OpenWindows Developer's Guide: Motif Conversion Utilities Guide

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

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Preface. xi

1. Introduction to GMF and GUIL 1

Devguide, GMF, and GUIL. 1

Names and Terminology. 1

Installation and Environment Setting for Devguide, GMF, and GUIL 2

Setting Environment Variables for GUIDEUTILHOME . . . 2

Setting Environment Variables for MOTIFHOME. 2

How you interact with Devguide and GMF. 3

2. Getting Started with GMF. 5

Summary of How to Create an Application with GMF 6

Generating User Interface Code from GIL Files. 6

GMF-Generated Files 7

Compiling and Testing Interface Code 8

Integrating Application Code with GMF Interface Code 8

Regenerating Code for a Modified Interface 9

3. GMF Functionality in Detail. 11

When Menu Items 11

Action Menu Items 12

Using GMF to Generate Code 12

GMF Command-Line Options 13

Files Generated for a Single GIL File. 14

_ui.c File 15

_ui.h File 17

_stubs.c. 18

Makefile 19

Files Generated for a Project. 19

<project_name>.c File 19

<project_name>.h File. 20

<project_name>.make File 20

Regenerating Code for a Modified Interface 20

Regenerating Code for an Individual GIL File. 21

Regenerating Code for a Project 21

Removing Obsolete Callbacks 21

Integrating Your Application Code with the Interface Code . . 22

Changing Widget Resources 22

Compiling 25

4. Internationalization 27

Overview of Internationalization Concepts 27

Levels of Internationalization 27

GMF Support for Internationalization 29
Generating Code for an Internationalized Application 29
Using GMF Command Line Options 29
Generated Resource Files 29
Level-3 Resources 30
Common Resources 30
Using XFILESEARCHPATH 30
5. Using GUIL to Create a UIL File 31
Generating a UIL File 31
GUIL Command-Line Options 32
A. Files Shipped with Devguide Motif Utilities 33
The Bin Subdirectory 33
The Include Subdirectory 33
The Lib Subdirectory 33
The Man Subdirectory 34
The Src Subdirectory 34
The demo Subdirectory 34
B. Unsupported Devguide Features 35
Bold Labels 35
Help 35
Connections Between Base Windows 35
Miscellaneous Unsupported GUI Elements 36

Figures

Figure 1-1 GMF-Generated Source Files 3

Figure 3-1 Overview of gmf Files for a Single GIL File 15

Figure 3-2 Example Widget Tree for a Base Window 17

Tables

Table 3-1 When Items and the Resources Generated for Them. 11

Table 3-2 Devguide Panes & Windows and their gmf Widget Hierarchies 23

Table 3-3 Devguide Control Objects and their gmf Widget Hierarchies 24

Preface

This manual describes how to use the GMF or GUIL utilities to generate Motif® C code or User Interface Language (UIL) files from Devguide GIL files.

Note – `gmf` and `guil` are for one-time conversion of a previously-generated GIL file to Motif C code or to UIL; because Devguide is an OPEN LOOK application and the conversion to the Motif look and feel is not always perfect (size and alignment of objects may not be correct, for instance), it is unlikely that you will want to create a new user interface with Devguide for a Motif application, nor is it likely that you will want to modify your Devguide interface after once running `gmf` or `guil`.

Who Should Use This Book

This manual is for applications programmers who have an existing Devguide user interface; they can use `gmf` to generate Motif C code and integrate this code with their own application code or use `guil` to generate UIL files.

Before You Read This Book

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

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

- *Solaris 2.4 Introduction*
- *Software Developer Kit Installation Guide*

How This Book Is Organized

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

Chapter 1, "Introduction to GMF and GUIL" provides overviews of the GMF and GUIL utilities.

Chapter 2, "Getting Started with GMF" gets you acquainted with how GMF 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, "GMF Functionality in Detail" describes GMF, the files it generates, and the `libgmf` library routines. If you want to write your own widget creation routines, you should read this chapter.

