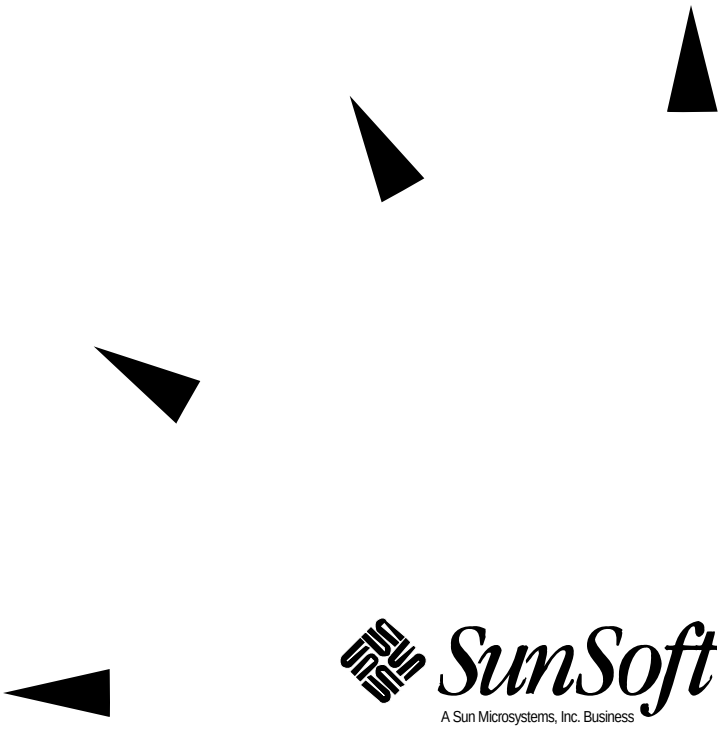
OpenWindows Developer's Guide: XView Code Generator Programmer's Guide



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

SunSoft
A Sun Microsystems, Inc. Business

Contents

Figures

Code Samples

Preface

This manual describes the XView™ toolkit code generator (GXV) for the OpenWindows™ Developer's Guide (Devguide). Use this manual to help you generate code for the user interface you build with Devguide. The Programmer's Guide includes a tutorial to introduce you to many of Devguide's features, and shows you how to modify GXV-generated files to customize the tutorial application.

Who Should Use This Book

This manual is written for applications programmers who want to generate XView toolkit code from Guide Interface Language (GIL) files. You should be familiar with your workstation's operating system, the OPEN LOOK® User Interface, Devguide, and the C programming language. To use some of the more advanced features described in this manual, you should also understand the fundamentals of creating user interfaces with the XView toolkit.

SPARC and x86 Differences

Depending on what kind of system you have (SPARC or x86), you may have a different kind of mouse and a different type of keyboard, and devices may be named differently; the differences are described below.

Two- or Three-Button Mouse

Your system may have a two- or three-button mouse. On either type of mouse, the left mouse button is referred to as SELECT and the right mouse button as MENU in this manual. To perform a SELECT function, click the left button; to perform a MENU function, click or press the right button.

On a three-button mouse, the middle button is referred to as ADJUST. To perform an ADJUST function on a two-button mouse, hold down the Shift key and click SELECT (the left button).

Keyboard Alternatives for the Meta and Other Special Keys

Some keyboards do not include the Meta key, the function keys L1 - L10, or a Help key; keyboard alternatives are available for the Meta and Help keys and most of the function keys.

Alternative to the Meta Key

On keyboards without the Meta key (marked as a diamond on most SPARC keyboards), the Alt key is mapped to the Meta key. If you have installed Devguide on an x86 system, use the key labeled “Alt” whenever the Meta key is prescribed.

Note – Functions which require the Alt key itself will *not* work on systems with keyboards that do not include a Meta key.

Alternatives to Function Keys

On keyboards without the L1 to L10 function keys, Meta-key equivalents are available for most of the functions, as shown below. Use the key labeled “Alt” instead of the Meta key on keyboards without the Meta key. The alternatives are:

Cut	Meta-x
Copy	Meta-c
Paste	Meta-v
Find	Meta-f

Props	Meta-i
Undo	Meta-z
Open	Meta-w

There are no keyboard alternatives for the Front and Again keys.

