XtNresizeCallback
Select	XtNconsumeEvent
Unselect	XtNunselect

Note that the resource generated for Notify depends on the source object.

Action Menu Items

The Action menu items you can choose from are listed below, along with a brief description of their functionality.

- CallFunction - calls the specified function
- Disable - disables the Target object
- Enable - enables the Target object
- ExecuteCode - executes the specified code
- GetLabel - returns the Target object's label

- GetValueNumber - returns the Target object’s numeric field value
- Hide - makes the Target object invisible
- LoadTextfile - loads the specified text file
- SetLabel - sets the Target object’s label
- SetValueNumber - sets the Target object’s numeric field value
- Show - displays the Target object

Setting up Create Procs to Conduct Your Own Widget Creation

To conduct some operations, such as setting fonts or registering help, you must intercept Golit’s default widget creation and provide your own widget creation routines. To do this, you use *Create Procs*. A Create Proc is a special type of callback that is invoked when a user interface object is created.

To generate a Create Proc, you specify a connection for it in Devguide. You must make sure the event is set to Create, and the Source and Target are both set to the object for which you want to generate the Create Proc. You must also set the Action to CallFunction. You can specify a single function as the Create Proc for several objects.

Note – If you do not set the Source and Target to the same object when you specify a Create Proc in the Devguide Connections Manager window, Golit will not generate the Create Proc template.

Using Golit to Generate Code

`golit` is the executable program that takes the Devguide GIL (`.G`) or project (`.P`) file and generates C files and the `Makefile`. The format of the `golit` command is as follows:

```
golit [option flags] <UIFileName>
```

Where `UIFilename` is the name of the GIL or project file that you want to generate code for.

GolitCommand-Line Options

Golit provides the following command-line options:

-h (-help)

Displays a message describing Golit usage and options.

l-i or -international

Writes all Leve 3 internationalization-specific resources into a resource file. For more information on internationalizing OLIT applications, see Chapter 6, “Internationalization.”

-k or -kandr

Writes K&R C code that doesn’t contain function prototypes. Golit produces ANSI C code by default.

-m (-main)

Generates code only for the `main()` program `<project_name>.c` and `<project_name.h>` files. Only works with `the -p` option. Use the `-m` option if you have already run Golit on a project’s `.G` files.

-n or -nomerge

Keeps Golit from merging existing and new `_stubs.c` files. If you are prototyping an interface and have never added any code to the `_stubs.c` file, you should run Golit in `-n` mode. This prevents unused callbacks from accumulating in your `_stubs.c` file. If you have added code to your `_stubs.c` file, don‘t use `-n`, since Golit will omit the code you added.

-p project

Generates code for a project. When you use `-p`, `UIFileName` must be a project name.

-r or -resources

Writes *all* resources into a resource file.

-s or -silent

Instructs Golit to operate silently; no messages will appear.

Files Generated for a Single GIL File

Golit names its generated files after the original GIL filename. It strips off the `.G` extension and adds new extensions to identify each file. If you generate code for a single GIL file, Golit generates the following files:

- `<GIL_filename>_ui.c`
- `<GIL_filename>_ui.h`
- `<GIL_filename>_stubs.c`

For example, if you use Golit to generate source code for a GIL file named `newt.G`, Golit generates the files `newt_ui.c`, `newt_ui.h`, and `newt_stubs.c`. Golit also creates a makefile under the name `Makefile`, if it doesn't already exist. Figure 4-1 provides an overview of the files Golit generates for a single GIL file (`fn`) and the other files necessary to create an executable program.

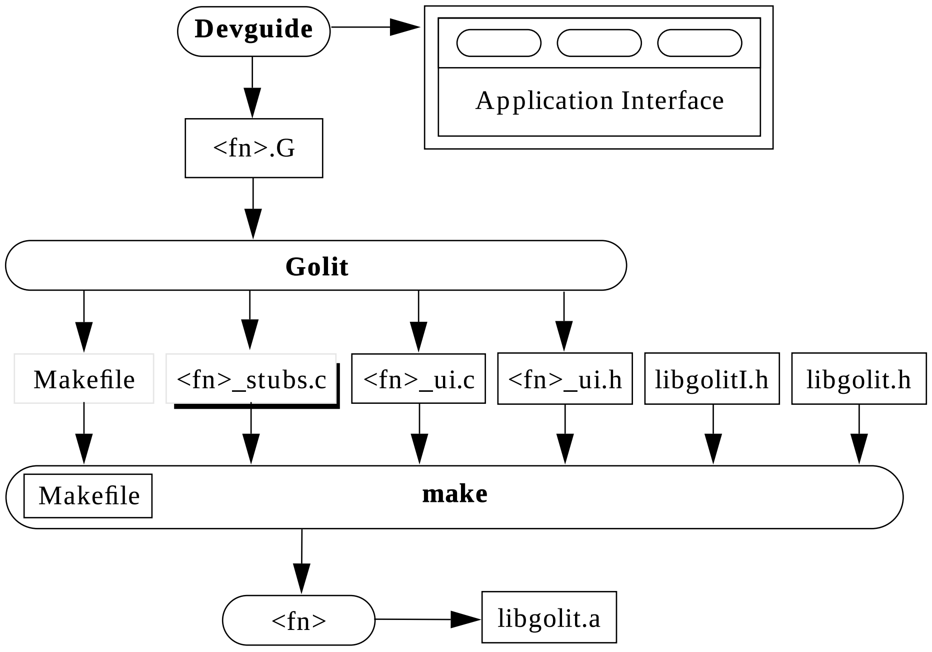


Figure 4-1 Overview of Golit Files for a Single GIL File

In Figure 4-1, the following conventions apply:

▭ Rounded boxes indicate programs.

▭ Square boxes indicate data files.

▭ Boxes with drop shadows indicate files for which <fn>.BAK files are created.

▭ Dashed boxes indicate data files that you can modify.

_ui.c File

This file calls macros to create a widget tree that describes the interface in the GIL file. The widget tree comprises statically initialized data structures (*widget descriptors*) that describe the objects in the user interface. Each widget descriptor provides the following information:

- a reference to the object's immediate sibling
- a reference to the object's children
- a reference to the object's popups
- the instance name of the object
- the object's class
- a reference to the object's resources and their values
  - ArgList
  - Typed ArgList
- a reference to the object's callbacks
- a method of instantiating the object

The libgolit routines in the generated `main()` program and your application code use the widget descriptors to instantiate widgets.

Note – Do not modify the `_ui.c` file by hand. Any changes you make to this file will be lost when you regenerate code.

Note that there isn't a one-to-one correspondence between objects you design in Devguide and widget descriptors in the tree. Many Devguide objects consist of a hierarchy of several OLIT widgets. For example, a Text Field object with a label in Devguide is actually two widgets in the Golit generated code: a textField widget and a caption widget.

The `_ui.c` file provides external *widget description pointers* that enable you to access widget descriptors. When Golit generates the widget hierarchy for a Devguide object, it bases the widget description pointer names on the name of the object. If a Devguide object comprises only one widget, Golit creates the widget description pointer by joining the GIL file name and the Devguide object name. For example, the widget description pointer for a Message object called `mymessage` in a GIL file named `mygilfile` will be `Mygilfile_mymessage`. Note that the first letter of the name is automatically capitalized.

If a Devguide object comprises more than one widget, Golit adds different prefixes to the object's name to create the widget descriptors. For example, if you create a Text Field object (with a label) named `mytextfield` in `mygilfile`, Golit names the textfield widget description pointer `Mygilfile_mytextfield` and the caption widget description pointer `Mygilfile_G_Caption_mytextfield.`

Figure 4-2 and Figure 4-3 show examples of widget trees created by Golit. If you create an interface (called `fn`) with a base window (`window1`) that has a single control area (`Controls1`) with a textfield in it (`textfield1`) and a single popup window (`popup1`), Golit generates the widget tree depicted in Figure 4-2. If you create the base window with a footer, Golit generates the widget tree depicted in Figure 4-3. The widget class for each widget in these trees is shown in parentheses. The solid lines indicate a parent-child relationship. The dotted line indicates a sibling relationship.