Chapter 4, "Internationalization" describes how to internationalize your GMF application.

Chapter 5, "Using GUIL to Create a UIL File" describes how to use GUIL to produce User Interface Language (UIL) files from a Devguide GIL file.

Appendix A, "Files Shipped with Devguide Motif Utilities" describes the Devguide Motif Utilities directory structure and lists the files there. The demo subdirectory contains a sample program that shows you how to integrate a UIL file into a working application.

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

The **Index** lists references to important names and topics in alphabetic order.

Related Books

This manual is a supplement to the OpenWindows Developer's Guide Set, which contains the following manuals:

- *OpenWindows Developer's Guide: User's Guide*

- *OpenWindows Developer's Guide: XView Code Generator Programmer's Guide*
- *OpenWindows Developer's Guide: OLIT Code Generator Programmer's Guide*

Note that the *OpenWindows Developer's Guide: User's Guide* includes Appendix D, "Devguide 3.0.1 Release Notes"; new features, problems, and bugs are described in this appendix.

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

- *Xt Intrinsics Reference Manual*, O'Reilly & Associates, Inc., 1991
- *OSF/Motif Reference Manual*, PTR Prentice Hall, Inc., 1993
- *OSF/Motif Programmer's Manual*, PTR Prentice Hall, Inc., 1993
- *OSF/Motif Style Guide*, PTR Prentice Hall, Inc., 1993

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:

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Examples
%	UNIX C shell prompt	% *or* system%
$	UNIX Bourne shell prompt	$ *or* system$
#	Superuser prompt, either shell	# *or* system#

Introduction to GMF and GUIL 1

This chapter provides an overview of the GIL-to-Motif (`gmf`) and GIL-to UIL (`guil`) utilities (GIL is the Guide Interface Language output of Devguide).

Devguide, GMF, and GUIL

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

Devguide produces GIL files that you can convert into Motif C code with the utility, `gmf`, or into UIL with the utility, `guil.`

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 *gmf* refers to the utility that takes the GIL file produced by Devguide to produce Motif source code for your user interface. This code is referred to as the *UI* (user interface) code, which contrasts with the *application code.*

Installation and Environment Setting for Devguide, GMF, and GUIL

To install Devguide and the `gmf` and `guil` utilities, follow the instructions in the *Software Developer Kit Installation Guide*. See the *OpenWindows Developer's Guide: User's Guide* for a more complete discussion of the process of setting environment variables.

Setting Environment Variables for GUIDEUTILHOME

By default, the `SUNWgmfu` package is installed in the directory, `/opt/SUNWgmfu`. Wherever `SUNWgmfu` is installed, the GUIDEUTILHOME directory should be set to the directory where the package is stored. For example, if the package were installed in `/export/home/SUNWgmfu`, then:

for `C` shell:

```
% setenv GUIDEUTILHOME /export/home/SUNWgmfu
```

for Bourne shell:

```
$ GUIDEUTILHOME=/export/home/SUNWgmfu
$ export GUIDEUTILHOME
```

If you want to access `gmf` and `guil` from anywhere, `GUIDEUTILHOME` needs to be set and the `PATH` environment variable needs to have `GUIDEUTILHOME/bin` in it.

For access to the `gmf` and `guil` man pages, `MANPATH` needs to have `GUIDEUTILHOME/man` in it.

Setting Environment Variables for MOTIFHOME

In addition to setting the variable for GUIDEUTILHOME, you should set MOTIFHOME to the location where Motif is installed (`/usr/opt/SUNWmotif`, for example).

If you are using a non-Sun version of Motif, you may have to edit the Makefile to change `-I$MOTIFHOME/include` to `-I$MOTIFHOME/usr/include` and to change `-L$MOTIFHOME/lib` to `-L$MOTIFHOME/usr/lib`.

How you interact with Devguide and GMF

The basic interaction with Devguide is very simple. When 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.*

Note – To see the interface with the Motif look and feel, you have to compile the generated code.