Alternative to the Help Key

On keyboards without a “Help” key, use the key marked “F1” for help.

Before You Read This Book

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*

How This Book Is Organized

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

Chapter 1, “Introduction,” provides an overview of GXV.

Chapter 2, “Getting Started with GXV,” gets you acquainted with how GXV works. It shows you how to generate code and how to compile this code with your application code. It provides a very simple example.

Chapter 3, “GXV Tutorial,” describes how to use Devguide and GXV to build a complete OpenWindows application.

Chapter 4, “GXV Functionality in Detail,” contains information on generating GXV code, describes GXV’s libraries, and provides some programming tips.

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

Appendix B, "Internationalization," provides information on internationalizing and localizing applications using Devguide and GXV.

Related Books

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

- *Solaris 2.4 Introduction*, *Solaris 2.4 System Administrator AnswerBook*, 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 GXV, the XView toolkit code generator for Devguide.

Devguide and GXV

Devguide is a development tool designed to make the interface programmer's job much easier. It gives you the ability to create and test user interfaces without writing any code.

Devguide can be used with either of two code generators to turn the GIL file that it generates into executable code.

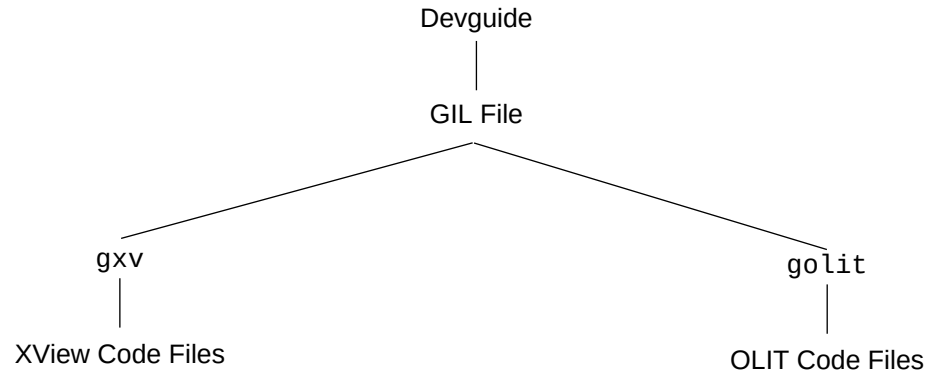


Figure 1-1 Devguide Tools and Files

1

The two code generators, GXV, and GOLIT, are described in this manual set. This manual describes the GXV code generator for the XView toolkit.

Names and Terminology

The word “Guide” in the full name “OpenWindows Developer’s Guide” is an acronym for Graphical User Interface Design Editor.

When the abbreviated name, Devguide, is used, it refers to the graphical interface tool you use to develop the user interface.

The terms *postprocessor* and *GXV* (GXV++) are used to refer to the code generator that takes the GIL file produced by Devguide and produces executable C (C++) code and XView calls that create your user interface.

`gxv` and `gxv++` refer to the UNIX commands invoked to run the GXV and GXV++ postprocessors.

What You Need to Run GXV and Devguide

GXV and Devguide run on any machine that runs the Solaris 2.x (Solaris 2.0 or later) environment. You need—in addition to Solaris 2.x—the Devguide software, the XView toolkit, and a C compiler.

Installing GXV and Devguide

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

How You Interact with Devguide and GXV

The basic interaction with Devguide is very simple. You use the Devguide program to develop your user interface with the OPEN LOOK look and feel by dragging and positioning glyphs on interface windows. You can save a description of the interface in one or more `.G` files. You can create a project (`.P`) file if you save the interface in more than one `.G` file. Interface building 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 saved it in one or more `.G` files, run the GXV program on the .G or .P file you created. GXV produces four files from a single .G file, referred to as `<fn>.G` in Figure 1-2 below. The `.G` suffix is the standard suffix for GIL files.