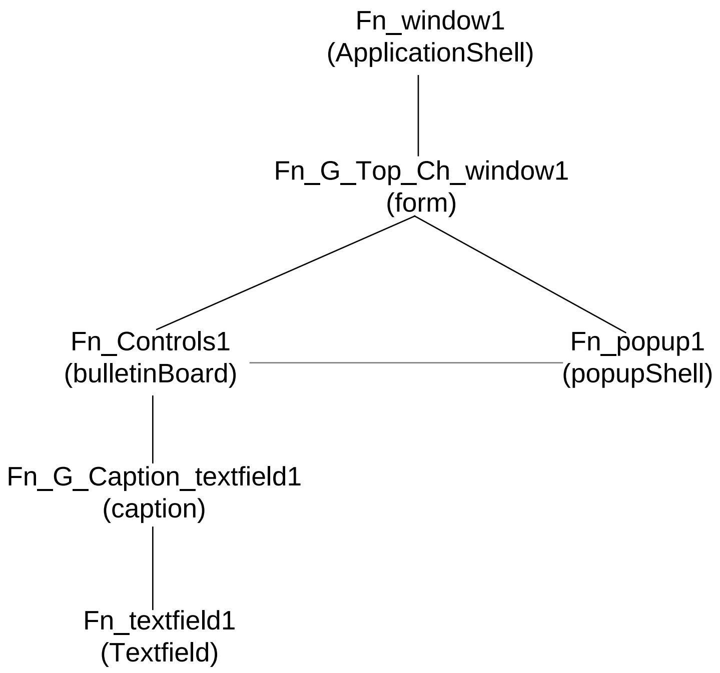


Figure 4-2 Example Widget Tree for a Base Window

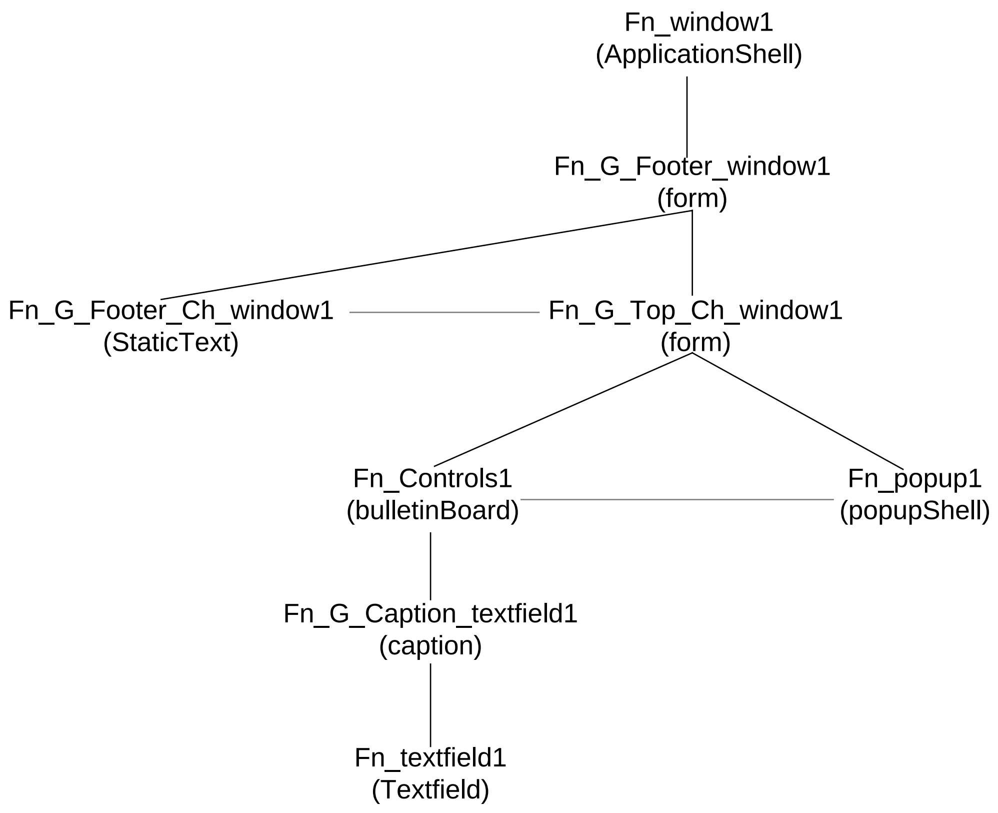


Figure 4-3 Example Widget Tree for a Base Window with a Footer Panel

Table 4-2 and Table 4-3 provide a list of the widgets used to created each Devguide Object.

_ui.h File

This file contains prototypes of all the functions and callbacks. The Golit code generator automatically places an `#include` statement in the `_ui.c` file to include this header file. Do not modify the `_ui.h` file by hand.

Normally, you don’t need to look at the `_ui.h` file. However, it can help you quickly ascertain the names of Golit generated widget descriptions available to the application programmer. At the top of every `_ui.h` file is a series of macros that declare all the generated widget description pointers available to the application programmer. These macros take the form `ExternWidgetDesc(wdp)`, where `wdp` is a widget description pointer, such as `Fn_window1` or `Fn_G_Top_Ch_window1`.

_stubs.c

The `_stubs.c` file consists of two principal parts: the `main()` program and the callbacks.

The main() Program in the _stubs.c File

The `main()` program performs initialization for the Xt Intrinsics and the OLIT toolkit and sets up an event handling loop. It uses libgolit functions to instantiate the widget tree: it calls `GolitFetchShellHier()` to instantiate the base window and all of its children and `GolitFetchWidgetHier()` to instantiate each popup (and all of its children) in your interface. These functions traverse the widget tree described in the `_ui.c` file, instantiating a widget for each widget descriptor in the tree.

The libgolit functions give each widget they create an *instance name*. The instance names of the widgets created in the `main()` program are the same as those of the widget description pointers that describe them. For example, if you create a base window and call it `basewindow` and save it in a GIL file called `fn`, the base window’s widget description pointer *and* its instance name will be `Fn_basewindow`.

The `main()` program also provides code that applies any fallback resources you have specified.

Callbacks in the _stubs.c File

For each connection you specify in Devguide, Golit generates a callback function template in the `_stubs.c` file. There are two principal varieties of callbacks:

- Predefined action callbacks
  These templates are generated for connections for which you specified a Devguide-defined action (for example, Show or Hide). The template contains all the code necessary to carry out the action and a `printf()` statement that prints the callback name to the console.

- CallFunction callbacks
  These templates are generated for connections for which you specified CallFunction as the action in Devguide. The template uses the function name that you specified in Devguide. It is empty except for the `printf()` statement.

You can add your application code to either type of callback template. Note that predefined action templates already provide a `GolitNameToWidget()` call to retrieve the target widget id.

The following arguments are passed to callbacks:

- widget - the Source widget specified for the connection
- clientData - not implemented
- callData - widget specific data for the callback, for example, slider values

Makefile

This file is a standard C template makefile that builds the executable. When you run Golit, it checks the current directory to see if a file called `Makefile` already exists. If there is no `Makefile`, it generates one. If it detects a `Makefile`, it does not generate one. This feature protects a custom makefile, ensuring that Golit doesn't overwrite it.

You can easily customize `Makefile` to compile your own code files. The beginning of the `Makefile` lists the source files in a section labeled "Parameters." The five parameters are:

- `PROGRAM`, which lists the name of the executable file created during compilation

- `SOURCES.c`, which lists the names of user-supplied C source files for the application

- `SOURCES.h`, which lists the names of user-supplied header files for the application

- `SOURCES.G`, which lists the names of GIL files used to store interfaces for the application

When Golit first generates the `Makefile`, the `SOURCES.c` and `SOURCES.h` parameters are not set. To add your own source code files to the Parameters list, you must add their filenames by hand. For example, if you have several C source files that provide functions called in the callbacks, you must include their names in the `SOURCES.c` parameter.

Files Generated for a Project

If you run Golit on a project file, Golit generates `_ui.c`, `_ui.h` and `_stubs.c` files for each GIL file in the project. It also generates the following files:

- `<project_name>.c`
- `<project_name>.h`
- `<project_name>.make`

<project_name>.c File

When you generate code for a project, Golit generates the `main()` function in the `<project_name>.c` file.