To create Motif C code from a GIL file, run the `gmf` utility on the GIL file, shown below as `<fn>.G`. The `.G` suffix is the standard suffix for GIL files. `gmf` produces the four files shown in Figure 1-1.

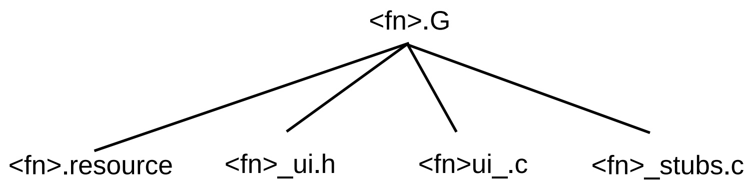

Figure 1-1 GMF-Generated Source Files

`gmf` also automatically generates a program Makefile.

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

Note – `gmf`-generated code is identical to handwritten Motif code; you can use `gmf`'s output to learn Motif.

Getting Started with GMF 2

This chapter describes how to generate simple user interfaces and integrate them with your applications. This chapter is intended to be a "quick start guide" that provides you with the minimal instructions necessary to start using `gmf`. The topics discussed here are covered in greater detail in Chapter 3, "GMF Functionality in Detail."

Note – Although this chapter discusses the creation of a Devguide user interface from scratch, remember that Devguide produces Open Look output and it is unlikely that you will want to use Devguide to create a user interface if the final application will be a Motif application. If, despite this warning, you decide to create a new Devguide interface for a Motif application, read the *OpenWindows Developer's Guide: User's Guide*, setting the Toolkit to OLIT if you plan on doing connections (note, however, that some connections may not work for a Motif application).

If you already have a Devguide-created GIL file, you can use `gmf` to convert the GIL file to Motif C code.

Summary of How to Create an Application with GMF

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

1. **Generate Devguide UI code by running `gmf` on a GIL or project file.**
   `gmf` 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 `gmf` on a project file, it generates two additional C files and a supplement to the makefile.

2. **Use `make` to build a test copy of the interface.**
   `gmf` 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.

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

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

Generating User Interface Code from GIL Files

To generate code from a GIL file, run `gmf` as follows: :

```
% gmf <GIL_filename>
```

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

```
% gmf -p <project_filename>
```

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

GMF-Generated Files

gmf-generated files contain all the source code necessary to create an executable that uses the Motif toolkit and the libgmf runtime library.

When you run gmf 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
- <GIL_filename>.resource - resource file created by -r (all resources) or -i (internationalization) flags.

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

When you run gmf 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 `gmf` to generate source code for a project file named `myproject.P`, `gmf` creates the files `myproject.c`, `myproject.h`, and `myproject.make`.

Compiling and Testing Interface Code

To compile the interface code generated by `gmf`, type the following in the appropriate directory after you have generated the code:

```
% make
```

The `make` utility compiles and links the `gmf`-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 `gmf` do not work in the Devguide test mode. Also, since Devguide displays an OPEN LOOK look and feel, the interface will look different in the test mode than it will in the compiled application. 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 GMF Interface Code

To add your application code to the `gmf` interface code, 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. Use XtVaSetValues or XtVaGetValues to set or get resources, as you normally would.

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 `gmf`.

Regenerating Code for a Modified Interface

As noted above, it is unlikely that you will want to modify your Devguide interface after running `gmf`; if you do, however, `gmf` will do the following each time it is run:

- Overwrite the existing `_ui.c` and `_ui.h` files.
- Back up the current `_stubs.c file` to `_stubs.c.BAK`.
- Generate a new `_stubs.c` file that contains templates for all the connections specified in the GIL file.
- Merge any code that you inserted in the original `_stubs.c` into the new `_stubs.c`.
- Create 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*, `gmf` will perform the steps above for each of the GIL files in the project. It will also do the following:

- Back up the `<projectname>.c` and `<project_name>.h` files to `.BAK` files.
- Generate a new `<projectname>.c` file and merges it with the old one, listing the changes in a `.delta` file.
- Overwrite the original `<project_name>.h` file.