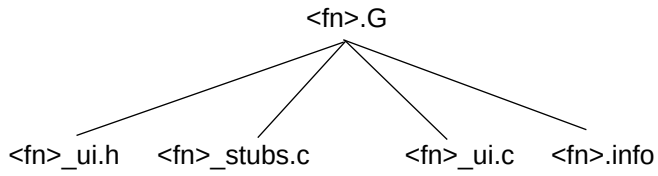


Figure 1-2 GXV Generated Files from Single `.G` File

GXV also automatically generates a program `Makefile` — and a `.db` file if the `-r` argument is used.

The C code GXV produces is by default ANSI C. All the sample code in this manual is written in ANSI C. Use `gxv`'s `-k` option to produce Kernighan and Richie C.

1

Getting Started with GXV 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 GXV. This chapter is intended to be a "quick start guide" that provides you with the minimal instructions necessary to start using GXV. The topics discussed here are covered in greater detail in Chapter 3, "GXV Tutorial" and Chapter 4, "GXV Functionality in Detail."

Creating an Application with GXV

To create an application with Devguide and GXV, you generally follow these steps:

1. **Design an interface with Devguide and save it in a GIL file.**
   You can optionally save an interface in several `.G` files and create a project file. See the *OpenWindows Developer's Guide: User's Guide* for more information on designing interfaces.

2. **Use Devguide's Test Mode to experiment with a partial prototype of your application.**
   In Test mode, the elements of your user interface generally work as they will in a finished application: buttons highlight, selections set choices, and menus appear. Any GXV predefined connection you set also works. After you are done with testing, reenter Build mode.

3. **Generate XView code by running GXV on the GIL or project file.**
   GXV generates three C files, `a.info` file (for help text), and a `Makefile` for a single GIL file. The C files contain the interface data structures and source code. The `Makefile` contains the `make` utility commands to build the interface. GXV defaults to ANSI C. You can generate Kernighan and Richie C code by using `gxv`'s `-k` option. GXV may also produce a `.db` file if menu accelerators are used.

   If you run GXV on a project (`.P`) file, it generates the C and help files for individual GIL files, but only one `Makefile`. It also generates two additional C files—one with a `.c` and one with a `.h` extension.

   `gxv(++)` can also be run from the code generator tool.

4. **Use `make` to build a working prototype of the interface.**
   GXV generates all the code necessary to create an executable file, so you can compile the code by itself immediately after you generate it.

5. **Insert your application code in the callback function templates generated by GXV in the `_stubs.c` file.**
   If you create any additional source files for your application, add the files' names to the `SOURCES.c` line in the `Makefile`.

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

In steps 3, 4, 5, and 6 above, you can use Devguide's Code Generator Tool to compile code, edit files, and run the `make` utility. See "Using the Code Generator Tool" on page 8 for details on using this tool.

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

Designing an Interface in Devguide

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

1. **In Devguide, choose "Devguide..." from the Properties menu.**
   The Devguide Properties window appears (see Figure 2-1 on page 7).

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

3. **Dismiss the Devguide Properties window.**

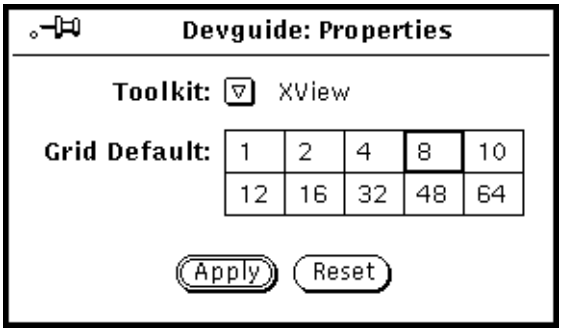


Figure 2-1 The Devguide Properties Window

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*). For example, you might want to specify that when SELECT is clicked (the event) on a button (the Source), a pop-up (the Target) appears (the action). (See "Tying a Pop-Up Window to a Button" on page 90 for details on setting up this connection.)

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

Devguide and GXV provide a variety of predefined actions. "Action Menu Items" on page 50 furnishes a complete list of these actions. For each connection you set up in Devguide, GXV 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 and a line of dummy code that prints the connection name to the shell tool you use. You can supplement or modify this code with your own application code. You can eliminate the print statements after your application is working correctly.

