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

3

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.

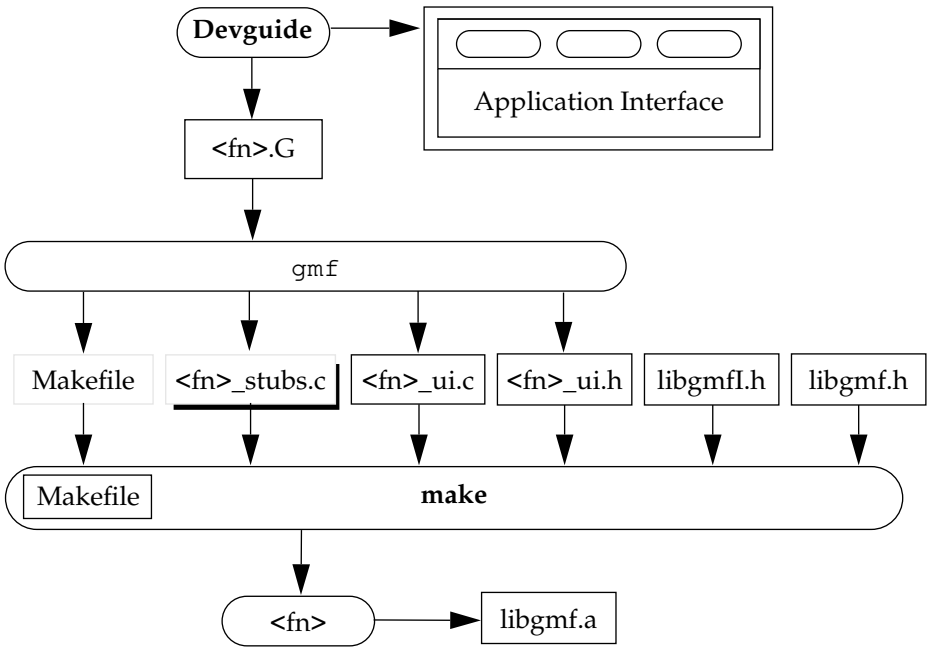

Figure 3-1 Overview of `gmf` Files for a Single GIL File

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.

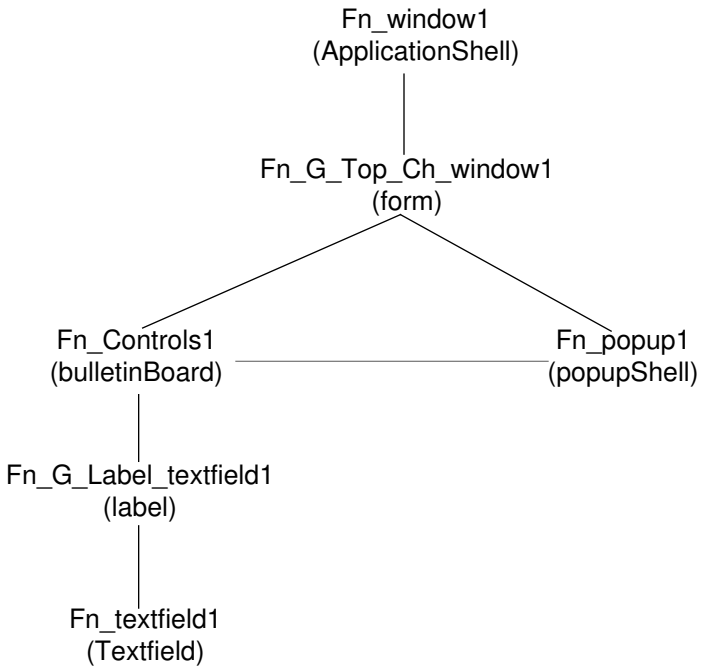

Figure 3-2 Example Widget Tree for a Base Window

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.

3

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

3