GMF Functionality in Detail 3

This chapter explains in detail how to use `gmf` 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 libgmf functions.

Note – Although this chapter discusses the creation of a Devguide user interface from scratch, remember that Devguide produces Open Look output and it is unlikely that you will want to use Devguide to create a user interface if the final application will be a Motif application. If you already have a Devguide-created GIL file, you may wish to use `gmf` to convert the GIL file to Motif C code.

When Menu Items

Note that gmf generates the same resource for many of the When items; it is up to the application to distinguish between them. Table 3-1 below lists the resource generated for each When item.

Table 3-1 When Items and the Resources Generated for Them

When Item	Resource
Destroy	XmNdestroyCallback
Done	XmNpopdownCallback
Notify	XmNvalueChangedCallback or XmNactivateCallback

Table 3-1 When Items and the Resources Generated for Them

Popdown	XmNpopdownCallback
Popup	XmNpopupCallback
Repaint	XmNexposeCallback
Resize	XmNresizeCallback
Unselect	XmNdisarmCallback

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

Using GMF to Generate Code

`gmf` 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 `gmf` command is as follows:

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

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

GMF Command-Line Options

`gmf` provides the following command-line options:

-h (-help)

Displays a message describing `gmf` usage and options.

-k or -kandr

Writes K&R C code that doesn’t contain function prototypes. `gmf` 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 `gmf` on a project’s `.G` files.

-n or -nomerge

Keeps `gmf` 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 `gmf` 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 `gmf` will overwrite 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 gmf to operate silently; no messages will appear.

Files Generated for a Single GIL File

gmf 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, gmf generates the following files:

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

For example, if you use gmf to generate source code for a GIL file named newt.G, gmf generates the files newt_ui.c, newt_ui.h, and newt_stubs.c. gmf also creates a makefile under the name Makefile, if it doesn't already exist. Figure 3-1 provides an overview of the files gmf generates for a single GIL file (fn) and the other files necessary to create an executable program.

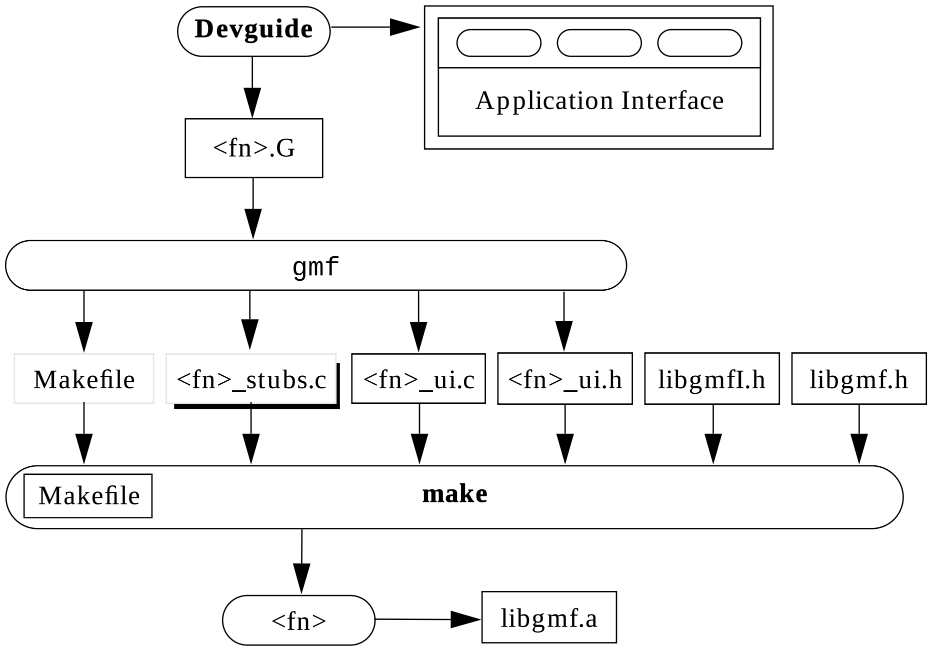

Figure 3-1 Overview of `gmf` Files for a Single GIL File

In Figure 3-1, 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

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 widgets generated by `gmf`. Many Devguide objects consist of a hierarchy of several Motif widgets. For example, a Text Field object with a label in Devguide is actually three widgets in the `gmf`-generated code: a RowColumn containing a textField widget and a label widget.