To set up a callback that consists entirely of your own code, choose CallFunction for the action when you create a connection in Devguide. Then specify a function name. GXV generates a callback under the specified name that contains a line of dummy code. It is usually helpful to choose CallFunction callback names that indicate the When and the Source items for connections, for example, `button1_notify`. This makes it much easier to identify the callback templates in the generated 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 pop-up into which you can add your code. GXV includes this code in the generated callback.

Note – If you have created a project file where more than one of its `.G` files invokes the same CallFunction callback, the callback appears in the `<project_name>.c` file.

Using the Code Generator Tool

Use Devguide's code generator tool to generate and compile code, and to run applications for these code generators, without having to type commands at the command line. The code generator tool's main window is resizeable.The inner pane is a command tool.

To start the code generator tool, choose Code Generator from Devguide's File menu.

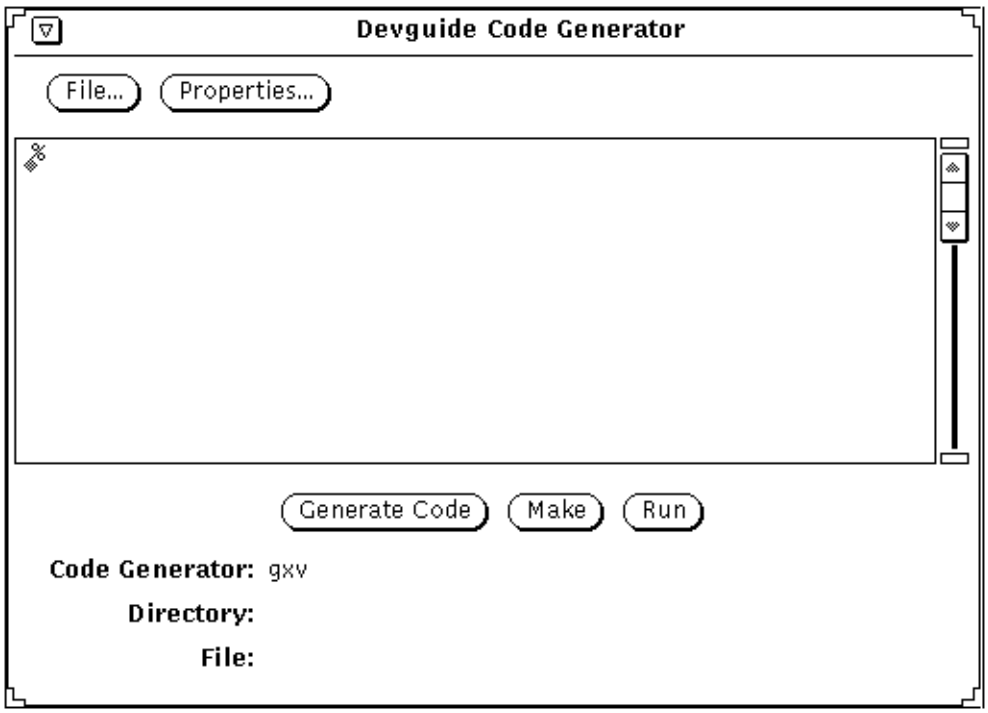


Figure 2-2 The Code Generator Tool

Click SELECT on the File... button to display a file chooser pop-up window. Use this to select a GIL or project file. Click SELECT on the Properties... button to display a pop-up window to set make and run-time arguments, and code generator properties. Click SELECT on the Generate Code button to generate code from your GIL or project file. Click SELECT on the Make button to compile source code and link object code. Click SELECT on the Run button to run the executable program.

See the *OpenWindows Developer's Guide: User's Guide* for more information about the file chooser.

Note – The code generator window can remain open regardless of whether the Devguide window is open.

Using the File... Button

When you click SELECT on the File... button, a file chooser pop-up window appears. When you select a file, the main window's Directory and File fields are filled in. The file name also appears after the code generator name in the Code Generator field.

Using the Properties Window

When you click SELECT on the Properties... button, the Code Generator Properties pop-up window appears. The GXV Properties pop-up window is shown in Figure 2-3 on page 11.