This file also contains all callback functions that are common to more than one `.G` file in your project. For example, suppose you create a project which includes two menus saved in separate `.G` files. Both menus have a Save menu item for which you specify a CallFunction connection to a function called `mysave`. Golit generates the code for `mysave` in the `.c` file. The `mysave` callback does not appear in the `_stubs.c` file for either menu.

You can add variable initializations and other application code to the `<project_name>.c` file.

<project_name>.h File

This file contains the external declarations for the interface objects in the project. It includes the `_ui.h` files from the GIL files in the project and any objects common to the GIL files. You can add your own declarations to this file.

<project_name>.make File

This file contains lines that are inserted into the makefile to compile the `_stubs.c` files from the different GIL files in the project.

Regenerating Code for a Modified Interface

To regenerate code for an interface after you have modified it, simply re-run Golit on the GIL or project file. If you want to re-run Golit and immediately proceed to compile the generated code in one step, just type `make`.

When you re-run Golit, it overwrites the original `_ui.c` and `_ui.h` files. However, it preserves any application code which you have added to the `_stubs.c`, `<project_name>.c` and `<project_name>.h` files by merging this code in with any new code it generates.

Note – If you change the action for a connection in Devguide, the change will not be reflected in the new `_stubs.c` or `<project_name>.c` file unless you manually delete the original callback for that connection before you re-generate code.

Regenerating Code for an Individual GIL File

If you re-run Golit on a single GIL file, it generates the following files:

- new _stubs.c - Golit creates this file by merging the old _stubs.c with any new code it generates. Any source code you added to the old _stubs.c file is preserved in the new one.
- `_stubs.c.BAK` - back-up of the original `_stubs.c` file
- `_stubs.c.delta` - a list of differences between the original and the new `_stubs.c` file

Regenerating Code for a Project

If you re-run Golit on a project, it merges the `_stubs.c` for each GIL file in the project. It also generates the following files:

- new <project_name>.c - Golit creates this file by merging the old <project_name>.c file with any new code it generates. Any source code you added to the old <project_name>.c is preserved in the new one.
- new <project_name>.h - Golit overwrites the existing <project_name>.h file
- `<project_name>.c.BAK` and `<project_name>.h.BAK` - backups of the original `<project_name>.c` and `<project_name>.h` files
- `<project_name>.c.delta` - a list of differences between the original and the new `<project_name>.c` files

Removing Obsolete Callbacks

When Golit merges the original `_stubs.c` and <`project_name>.c` into the new files, it does not delete or replace any of the original callbacks. This results in unused callbacks accumulating in the following cases:

- When you remove an object or a connection in Devguide
  Golit does not delete callbacks associated with deleted connections or connections for an object that has been deleted. You must remove these callbacks manually.
- When you change the name of an object or connection in Devguide
  Golit adds a new callback for any connection that has a new name, or that connects objects that have new names. It retains the original callback along with any application code you inserted in it. Be sure to transfer that application code to the new callback. You can then delete the original callback.

Integrating Your Application Code with the Interface Code

To add your application code to the Golit-generated interface code, you insert it in the callback templates in the `_stubs.c` file. Simply replace the `printf()` statement in each callback with your own code. Of course, you can include calls to functions that you keep in other source files. Just remember to modify the `Makefile` so that the Make utility will compile these files.

There are essentially three things your application code can do to the interface you designed in Golit:

- change widget resources
- create additional instances of parts of the widget tree

- intercept Golit default widget creation and provide a custom creation procedure (*Create Proc*).

These operations are discussed in the subsections below.

Changing Widget Resources

To change a widget's resources, you need to know the widget's id. Since the `main()` program generated by Golit does not return all the widget ids when it creates the interface, you have to include code to get the widget ids you want. The libgolit library provides functions that enable you to retrieve ids.

Retrieving Widget ids

The `GolitNameToWidget()` function returns a widget id when you pass it a widget instance name, and a location in the tree to start searching for the widget. You use it as follows:

```
Widget w;

w = GolitNametoWidget(root, “widget_name”);
```

where:

`widget_name` is the instance name of the widget for which you want to get an id.

`root` is a widget id that specifies where you would like to start searching the widget tree for the widget designated by `widget_name`. Normally, you can just use the widget that is passed to the callback.

The `GolitFetchLastWidgetID()` function returns the id of the last widget created of a given type. You use it as follows:

```
Widget w;

w = GolitFetchLastWidgetID(last_widget_description_ptr);
```

where:

`last_widget_description_ptr` is the widget description pointer of the widget for which you want to get an id. This will be something like `Fn_window1`, or `Fn_button2` (where `fn` is the name of the GIL file).

Getting the Right Widget Instance Name

As discussed in “_ui.c File” on page 54, there isn’t a one-to-one correspondence between objects you design in Devguide and widgets in the widget tree. Many Devguide objects consist of a hierarchy of several OLIT widgets.

When you specify a connection for an object in Devguide, the generated callback is only passed the id of one of the widgets that the source object comprises. Normally this design will not pose any problems since the returned id is for the widget you are most likely to want to manipulate.

For example, if you put a Text Field object with a label in your interface in Devguide, Golit creates a textField widget and a caption widget. If you set up a CallFunction connection for the Text Field object, the callback is only passed the textField widget (and not the caption widget). Normally, this is convenient since you are more likely to want to manipulate the textField widget. If you want to manipulate the caption, you must find out its instance name and use `GolitNameToWidget()` to get its id.

It is relatively easy to figure out the instance name. For the widgets that the Golit-generated `main()` instantiates, the instance names are the same as the widget description pointers. Golit bases the widget description pointers on the names of the corresponding Devguide objects. Table 4-2 and Table 4-3 show Devguide objects and the widget hierarchy that Golit uses to create each of them. The Widget Description Pointer column shows the patterns which Golit uses to create each widget description pointer (assuming the GIL file is called `fn`). In each widget hierarchy, the widget that appears in bold type is the one that is passed to callbacks for the object. Note that if you do not specify a label for an object in Devguide, Golit omits the caption widget from the object’s hierarchy.

To demonstrate how you use Table 4-2 and Table 4-3, take the example of a Exclusive Setting object. In Devguide, you name the setting `setting1`. You provide two choices: `choice1` and `choice2`. You provide a label for the setting and save it in a GIL filed called `fn`. Golit creates the following widget description pointers (and instance names) for the Setting object:

`Fn_G_Caption_setting1`	(caption widget)
`Fn_setting1`	(Exclusives widget)
`Fn_setting1_choice1`	(Rectbutton widget)
`Fn_setting1_choice2`	(Rectbutton widget)

Note that the first letter of each name is capitalized. If you were to set up a Callfunction callback for the setting object, the callback would be passed the id of `Fn_setting1` (the Exclusives widget).

Table 4-2 Devguide Panes and Windows and their Golit Widget Hierarchies

Devguide Object	Widget Hierarchy	Widget Description Pointer
Control Area	**bulletinBoard**	Fn_<objectname>
Canvas	scrolledWindow	Fn_G_Scr_Win_<objectname>
	DrawArea	<objectname>
Text Pane	scrolledWindow	Fn_G_Scr_Win_<objectname>
	textEdit	<objectname>
Base Window	**applicationShell**	Fn_<objectname>
	form	Fn_G_Top_Ch_<objectname>
Base Window	**applicationShell**	Fn_<objectname>
(with footer)	form	Fn_G_Footer_<objectname>
	form	Fn_G_Top_Ch_<objectname>
	StaticText	Fn_G_Footer_ch_<objectname>
Popup Window	**popupShell**	Fn_<objectname>
Menu	**menuShell**	Fn_<objectname>

Table 4-3 Devguide Control Objects and their Golit Widget Hierarchies

Devguide Object	**Widget Hierarchy**	**Widget Description Pointer**
Scrolling List	caption*	Fn_G_Caption_<objectname>
	ScrollingList	Fn_<object_name>
Slider	caption*	Fn_G_Caption_<objectname>
	Slider	Fn_<object_name>
Gauge	caption*	Fn_G_Caption_<objectname>
	Gauge	Fn_<object_name>
Text Field	caption*	Fn_G_Caption_<objectname>
	TextField	Fn_<object_name>
Multiline Text Field	caption*	Fn_G_Caption_<objectname>
	scrolledWindow	Fn_G_Scr_Win_<objectname>
	TextEdit	Fn_<object_name>
Exclusive Settings	caption*	Fn_G_Caption_<objectname>
	Exclusives	Fn_<objectname>
	RectButton**	Fn_<object_name>_<buttonname>
Nonexclusive Settings	caption*	Fn_G_Caption_<objectname>
	Nonexclusives	Fn_<objectname>
	RectButton**	Fn_<object_name>_<buttonname>
Checkbox Settings	caption*	Fn_G_Caption_<objectname>
	Exclusives	Fn_<objectname>
	CheckBox**	Fn_<object_name>_<checkboxname>
Settings Stack	caption*	Fn_G_Caption_<objectname>
	AbbrevMenuButton	Fn_G_Abbrev_<objectname>
	Exclusives	Fn_<objectname>
	RectButton**	Fn_<objectname>_<buttonname>
Button	**OblongButton**	Fn_<objectname>
Menu Button	**MenuButton**	Fn_<objectname>
Message	**StaticText**	Fn_<objectname>

*If you do not specify a label in Devguide, Golit does not create the caption widget.
**Buttons and boxes in settings can have their own callbacks.