If a Devguide object comprises more than one widget, `gmf` 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`, `gmf` names the textfield widget description pointer `Mygilfile_mytextfield` and the label widget description pointer `Mygilfile_G_Label_mytextfield.`

Figure 3-2 shows an example of a widget tree created by `gmf`. 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`), `gmf` generates the widget tree depicted in Figure 3-2.

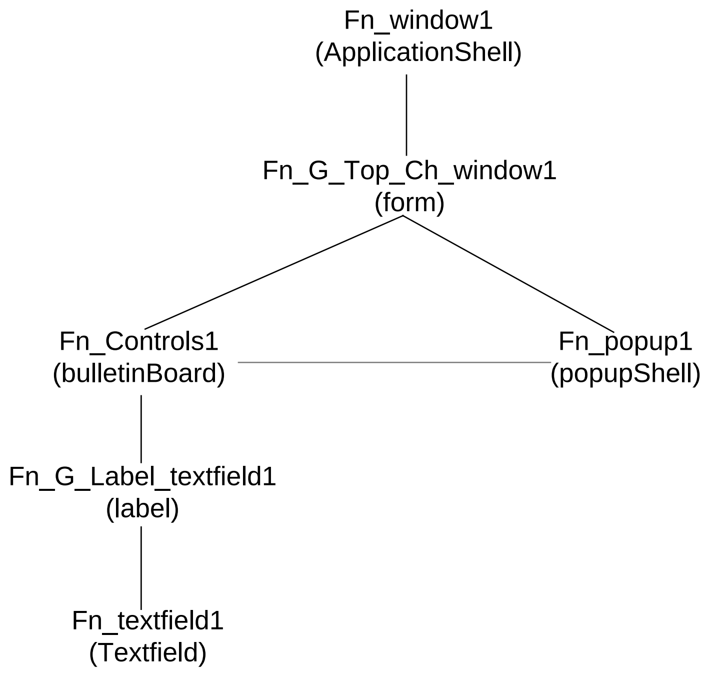

Figure 3-2 Example Widget Tree for a Base Window

Table 3-2 and Table 3-3 in "Changing Widget Resources" on page 22 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 `gmf` 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 `gmf`-generated widget descriptions available to the application programmer. At the top of every `_ui.h` file is a series of widget declarations that declare all the generated widgets available to the application programmer.

_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 Motif toolkit and sets up an event handling loop. It creates the base window and all of its children and shells for all other windows. Note that the contents of each popup window is not created until the popup is displayed.

Callbacks in the _stubs.c File

For each connection you specify in Devguide, `gmf` 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. 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 `gmf`, 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 `gmf` 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 `gmf` 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 `gmf` on a project file, `gmf` 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, `gmf` 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`. `gmf` 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 `gmf` on the GIL or project file. If you want to re-run `gmf` and immediately proceed to compile the generated code in one step, just type `make`.

When you re-run `gmf`, 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 `gmf` on a single GIL file, it generates the following files:

- new _stubs.c - `gmf` 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 `gmf` 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 - `gmf` 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 - `gmf` 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 `gmf` 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
  `gmf` 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
  `gmf` 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 `gmf`-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 is essentially one thing your application code can do to the interface you designed in `gmf`:

- change widget resources

This is discussed in the subsection below.