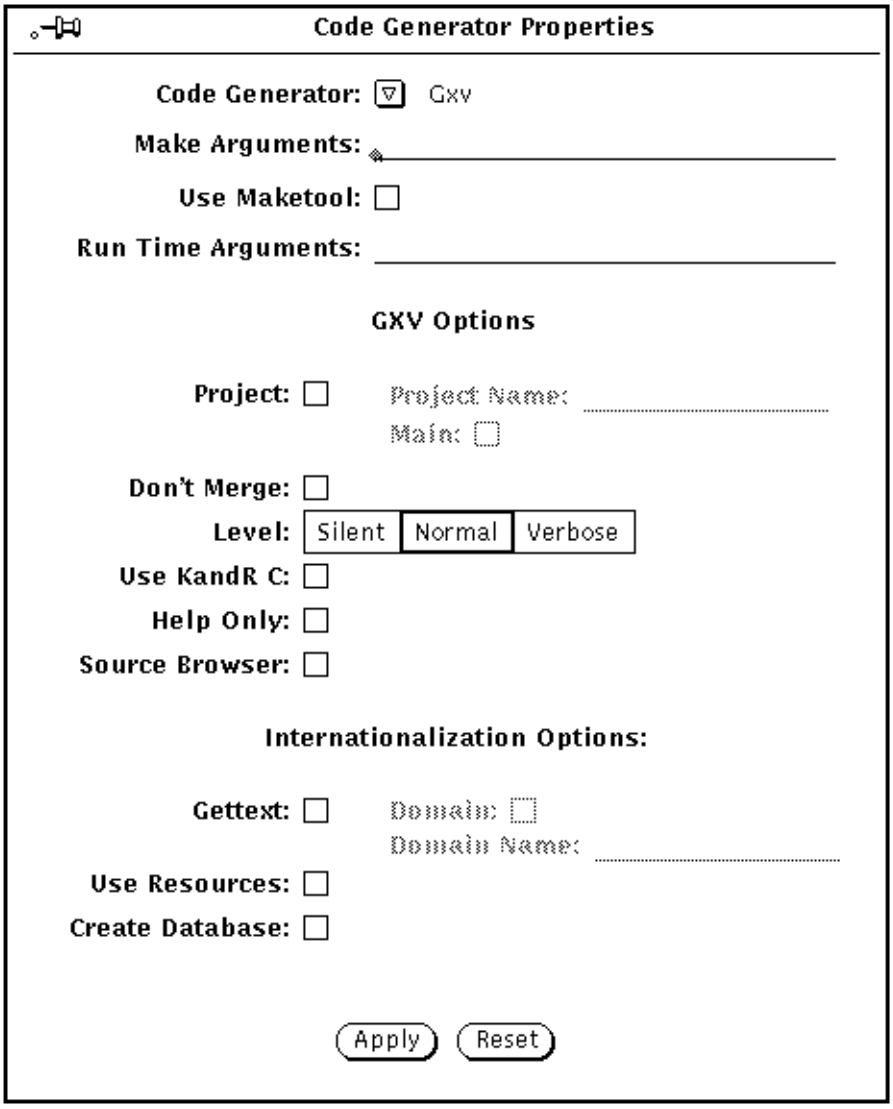


Figure 2-3 GXV Code Generator Properites window

- **Select a Code Generator**

Use the Code Generator abbreviated menu button to select the GXV, GXV++, or GOLIT code generator.

Note – If an application has already used a particular code generator to generate code from a GIL file, use the same code generator for that application. For example, if you used GXV, continue to use GXV. If you use a different code generator, you will obtain unusable hybrid results.

- **make Arguments**

Specify `make` arguments in the space provided. The `clear` argument is one example of a `make` argument.

Note – The `Makefile` contains all the property options that you used the last time you generated code using the Code Generator Tool. 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.

Note – The Maketool option works only if you have installed the SPARCworks or ProWorks software and updated your path accordingly.

- **Runtime Arguments**

Specify any runtime arguments in the space provided. The `-scale large` argument is one example of a run-time argument.