Instantiating Parts of the Widget Tree with libgolit

The `main()` program generated by Golit calls libgolit functions to create one instance of each object in your interface. You can also use the libgolit functions within your application code to create additional instances of an object or part of the widget tree. You simply re-use the widget descriptors in the `_ui.c` file and provide different instance names.

If your interface uses a number of objects that are similar, you can save time by instantiating the objects with libgolit library functions. Instead of using Devguide to design each possible object, you just design one object. You then include code to repeatedly instantiate the object and customize it as needed at runtime. This saves you time and makes your code much more efficient.

The following are the libgolit functions you can use to instantiate parts of the widget tree:

- `GolitFetchWidget()` - instantiates the specified widget
- `GolitFetchWidgetUnmanaged()` - instantiates the specified widget, but does not add it to its parent's managed set
- `GolitFetchChildren()` - instantiates the children of the specified widget
- `GolitFetchChildrenUnmanaged()` - instantiates the children of the specified widget, but does not add them to the widget's managed set
- `GolitFetchPopups()` - instantiates the popups of the specified widget
- `GolitFetchPopupsUnmanaged()` - instantiates the popups of the specified widget, but does not add them to the widget's managed set
- `GolitFetchWidgetHier()` - instantiates the specified widget, its popups, and its children
- `GolitFetchWidgetHierUnmanaged()` - instantiates the specified widget, its popups, and its children but does not add the widget to its parent's managed set.
- `GolitFetchShellHier()` - instantiates the shell widget and all its children

Since there isn't always a one-to-one correspondence between objects you design in Devguide and widgets in the widget tree, you should be careful to use a function that creates the portion of the object you want. When in doubt, it is usually safest and easiest to use `GolitFetchWidgetHier()`.

The functions all have similar arguments. The following is the syntax for `GolitFetchWidgetHier()`, the function that you will probably use most frequently:

```
Widget
GolitFetchWidgetHier (String  name,
 GolitWidgetDescPtr wdp,
 Widget parent,
 ArgList args,
 Cardinal num_args,
 XtPointer closure);
```

Where:

`name`	The instance name of the widget whose hierarchy you want to instantiate.
`wdp`	The widget description pointer. This refers to an object that you set up in Devguide, for example, `Fn_button1` or `Fn_popup2`. `GolitFetchWidgetHier()` uses the widget descriptor specified by `wdp` to set the resources of the widget.
`parent`	The id of the parent of the widget you want to fetch.
`args`	A pointer to the Arg structure for the widget you want to fetch.
`num_args`	The number of args in args.
`closure`	A pointer to the application data structure that you want to pass down the widget tree.

Example of Using libgolit to Instantiate Part of the Widget Tree

Let's say you want to create a popup data entry form. You want the user to be able to open several separate copies of the form at once; each one should be numbered so the user can tell which one is which. However, you don't know exactly how many copies the user might want to open.

If you were to create separate forms in Devguide, you would have to create an arbitrarily large number. This would be an awkward process and would create bulky and inefficient interface code. With libgolit routines, there is a much easier way to solve this problem. You just prototype one form in Devguide and add a libgolit call to the `_stubs.c` file to instantiate the form whenever the user presses a button:

1. **Create an interface with a base window and a popup.**
   Put whatever elements the form includes in the popup. Use Devguide's default names for the objects.

2. **Create a button in the base window.**
   Specify a connection for the button that shows the popup - that is, set the Source to button1, the Target to popup1, and the Action to Show.

3. **Save the interface and run Golit on it**
   For this example, call the interface `ex2`. The resulting `_stubs.c` file will contain a callback like the one shown in Figure 4-4.

```
void
button1_connectionCB1(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
      Widget target = GolitNameToWidget(widget,
                                "Ex2_popup1");

      XtPopup(target, XtGrabNone);
      printf("connection: button1_connectionCB1\n");
}
```

Figure 4-4 Generated Callback that Displays a Popup Window

4. **Modify the** `button1_connectionCB1()` **callback.**
   a. **Add code that creates a sequential instance name for the popup.**
   b. **Call** `GolitFetchWidgetHier()` **to instantiate the popup.**
      Use the popup object you designed in Devguide as the prototype (or widget description pointer) and the sequential name as the instance name.

c. **Customize the instance of the popup.**
Call `GolitNameToWidget()` with the popup widget's instance name to get its id. Then use Intrinsics functions to set the popup's title to its instance name.

The modified `button1_connectionCB1()` in Figure 4-5 demonstrates how you might do this. This modified callback gives each popup a title of the pattern, "popup<n>", where n is the number of the popup. Note that the callback should not attempt to instantiate the popup the first time the user presses the button. This is because the main program has already instantiated the entire widget tree—including the popup—once.

```
void
button1_connectionCB1(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
      char       buf[32];
      Arg        arg[1];
      static int popup_number;
      Widget     target;
      /* Create sequential instance name in buf for each popup */
      sprintf(buf,"popup%1d",++popup_number);

      /* Instantiate a popup for each press after the first */
      if (popup_number > 1)
      {
      target = GolitFetchWidgetHier(buf, Ex2_popup1, widget,
                                 NULL, NULL, NULL);
      }
      /* Get the popup's widget id */
      target = GolitNameToWidget(widget,buf);

      /* Set the popup's title to match its instance name */
      XtSetArg(arg[0], XtNtitle, buf);
      XtSetValues(target, arg, 1);

      /* Make the popup appear */
      XtPopup(target, XtGrabNone);
}
```

Figure 4-5 Callback that Displays a New Popup Window for each Button Press

Using Create Procs to Provide Your Own Widget Creation

Golit's design allows you to intercept Golit's default widget creation and provide your own creation procedure (Create Proc). This enables you to perform special operations immediately before and after widget creation. For example, prior to creation, you may want to specify some resources. After widget creation you may want to:

- Get a widget id
- Add help for a widget with `OlRegisterHelp()`
- Use `SetValues()` on a widget just after its creation
- Use a specific font for a user interface object

To set up a Create Proc in Devguide, see "Setting up Create Procs to Conduct Your Own Widget Creation" on page 50. Each time an object is instantiated, the Create Proc you have assigned to it will be called. A Create Proc must return a widget id. The Create Proc template that Golit generates includes a call to the appropriate libgolit function (for example, `GolitFetchWidgetHier()`) to return the widget id; this function also instantiates the widget and its children. The arguments for the function are the same as those passed to the Create Proc.

Caution – Treat Create Procs as you would treat any other callback. Do not call them from your application code.

Example of using a Create Proc to Register Help for an Object

Let's say you have designed a Text Field in Devguide and you want to register help for it at its creation time. To do this, you make a Create Proc for the Text Field object:

1. **Set up the Create Proc in Devguide.**
   Set up a connection for the Text Field Object. Make sure that both the Source and the Target are set to the object (`textfield1` in this example). Set the When setting to Create and the Action to CallFunction. Call the function `create_textfield`.