Changing Widget Resources

Getting the Right Widget Instance Name

As discussed in "_ui.c File" on page 15, 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 Motif 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, `gmf` creates a RowColumn containing a textField widget and a label widget. If you set up a CallFunction connection for the Text Field object, the callback is only passed the textField widget (and not the label widget). Normally, this is convenient since you are more likely to want to manipulate the textField widget. If you want to manipulate the label, you must find out its instance name.

It is relatively easy to figure out the instance name. For the widgets that the `gmf`-generated `main()` instantiates, the instance names are the same as the widget description pointers. `gmf` bases the widget description pointers on the names of the corresponding Devguide objects. Table 3-2 and Table 3-3 show Devguide objects and the widget hierarchy that `gmf` uses to create each of them. The Widget variable column shows the patterns which `gmf` 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, `gmf` omits the label widget from the object's hierarchy.

To demonstrate how you use Table 3-2 and Table 3-3, 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`. `gmf` creates the following widget variables (and instance names) for the Setting object:

`Fn_G_Label_setting1`	(label 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 for an interface named `Fn`, the callback would be passed the id `Fn_setting1` (the Exclusives widget).

Table 3-2 Devguide Panes & Windows and their `gmf` Widget Hierarchies

Devguide Object	**Widget Hierarchy**	**Widget Variable**
Control Area	**bulletinBoard**	Fn_<objectname>
Canvas	scrolledWindow **DrawArea**	Fn_G_Scrollwin_<objectname> <objectname>
Text Pane	scrolledWindow **textEdit**	Fn_G_Scrollwin_<objectname> <objectname>
Base Window	**applicationShell** form	Fn_<objectname> Fn_G_Toplevel_<objectname>
Popup Window	**popupShell**	Fn_<objectname>
Menu	**menuShell**	Fn_<objectname>

Table 3-3 Devguide Control Objects and their `gmf` Widget Hierarchies

Devguide Object	**Widget Hierarchy**	**Widget Description Pointer**
Scrolling List	label*	Fn_G_Label_<objectname>
	ScrollingList	Fn_<object_name>
Slider	label*	Fn_G_Label_<objectname>
	Slider	Fn_<object_name>
Gauge	label*	Fn_G_Label_<objectname>
	Gauge	Fn_<object_name>
Text Field	label*	Fn_G_Label_<objectname>
	TextField	Fn_<object_name>
Multiline Text Field	label*	Fn_G_Label_<objectname>
	scrolledWindow	Fn_G_Scrollwin_<objectname>
	Text	Fn_<object_name>
Exclusive Settings	label*	Fn_G_Label_<objectname>
	Exclusives	Fn_<objectname>
	RectButton**	Fn_<object_name>_<buttonname>
Nonexclusive Settings	label*	Fn_G_Label_<objectname>
	Nonexclusives	Fn_<objectname>
	ToggleButton**	Fn_<object_name>_<buttonname>
Checkbox Settings	label*	Fn_G_Label_<objectname>
	Exclusives	Fn_<objectname>
	ToggleButton**	Fn_<object_name>_<checkboxname>
Setting Stack	RowColumn	Fn_G_Rowcol_<objectname>
	Label	Fn_G_Label_<objectname>
	PulldownMenu**	Fn_G_Pulldown_<objectname>
Button	**OblongButton**	Fn_<objectname>
Menu Button	**MainWindow**	Fn_G_Mainwin_<objectname>
	MenuBar	Fn_G_Menubar_<objectname>
	CascadeButton	Fn_<objectname>
Message	**Label**	Fn_<objectname>

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

Compiling

After you've created an interface with Devguide, generated source code files with `gmf`, and created your own custom source code files, make sure that you are ready to compile. Make sure the environment variables GUIDEUTILHOME and MOTIFHOME are set to point to the Devguide and Motif 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 `gmf` 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.

Internationalization 4

This chapter describes how to internationalize your `gmf` 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,

`gmf` supports the `dgettext` method of internationalization; with the `-r` flag, `gmf` generates `bindtextdomain` for application initialization and `dgettext` for all strings.

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.