2. **Use Golit to generate the code.**
   Golit generates a template similar to that shown in Figure 4-6.

```
Widget
create_textfield1(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      printf("create proc: create_textfield1\n");
      return(GolitFetchWidgetHier(name, wdp, parent, args,
                                   num_args, closure));
}
```

Figure 4-6 Template for the Create Proc, `create_textfield`()

3. **Modify the `create_textfield()` callback.**
   a. **Use `GolitFetchWidgetHier()` to instantiate the textField widget and save the widget's id.**
   b. **Use `OlRegisterHelp()` to perform the help registration.**
   Figure 4-7 shows how to modify `create_textfield1()` to do this.

Note that the only widget that `GolitFetchWidgetHier()` gets an id for is the textField widget itself. It does not get the id for the caption widget that displays the Text Field caption or for any of the other widgets in the hierarchy. This caption widget is above the textField widget on the hierarchy (see Table 4-3 on page 65) and has already been instantiated. This means that you have only registered help for the textField widget. The help message will not appear if the cursor is on the Text Field label when the user presses the help button.

```
Widget
create_textfield1(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
Widget textfield;

      /* Instantiate the textfield and save the id  */
      textfield = GolitFetchWidgetHier(name, wdp, parent, args,
                                num_args, closure);

      /* Set help text to the specified string  */
      OlRegisterHelp(OL_WIDGET_HELP,
                 textfield, “TextField Help”,
                 OL_STRING_SOURCE, “Enter some text on the
line.”);

      return textfield;
}
```

Figure 4-7 Create Proc Modified to Register Help for a TextField Widget

To register help for the Text Field caption:

1. **Figure out the caption widget’s instance name.**
   Assuming you called the interface `ex3`, you can do this by adding the prefix, `Ex3_G_Caption_` , to the Text Field object’s name. Since the Text Field object’s name is `textfield1`, the caption widget’s name will be `Ex3_G_Caption_textfield1`.

2. **Use `GolitNametoWidget()` to get the caption’s widget id.**

3. **Use `OlRegisterHelp()` to perform the help registration.**

Figure 4-8 shows how to modify the callback to do this.

```
Widget
create_textfield(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
Widget textfield, caption;

      /* Instantiate the textField widget and save the id  */
      textfield = GolitFetchWidgetHier(name, wdp, parent, args,
                                 num_args, closure);

      /* Set help text to the specified string  */
      OlRegisterHelp(OL_WIDGET_HELP, textfield, “TextField
Help”,
                 OL_STRING_SOURCE, “Enter some text on the
line.”);

      /*  Get the widget id for the caption.  */
      /*  Start searching at the textfield widget */
      caption = GolitNameToWidget(textfield,
                                “Ex3_G_Caption_textfield1”);

      /* Set help for the caption to the specified string */
      OlRegisterHelp(OL_WIDGET_HELP, caption, “TextField Help”,
           OL_STRING_SOURCE, “Your cursor is on the caption.”);

      return textfield;
}
```

Figure 4-8 Create Proc Modified to Register Help for both a Text Field and its Caption

Example of Using Create Procs to Set Fonts

Let’s say you have designed an interface with a Checkbox Setting and you want to set the font for both the checkboxes and the Setting label at creation time.

To do this, you set up connections that specify Create Procs for each of the objects. Specify the same function name for the individual checkboxes' Create Procs (for example, `create_checkbox`). Specify another function name for the Setting Create Proc (for example, `create_setting`). Golit will generate Create Proc templates similar to those shown in Figure 4-9.

```
Widget
create_checkbox(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      printf("create proc: create_checkbox\n");
      return(GolitFetchWidgetHier(name, wdp, parent, args,
       num_args, closure));
}

Widget
create_setting(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      printf("create proc: create_settings\n");
      return(GolitFetchWidgetHier(name, wdp, parent, args,
       num_args, closure));
}
```

Figure 4-9 Create Proc Templates for a Checkbox and a Setting

Setting the font for the checkboxes is relatively simple. You just modify the `create_checkbox()` Create Proc so that it uses `GolitFetchWidgetHier()` to instantiate the checkbox widget. Then you use XtVaSetValues to set the font (see Figure 4-10). Any checkbox that you specified `create_checkbox()` as the Create Proc for will have a bold label.

Setting the font for the Setting label is not quite as simple. The `create_settting()` Create Proc instantiates the nonExclusives widget, not the caption widget that shows the label. Therefore, you must you must first retrieve the caption widget's id and then set the font for that widget. To get the id, you use `GolitNameToWidget()`. `GolitNameToWidget()` requires the caption widget's name. To get this name, you add the prefix, `Ex4_G_Caption_` , to the Setting object's name. In this example, the name would be `Ex4_G_Caption_setting1`. Figure 4-10 shows how you might modify the `create_settings()` callback to set the font for a Setting label.

```
Widget
create_choice(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      Widget checkbox;
      /*  Instantiate checkbox and get an id  */
      checkbox = GolitFetchWidgetHier(name, wdp, parent, args,
                                   num_args, closure);
      /* Set the checkbox label’s font  */
      XtVaSetValues(checkbox, XtVaTypedArg, XtNfont, XtRString,
           “lucidasans-bold”, strlen(“lucidasans-bold”) + 1,
           NULL);
      return(checkbox);
}
Widget
create_setting(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      Widget setting, caption;
      /* Instantiate setting and save id */
      setting = GolitFetchWidgetHier(name, wdp, parent, args,
                             num_args, closure);
      /* Get id for setting’s caption  */
      caption = GolitNameToWidget(setting,
                      “Ex4_G_Caption_setting1”);
      /* Set the setting caption’s font  */
      XtVaSetValues(caption, XtVaTypedArg, XtNfont, XtRString,
           “lucidasans-bold”, strlen(“lucidasans-bold”) + 1,
            NULL);
return(setting);}
```

Figure 4-10 Checkbox and Setting Create Procs Modified to Change Fonts

Compiling

After you've created an interface with Devguide, generated source code files with Golit, and created your own custom source code files, make sure that you are ready to compile. Make sure the environment variables GUIDEHOME and OPENWINHOME are set to point to the Devguide and OpenWindows' home directories respectively. Edit the `Makefile` to include any of your own source code files in the Parameters section. Enter the names of all your source code files in the `SOURCES.c` line and the names of all your header files in the `SOURCES.h` line.

Once your `Makefile` is set to show all of the associated source code files, you compile them by entering the command **`make`**. It runs according to the contents of the `Makefile`: it first checks all files specified in the `SOURCES.G` parameter to see if any have been changed since the last compile. It then uses Golit to generate fresh source code files if you changed the GIL file. It finishes by compiling and linking all specified source code files.

The compiled code is placed in the file named in the `PROGRAM` parameter of the `Makefile`. To run it, simply enter the filename on the command line.

Advanced Topics 5

This chapter provides advanced examples of callbacks that you may find useful in writing your own applications.

Specifying an Icon for an Application

Golit does not allow you to specify an icon for your application in Devguide. However, you can write a Create Proc that displays a specified icon when the application is in an iconified state. To do this, you set up a Create Proc in Devguide for the Base Window. Assuming you call the icon `my.xbm`, you would modify the generated callback to appear like the one shown in Figure 5-1. You can create an icon with bitmap or IconEdit (make sure that the format is set to X Bitmap).

```
. . .
/* Put this line at top of  _stubs.c file  * /
#include “my.xbm”

. . .
Widget
create_window1(app_name, app_class, wdp, dpy, args, num_args,
               closure)
      String app_name;
      String app_class;
      register GolitWidgetDescPtr wdp;
      Display *dpy;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      Widget toplevel;
      Pixmap toplevel_icon;
      Screen *screen;
      unsigned long fg, bg;
      unsigned int depth = 1;
      Arg arg;

      toplevel = GolitFetchShellHier(app_name, app_class, wdp,
                              dpy, args, num_args, closure);
      screen = XtScreen(toplevel);
      toplevel_icon = XCreateBitmapFromData(
                     XtDisplay(toplevel),  /* display */
                     RootWindowOfScreen(screen), /* drawable */
                     myicon_default_bits, /* bitmap data */
                     myicon_default_width,  /* width */
                     myicon_default_height /* height */);
      XtVaSetValues(toplevel,
           XtNiconPixmap, (XtArgVal)toplevel_icon,
           XtNiconName, (XtArgVal)”toplevel”,
           NULL);

      return toplevel;
}
```

Figure 5-1 Base Window Create Proc Modified to Display Icon

Making a Text Field Read-Only

In OLIT there is no direct resource to make a Text Field read-write or read only. The only way to change this aspect of the Text Field's behavior is to access the textEdit widget that is associated with the Text Field and then set the resource XtNeditType with the appropriate value. Specify a create proc for the text field object and modify it to look like the code in Figure 5-2.

```
Widget
textfield1CP(name, wdp, parent, args, num_args, closure)
 String name;
 register GolitWidgetDescPtr wdp;
 Widget parent;
 ArgList args;
 Cardinal num_args;
 XtPointer closure;
{
Widget textfield;
Widget textedit;
Arg arg;

textfield = GolitFetchWidget(name, wdp, parent, args, num_args,
                         closure);
      XtSetArg(arg, XtNtextEditWidget, &textedit);
      XtGetValues(textfield, &arg, 1);
      XtSetArg(arg, XtNeditType, OL_TEXT_READ);
      XtSetValues(textedit, &arg, 1);
      return textfield;
}
```

Figure 5-2 Text Field Create Proc Modified to Make the textEdit Widget Read-Only

Specifying a Label Image for a Button

You can also use Create Procs to specify label images for a button. To do this, set up a Create Proc for the button. Assuming you call the glyph `mylabel.xbm`, you would modify the generated callback to appear like the one shown in Figure 5-3. You can create an image with bitmap or IconEdit (make sure that the format is set to X Bitmap).

```
. . .
/* Put this line at top of  */
#include “mylabel.xbm”
/* _stubs.c file  * /
. . .
Widget
create_button(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
Widget button;
XImage *label_image;
Display *dpy;

      button = GolitFetchWidget(name, wdp,
                      parent, args, num_args,closure);
      dpy = XtDisplay(button);
      visual = OlVisualOfObject(button);
      label_image = XCreateImage( dpy, /* display */
                           visual, /* visual*/
                           1,   /* depth*/
                           XYBitmap, /* format*/
                           0,   /* offset*/
                           mylabel_bits, /* bitmap data */
                           mylabel_width, /* width*/
                           mylabel_height, /* height */
                           8,   /* bitmap_pad*/
                           0    /* bytes_per_line */
                      );
      XtVaSetValues(button,
           XtNlabelType, (XtArgVal)OL_IMAGE,
           XtNlabelImage, (XtArgVal)label_image,
           NULL);
      return button;
}
```

Figure 5-3 Button Create Proc Modified to Display Icon Label

Internationalization 6

This chapter describes how to internationalize your Golit applications.

Overview of Internationalization Concepts

Internationalization is the process of making software portable between languages or *locales*. An internationalized application runs in any locale without changes to the binary. Text strings and other locale-specific information is kept separate from application code in files which can be easily edited.

Localization is the process of adapting software for specific locales. It consists of translating the application's text strings and changing other locale-specific information for a locale. Internationalization is usually performed by the software writer; Localization is usually performed by experts familiar with the specific language or region,

Levels of Internationalization

There are currently four levels of internationalization. The requirements of each level are described in the sections below.

Level 1—Text and Codesets

Level 1-compliant software is "8-bit clean" and therefore can use the ISO 8859-1 (also called ISO Latin-1) codeset. The ASCII character set uses only 7 bits out of an 8-bit byte. The ISO Latin-1 codeset requires all 8 bits for each character.

Level 2—Formats and Collation

Many different formats are used throughout the world to represent date, time, currency, numbers, and units. Also, some alphabets have more letters than others and the sorting order may vary from one language to another. Level 2-compliant programs leave the format design and sorting order to the localizer in a particular country.

Level 3—Messages and Text Presentation

Text visible to the user on-screen must be easily translatable. This includes help text, error messages, property sheets, buttons, text on icons, and so forth. To assist localizers, text strings can be culled into a separate file, where they are translated. Because the text strings are sorted individually, level 3-compliant software does not contain compound messages—those created with separate `printf` statements, for example—because the separate parts of the message will not be kept together.

Level 4—Asian Language Support

Asian languages contain many characters (1500 to 15000). These cannot all be represented in eight bits and can be laborious to generate using keyboard characters. The EUC (Extended Unix Codeset) is a multi-byte character standard that can be used to represent Asian character sets. EUC does not support 8-bit codesets such as ISO Latin-1.

Golit Support for Internationalization

The current version of Golit supports Level 3 internationalization. Golit makes it easy to internationalize your application by writing out locale-specific resources to a resource file. In some cases, you may need to reposition widgets to accommodate different label lengths. The localizer only needs to edit this file to localize your application.

Generating Code for an Internationalized Application

Using Golit Command Line Options

To generate code for an internationalized application, use one of the following options when you run Golit on the GIL or project file for the application:

- `-i`, which writes all Level 3 internationalization-specific resources into a resource file. For a list of these resources, see Level-3 Resources on page 86.
- `-r`, which writes *all* resources into a resource file

When you run Golit on an individual file, the generated resource file is named `<GIL_filename>.resource`. When you run Golit on a project file, the generated resource file is named `<project_filname>.resource`.

For example, to generate code for a GIL file called `myapp.G`, you type:

```
% golit -i myapp
```

This command generates a resource file called `myapp.resource`.

Generated Resource Files

The resource files Golit generates consist of resource specifications for each widget in the application. The specification for simple widgets has the following form:

```
*<WidgetName>.<ResourceName>: <Value>
```

The specification for composite widgets, such as TextEdit and ScrolledWindow, has the following form:

```
*<WidgetName>.<SubWidgetClass>.<ResourceName>: <Value>
```

To find the name for a widget, see Getting the Right Widget Instance Name on page 63.

Level-3 Resources

The sections below list the Level-3 Internationalization-specific resources. For a full description of these resources, see the *OLIT 3.x Reference Manual*.

Common Resources

The following are the Level-3 Internationalization-specific resources common to all widgets:

- XtNx
- XtNy
- XtNlabel
- XtNtitle
- XtNstring
- XtNminLabel
- XtNmaxLabel

PopupWindowShell Resources

The following are the Level-3 Internationalization-specific resources that belong to the PopupWindowShell:

- XtNmenuTitle
- XtNapplyLabel
- XtNsetDefaultsLabel
- XtNresetLabel
- XtNresetFactoryLabel
- XtNapplyMnemonic
- XtNsetDefaultsMnemonic
- XtNresetMnemonic

- XtNresetFactoryMnemonic

TextEdit Resources

The following are the Level-3 Internationalization-specific resources that belong to the TextEdit widget:

- XtNmenuTitle
- XtNundoLabel
- XtNcutLabel
- XtNcopyLabel
- XtNpasteLabel
- XtNdeleteLabel
- XtNundoMnemonic
- XtNcutMnemonic
- XtNcopyMnemonic,
- XtNpasteMnemonic
- XtNdeleteMnemonic

Scrollbar Resources

The following are the Level-3 Internationalization-specific resources that belong to the Scrollbar widget:

- XtNmenuTitle
- XtNhereToTopLabel
- XtNtopToHereLabel
- XtNhereToLeftLabel
- XtNleftToHereLabel
- XtNpreviousLabel
- XtNhereToTopMnemonic
- XtNtopToHereMnemonic
- XtNhereToLeftMnemonic
- XtNleftToHereMnemonic
- XtNpreviousMnemonic

Using XFILESEARCHPATH

The `XFILESEARCHPATH` environment variable in conjunction with the `LANG` environment variable helps applications automatically set up locale-specific resource files. The default value of this variable collapses to : `/usr/lib/X11/$LANG/app-defaults/<Class>`, where `Class` is the class of an application. In case of Golit-generated applications, the class-name is the name of the application with the first letter capitalized. Thus one would install the suitably localized resource file so that `XFILESEARCHPATH` points to it. Refer to the Xt Intrinsics documentation for more information on specifying resources and installing resource files.

Files Shipped with Golit

A

The installation medium you receive with the Devguide package includes a set of files useful to Devguide users. These files are installed on your workstation or server if you have followed the instructions in the *Software Developer Kit Installation Guide*. You will find the files in separate subdirectories in Devguide's home directory.