GMF Support for Internationalization

The current version of gmf supports Level 3 internationalization. gmf 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 GMF Command Line Options

To generate code for an internationalized application, use the -r option when you run gmf on the GIL or project file for the application:

- -r, which writes *all* resources into a resource file

When you run gmf on an individual file, the generated resource file is named <GIL_filename>.resource. When you run gmf on a project file, the generated resource file is named <project_filename>.resource.

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

```
% gmf -r myapp
```

This command generates a resource file called myapp.resource.

Generated Resource Files

The resource files gmf 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 22.

Level-3 Resources

The sections below list the Level-3 Internationalization-specific resources.

Common Resources

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

- XmNx
- XmNy
- XmNlabel
- XmNtitle
- There are other resources that are specific to widgets; see the `.resources` file generated when you run `gmf` with the `-r` flag.

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 `gmf`-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.

Using GUIL to Create a UIL File 5

This chapter describes how to create a User Interface Language (UIL) file by running GUIL on a Devguide-created GIL file. See the `demo` directory to see how to integrate a UIL file into a working prototype. More examples can also be found in `MOTIFHOME/share/src`.

Generating a UIL File

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

```
% guil [option flags] <GIL_filename>
```

where `GIL_filename` is the name of the GIL or project file for which you want to generate code.

Note – You must include the .G or .P suffix when running `guil`; the man page for `guil` states that you do not need to include the suffix.

GUIL Command-Line Options

`guil` provides the following command-line options:

-f

Forces overwrite of existing files; if you have previously run `guil` in this directory, the existing files will not be overwritten unless you include the `-f` flag.

-h

Displays a message describing `guil` usage and options.

-p project

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

-s

Instructs `guil` to operate silently; no messages will appear.

Files Shipped with Devguide Motif Utilities A

The SUNWgmfu package includes a set of files useful to Devguide Motif Utilities 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 the home directories of the Devguide Motif Utilities.

The Bin Subdirectory

The `bin` subdirectory contains the executable files for `gmf` and `guil`.

The Include Subdirectory

The `include` subdirectory contains header files used by `gmf`. The files used by `gmf` are:

- `libgmf.h`: prototypes of the libgmf functions that execute widget creation.
- `Group.h`: definition of GroupWidget that implements groups in Devguide.

The Lib Subdirectory

The `lib` subdirectory contains the `libgmf.a` runtime library. It includes the Group widget that `gmf` uses to support Devguide group features.

The Man Subdirectory

The `man` subdirectory contains man pages for `gmf`. 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 demo Subdirectory

The demo subdirectory contains a sample program that shows you how to integrate a UIL file into a working application; see the README file for instructions.

Unsupported Devguide Features B

This appendix lists Devguide features that the `gmf` 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 `gmf`-generated interfaces is normal. When you run your `gmf` program, the labels will not be bold.

Help

`gmf` does not support Help as implemented in Devguide.

Connections Between Base Windows

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

B

Miscellaneous Unsupported GUI Elements

The following are user interface elements that are available in Devguide but are currently not supported by `gmf`:

- Icons for Base Windows
- Term Panes
- Numeric TextFields
- Drag and Drop from a Canvas or a Scrolling List
- Slider end values
  Although you can include slider end-values in your interface prototype in Devguide, they will not appear in your compiled application.

`gmf` simply ignores most of these unsupported elements when it reads the GIL file. However, if you attempt to use `gmf` to generate code for an interface that includes term panes or numeric TextFields, `gmf` will convert them to text panes and text fields, respectively.