The Bin Subdirectory

The `bin` subdirectory contains the executable files for Devguide and Golit (as well as for GNT and GXV).

The Demo Subdirectory

The `demo` subdirectory contains three subdirectories; one for each code generator.

The Include Subdirectory

The `include` subdirectory contains header files used by Golit and the other code generators. The files used by Golit are:

- `libgolit.h`: prototypes of the libgolit functions that execute widget creation.
- `libgolitI.h`: definitions of macros used in the `_ui.c` file.

- `Group.h`: definition of GroupWidget that implements groups in Devguide.

The Lib Subdirectory

The `lib` subdirectory contains the `libgolit.a` runtime library and libraries containing functions for the other code generators. It includes the Group widget that Golit uses to support Devguide group features.

The `lib` subdirectory also contains the `OLIT .config` file. This file is used to configure the Action-Events Connections window for Golit. It lists the events, actions, and targets that are permitted in Golit for each possible source object.

The Man Subdirectory

The `man` subdirectory contains man pages for Golit. To see them when you use the `man` command, you can copy them into the directory containing other man pages, or you can set the MANPATH variable to look in this directory when you use the man command. For more information, consult the Man Page Specification.

The Src Subdirectory

The `src` subdirectory contains source code for the libraries included in the `lib` subdirectory. This source code is supplied "as is" and is not supported by Sun Microsystems.

The Doc Subdirectory

The `doc` subdirectory contains miscellaneous documentation about Devguide.

Unsupported Devguide Features B

This appendix lists Devguide features that the Golit code generator does not support.

Bold Labels

The OPEN LOOK specification calls for the labels of all control objects to be bold. When you prototype an interface in Devguide, the label font will appear bold. However, the default font weight for labels in Golit-generated interfaces is normal. When you run your Golit program, the labels will not be bold.

You can use Create Procs to change the font to bold. Chapter 4, “Golit Functionality in Detail provides an example of how to do this. However, it is normally preferable to set the caption font resource on the command line in the following manner:

```
% my_application -xrm ‘*Caption.font: lucidasans-bold’
```

where `my_application` is the name of the application for which you want the labels to appear bold. You can achieve the same effect by editing the resource file, setting the fallback resources in the `_stubs.c` (or `<project>.c`) file, or application defaults file.

Help

Golit does not support Help as implemented in Devguide. To add help for a widget, you must use a Create Proc and OlRegisterHelp(). For more information, see Chapter 4, “Golit Functionality in Detail.”

Connections Between Base Windows

Golit will generate an error message if your interface has connections between separate Base Windows.

Miscellaneous Unsupported GUI Elements

The following are user interface elements that are available in Devguide but are currently not supported by Golit.

- Glyph types in labels
  Chapter 5, “Advanced Topics” provides an example of how you can use Create Procs to display any bitmap image in a label.
- Icons for Base Windows
  Chapter 5, “Advanced Topics” provides an example of how you can use Create Procs to assign an icon to your application
- Term Panes
- Numeric TextFields
- Drag and Drop from a Canvas or a Scrolling List
- List items
  The scrolling list itself is supported. However, Golit omits any list items that you inserted in Devguide. You must write your own code to insert list items.
- Setting stack
  Golit provides limited support for this object. The current setting is not displayed next to the abbreviated menu button.
- Layered panes in popup windows
  If your interface has layered panes in a popup, the panes will be tiled in your compiled application. Layered panes do work in base windows.

- Slider end values
  Although you can include slider end-values in your interface prototype in Devguide, they will not appear in your compiled application.

Golit simply ignores most of these unsupported elements when it reads the GIL file. However, if you attempt to use Golit to generate code for an interface that includes term panes or numeric TextFields, Golit will abort code generation and display an error message.

libgolit Library Function Reference C

This appendix lists the libgolit library functions and their usage.

GolitNameToWidget()

Retrieves the widget id for the specified widget.

```
Widget
GolitNameToWidget (Widget                          root,
                     String                        name);
```

Where:

`root`	The widget in the widget tree where you would like to begin searching for the widget of the specified name. Normally, you can just use the widget that is passed as one of the callback parameters.
`name`	The instance name of the widget for which you want to get an id.

GolitFetchLastWidgetID()

Retrieves the widget id for the last widget of the specified widget description pointer created.

```
Widget
GolitFetchLastWidgetID (GolitWidgetDescPtrwdp);
```

Where:

`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See “Getting the Right Widget Instance Name” on page 63 for instructions on finding widget description pointer names.

GolitMergeArgLists()

This function merges args that you provide with the existing args in a widget description.

```
XtVarArgsList
GolitMergeArgLists (GolitWidgetDescPtr          wdp,
                    ArgList                     override_args,
                    Cardinal                       num_args);
```

Where:

`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See “Getting the Right Widget Instance Name” on page 63 for instructions on finding widget description pointer names.
`override_args`	a pointer to the list of args that you want to merge in.
`num_args`	The number of args in ovveride_args.

GolitFetchWidget()

Instantiates the specified widget. It does not instantiate any children or popups.

```
Widget
GolitFetchWidget (String                        name,
                    GolitWidgetDescPtr,         wdp,
                    Widget                      parent,
                    ArgList                     args,
                    Cardinal                    num_args,
                    XtPointer                   closure);
```

Where:

`name`	The name of the widget that you want to fetch.
`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See “Getting the Right Widget Instance Name” on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget you want to fetch.
`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree.

GolitFetchWidgetUnmanaged()

This function instantiates the specified widget, but does not add it to its parent’s managed set. It does not instantiate any children or popups.

```
Widget
GolitFetchWidgetUnmanaged (String                name,
                     GolitWidgetDescPtr,         wdp,
                     Widget                      parent,
                     ArgList                     args,
                     Cardinal                    num_args,
                     XtPointer                   closure);
```

Where:

`name`	The name of the widget that you want to fetch.

`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See "Getting the Right Widget Instance Name" on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget you want to fetch.
`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree

GolitFetchChildren()

Instantiates the children of the specified widget. It does not instantiate popups.

```
Widget
GolitFetchChildren (String                     name,
                    GolitWidgetDescPtr,        wdp,
                    Widget                     parent,
                    ArgList                    args,
                    Cardinal                   num_args,
                    XtPointer                  closure);
```

Where:

`name`	The name of the widget whose children you want to fetch.
`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See "Getting the Right Widget Instance Name" on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget whose children you want to fetch.

`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree

GolitFetchChildrenUnmanaged()

Instantiates the children of the specified widget, but does not add them to the widget's managed set. It does not instantiate popups.

```
Widget
GolitFetchChildrenUnmanaged (String             name,
                   GolitWidgetDescPtr,          wdp,
                   Widget                       parent,
                   ArgList                      args,
                   Cardinal                     num_args,
                   XtPointer                    closure);
```

Where:

`name`	The name of the widget whose children you want to fetch.
`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See "Getting the Right Widget Instance Name" on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget whose children you want to fetch.
`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree

GolitFetchPopups()

Instantiates the popups of the specified widget and the children of those popups.

```
Widget
GolitFetchPopups (String                        name,
                   GolitWidgetDescPtr,          wdp,
                   Widget                       parent,
                   ArgList                      args,
                   Cardinal                     num_args,
                   XtPointer                    closure);
```

Where:

`name`	The name of the widget whose popups you want to fetch.
`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See "Getting the Right Widget Instance Name" on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget whose popups you want to fetch.
`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree

GolitFetchPopupsUnmanaged()

Instantiates the popups of the specified widget and the children of those popups. It does not add the popups to the widget's managed set.

```
Widget
GolitFetchPopupsUnmanaged (String               name,
                   GolitWidgetDescPtr,          wdp,
                   Widget                       parent,
                   ArgList                     args,
                   Cardinal                    num_args,
                   XtPointer                   closure);
```

Where:

`name`	The name of the widget whose popups you want to fetch.
`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See "Getting the Right Widget Instance Name" on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget whose popups you want to fetch.
`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	

GolitFetchWidgetHier()

Instantiates the specified widget, its popups, and its children

```
Widget
GolitFetchWidgetHier (String                    name,
                    GolitWidgetDescPtr,         wdp,
                    Widget                      parent,
                    ArgList                     args,
                    Cardinal                    num_args,
                    XtPointer                   closure);
```

Where:

name	The name of the widget whose hierarchy you want to fetch.