Index

Symbols

.c, 7
.c file, 19
.h, 7
.make, 7
_stubs.c, 3, 7, 9, 21
  callback templates, 18
  callbacks, 18
  merging, 13
_stubs.c.BAK, 9, 21
_stubs.c.delta, 9, 21
_ui.c, 3, 7
  description, 15
_ui.h, 3, 7
  description, 17, 18

A

about this book
  font conventions, xiii to xiv
  prerequisite reading, xii
Action Menu items, 12
Adjust, 11
AnyEvent, 11
As, 22

B

bin subdirectory, 33

C

C declarations, 20
callbacks, 18
  arguments, 18
  removing, 21
  setting up in Devguide, 20
  templates, 18
callData (callback argument), 18
CallFunction, 12
  templates, 18
clientData (callback argument), 18
command line options, 13
compiling source code, 8
connections
  Action menu items, 12
  changing actions for, 20
  changing names of, 21
  removing, 21
  setting up in Devguide, 20

D

Destroy, 11
Devguide

background on, 1
features not supported by GMF, 35
files included with, 33
interacting with, 3
meaning of name, 1
Disable, 12
Done, 11
DoubleClick, 11
drag and drop, 36
drop sites, 36
DroppedUpon, 11

E

Enable, 12
Enter, 11
ExecuteCode, 12
Exit, 11

F

files, 14, 19
functions
common to more than one `.G` file, 20

G

generated files, 3, 7, 14
_stubs.c, 18
_stubs.c.BAK, 21
_stubs.c.delta, 21
_ui.c, 15
Makefile, 19
overview, 15
project_name.c, 19
project_name.C.BAK, 21
project_name.c.BAK
project_name.c.BAK, 21
project_name.c.delta, 21
project_name.h, 20
project_name.h.BAK, 21
project_name.make, 20
ui.h, 17
GetLabel, 12
GetValueNumber, 12
GMF
generated files, 14
generated files overview, 15
getting started, 5
how to use it, 12
interacting with, 3
gmf (command)
options, 13
using options
-? (help), 13
-k (kandr), 13
-n (nomerge), 13
-r (resources), 13
gmf command
using options
-s (silent), 14
`gmf` command
using options
`-m (-main)`, 13
`-p`, 13
GUIDEHOME, 25
GUIL
how to use it, 31
guil (command)
using options
-? (help), 32
-f (force), 32
guil command
using options
-s (silent), 32
`guil` command
using options
`-p`, 32

H

Help, 35
help, 35
Hide, 12
home directories, 25
how to use GMF, 12
how to use GUIL, 31

I

icons, 36

If, 21

include subdirectory, 33

included files, 33

instance name, 23

integrating interface and application code, 22

internationalization, 35

L

labels
  fonts, 35

lib subdirectory, 33

libgmf.h, 33

libraries, 33

LoadTextfile, 12

M

`main()` function, 19

Makefile, 7, 19
  parameters, 19

man pages, 34

man subdirectory, 34

MANPATH, 34

manual
  GMF Reference, xii
  Unsupported Devguide features, xii

MOTIFHOME, 25

N

numeric text fields, 36

O

OpenWindows Developer's Guide
  meaning of name, 1

options (command line)
  ? (help), 13, 32
  m (main), 13
  -n (no merge), 13
  -p (project), 13, 32
  -s (silent), 14, 32

P

printf(), 18, 22

PROGRAM parameter, 19

project files, 19

project_name.c, 19

project_name.C.BAK, 21

project_name.c.delta, 21

project_name.h, 20

project_name.h.BAK, 21

project_name.make, 20

R

resources
  setting programmatically, 22

S

SetLabel, 12

SetValueNumber, 12

Show, 12

SOURCES.c parameter, 19

SOURCES.G parameter, 19

SOURCES.h parameter, 19

src subdirectory, 34

suffixes
  .BAK, 21
  .c, 3
  .delta, 21
  .G, 3
  .h, 3
  make, 7

T

term panes, 36

text fields, 22
  numeric, 36

U

user interface (UI) code, 1
    regenerating, 21

user's manual
    GMF reference, xii
    Unsupported Devguide features, xii

W

When, 22

widget hierarchy, 16

widgets
    GMF names for, 23
    setting resources, 22

X

XtNdestroyCallback, 11

XtNpopdownCallback, 11

XtNpopupCallback, 12

XtNresizeCallback, 12

XtNselect, 11

XtNsliderMover, 11

XtNunselect, 12

XtNverification, 11