`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See “Getting the Right Widget Instance Name” on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget whose hierarchy you want to fetch.
`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree

GolitFetchWidgetHierUnmanaged()

Instantiates the specified widget, its popups, and its children but does not add the widget to its parent’s managed set.

```
Widget
GolitFetchWidgetHierUnmanaged (String           name,
                    GolitWidgetDescPtr,         wdp,
                    Widget                      parent,
                    ArgList                     args,
                    Cardinal                    num_args,
                    XtPointer                   closure);
```

Where:

`name`	The name of the widget whose hierarchy you want to fetch.
`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See “Getting the Right Widget Instance Name” on page 63 for instructions on finding widget description pointer names.
`parent`	The id of the parent of the widget whose hierarchy you want to fetch.

`args`	Pointer to an Arg structure for the widget you want to fetch
`num_args`	The number of arg in args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree

GolitFetchShellHier()

Instantiates a shell widget and all its children. Use this only for base windows.

```
Widget
GolitFetchShellHier (String                    app_name,
                     String                    app_class,
                     GolitWidgetDescPtr,       wdp,
                     Display                   *dpy,
                     ArgList                   args,
                     Cardinal                  num_args,
                     XtPointer                 closure);
```

Where:

`app_name`	The base window's name.
`app_class`	The application's class.
`wdp`	The widget description pointer. This is based on the name of an object that you set up in Devguide. See "Getting the Right Widget Instance Name" on page 63 for instructions on finding widget description pointer names.
`dpy`	Display
`args`	Resources.
`num_args`	Number of args.
`closure`	Pointer to an application data structure that you want to pass down the widget tree

Index

Symbols

.c, 12
`.c` file, 59
.h, 12
.make, 12
_stubs.c, 2, 11, 14, 60, 61
    callback templates, 57
    callbacks, 57
    merging, 51
_stubs.c.BAK, 14, 60, 61
_stubs.c.delta, 14, 60, 61
_ui.c, 2, 11
    description, 54
_ui.h, 2, 11
    description, 56, 57

A

about this book
    font conventions, xvi
    prerequisite reading, xiii
Action Menu items, 49
Adjust, 48, 49
AnyEvent, 48, 49
ArgList, 54

B

background reading, xv
base window
    creating, 24
bin subdirectory, 89
building
    the interface, 23

C

C declarations, 59
callbacks, 57
    arguments, 58
    removing, 61
    setting up in Devguide, 60
    templates, 57
callData (callback argument), 58
CallFunction, 35, 48, 49
    templates, 58
clientData (callback argument), 58
command line options, 51
compiling source code, 12, 46, 76
connections
    Action menu items, 49
    CallFunction, 35
    changing actions for, 60
    changing names of, 61

removing, 61
setting up in Devguide, 7, 16, 35, 47, 60
When menu items, 48
Connections Manager, 7, 16, 47
control areas
layered, 37
Create, 48
create procs, 3, 70
setting up in Devguide, 50
using for glyphs, 81
using for icons, 79
creating
base windows, 24
the interface, 23

D

demo subdirectory, 89
Destroy, 48, 49
Devguide
background on, 1
configuring for GOLIT, 6
designing interfaces for GOLIT, 6
features not supported by GOLIT, 91
files included with, 89
installing, 2
interacting with, 2
meaning of name, 2
setting up create procs, 50
test mode, 39
Disable, 49
doc subdirectory, 90
documentation, 90
Done, 48, 49
DoubleClick, 48, 49
drag and drop, 92
drop sites, 92
DroppedUpon, 48, 49

E

Enable, 49
Enter, 48, 49
ExecuteCode, 48, 49
Exit, 48, 49

F

fallback resources, 57
files, 52, 59
fonts, using Create Procs to set, 73
functions
common to more than one .G file, 59

G

generated files, 3, 11, 52
_stubs.c, 57
_stubs.c.BAK, 60
_stubs.c.delta, 60
_ui.c, 54
Makefile, 58
overview, 53
project_name.c, 59
project_name.C.BAK, 61
project_name.c.BAK
project_name.c.BAK, 61
project_name.c.delta, 61
project_name.h, 59
project_name.h.BAK, 61
project_name.make, 60
ui.h, 56
generating code, 40
GetLabel, 49
GetValueNumber, 50
GIL (Graphical Interface Language) files, 47
glyphs, 92
using for labels, 81
GOLIT
background on, 3
definition of, 1
generated files, 52
generated files overview, 53
getting started, 5
how to use it, 50
interacting with, 2

golit (command), 8
  options, 51
  using options
    -? (help), 51
    -n (nomerge), 51
golit command
  using options
    -s (silent), 52
`golit` command
  using options
    `-m (-main)`, 51
    `-p`, 51
golit(command)
  using options
    -i, 51
    -k (kandr), 51
    -r (resources), 52
GolitFetchLastWidgetID(), 62
GolitFetchWidgetHier(), 67
GolitNameToWidget(), 13, 62
GUIDEHOME, 2, 77

H

Help, 92
help, 92
  using Create Procs to register, 70
Hide, 50
home directories, 2, 77
how to use GOLIT, 50

I

icons, 92
  using create procs to specify, 79
If, 60
include subdirectory, 89
included files, 89
instance name, 63
instantiating widgets, 66
integrating interface and application code, 61
interface
  building, 23
internationalization, 92

K

KeyPress, 49
Keypress, 48

L

label images, 81
labels
  fonts, 91
layered panes, 37
lib subdirectory, 90
libgolit functions, 66
libgolit.h, 89
libgolitI.h, 89
libraries, 90
LoadTextfile, 50

M

`main()` function, 59
Makefile, 11, 58
  parameters, 58
man pages, 90
man subdirectory, 90
MANPATH, 90
manual
  GOLIT Reference, xiv
  libgolit reference, xiv
  Unsupported Devguide features, xiv
Menu, 48, 49

N

Notify, 48, 49
numeric text fields, 92

O

OPEN LOOK, 3
  recommendations, 23

OPEN LOOK Style Guide, 23
OpenWindows Developer's Guide
  meaning of name, 2
OPENWINHOME, 2, 77
options (command line)
  ? (help), 51
  m (main), 51
  -n (no merge), 51
  -p (project), 51
  -s (silent), 52

P

Popdown, 48, 49
Popup, 48, 49
printf(), 58, 61
PROGRAM parameter, 58
project files, 59
project_name.c, 59
project_name.C.BAK, 61
project_name.c.delta, 61
project_name.h, 59
project_name.h.BAK, 61
project_name.make, 60

R

recommended reading, xv
references, xv
regenerating user interface code, 14
Repaint, 48, 49
Resize, 48, 49
resize corners, 33
resources
  fallback, 57
  setting programmatically, 62

S

scrolling lists, 92
Select, 48, 49
SetLabel, 50
setting stack, 92
SetValueNumber, 50
Show, 50
siblings, 54, 55
source code
  compiling, 46, 76
SOURCES.c parameter, 58
SOURCES.G parameter, 59
SOURCES.h parameter, 58
src subdirectory, 90
suffixes
  .BAK, 60
  .c, 2
  .delta, 60
  .G, 2
  .h, 2
  .P, 5
  make, 12

T

term panes, 92
test mode, 39
text fields, 63
  numeric, 92
  setting to read only, 81

U

Unselect, 48, 49
user interface (UI) code, 2
  regenerating, 14, 60
user's manual
  getting started, xiv
  GOLIT reference, xiv
  libgolit reference, xiv
  Unsupported Devguide features, xiv

W

When, 48
  menu items, 48
widget description pointer, 54, 63
widget descriptor, 54, 63
widget hierarchy, 54

widget tree
    instantiating parts of, 66
widgets
    getting ids for, 13, 62
    GOLIT names for, 54, 63
    setting resources, 62
    siblings, 54, 55

X

XtNconsumeEvent, 49
XtNdestroyCallback, 49
XtNdndTriggerCallback, 49
XtNpopdownCallback, 49
XtNpopupCallback, 49
XtNresizeCallback, 49
XtNselect, 49
XtNsliderMover, 49
XtNunselect, 49
XtNverification, 49