GXV Options

The Code Generator Properties window provides an easy way to specify command-line options without typing them. For more information about command-line options, see “Using Flags with GXV” on page 52.

- **Project** – Generates code for a project. You must specify a project name. See “Options to GXV” on page 101 for details on using the `-p` option for internationalization. `This` is equivalent to specifying the `-p` option.
  - **Project Name** – Specify a project name.

  - **Main** – Generates source code for *projname*`.c` and *projname*`.h` files. Only works when **Project** is selected. Use this option if you have already run GXV on `.G` files comprising a project. Equivalent to specifying the `-m (-main)` option.
- **Don't Merge** – Prohibits merging of existing and new `_stubs.c` files. Checking this box is equivalent to specifying the `-n (-nomerge)`option.
- **Level** – Provides three selections: **Silent**, **Normal**, and **Verbose**.
  - **Silent** – No trace messages are generated. Equivalent to the `-s (-silent)` option.
  - **Normal** – Normal mode.
  - **Verbose** – Generates a message as each new UI element is encountered. Equivalent to specifying the `-v (-verbose)` option.
- **Use K&R C** – Specifies that K&R C code be generated (the default is ANSI C). Equivalent to specifying the `-k (-kandr)` option.
- **Help Only** – Generates only the help text file (the `.info` file). Equivalent to specifying the `-h (-helpfile)` option.
- **Source Browser** – Adds the `-xsb` switch to the compiler options in the makefile if you are using ANSI C. Adds the `-sb` switch if you are using K&R C. The resultant compiled code can then be used with Source Browser in SPARCworks or ProWorks. Equivalent to specifying the `-sb` option.

Internationalization Options

- **Gettext** – Generates `gettext` string wrappers for internationalization. Equivalent to specifying `-g (gettext)` option.
  - **Domain** – Overwrites the default domain name of *filename*_labels generated for the `dgettext()` function for internationalization. You must specify a new domain name for its replacement. Equivalent to specifying the `-d domain` option.
  - **Domain name** – Provide a Domain name here.
- **Use Resources** – Generates XView `XV_USE_DB` attributes for internationalization. Equivalent to specifying the `-r` option.
- **Create a Database** – Creates an X resource database for internationalization (the `.db` file). Equivalent to specifying the `-x (-xdb)` option.

Applying Changes

Click SELECT on the Apply button to effect the changes. Click SELECT on the Reset button to clear the propery sheet. (You must select Apply for the clearing to take effect.)

Generating XView Code from Devguide GIL Files

To generate XView code from a `.G` file, run GXV on the file by entering the following command in a shell tool:

```
% gxv <filename>
```

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

```
% gxv -p <project_filename>
```

It is not necessary to type the `.G` or the `.P` file name extension after the file names. GXV adds the appropriate extension automatically if you have not done so.

GXV-Generated Files

GXV-generated files contain all the source code necessary to create an executable file that uses the XView toolkit and the `libguide` and `libguidexv` runtime libraries.

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

- `<UIFileName>_stubs.c` – Skeleton main program for the executable. It includes the callback templates that you insert your application code into.
- `<UIFileName>_ui.c` – Code that defines the user interface objects. You should *not* alter the `_ui.c` file; any changes you make to this file will be overwritten by GXV when you use `make`, and you will lose all your work.

- <UIFileName>_ui.h – Header file that declares the user interface objects, external callbacks, and external creation procedures. You should *not* alter the _ui.h file; any changes you make to this file will be overwritten by GXV when you use make, and you will lose all your work.
- Makefile – A template makefile to build the executable file. Makefile contains make utility commands to compile your interface and any application code you include in the _stubs.c file. See *Programming Utilities Guide* for more information on the make utility.
- <UIFileName>.info – A file that contains any help text you write for objects in the interface.

The names of these files are derived from the name of the associated GIL file. For example, if you use GXV to generate source code for a GIL file named display.G, GXV creates the files display_ui.c, display_ui.h, and display_stubs.c and display.info. GXV also generates a Makefile.

When you run GXV on a project (.P) 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 and templates for callbacks shared by the GIL files in the project.
- <project_name>.h – External declarations for callbacks shared by the GIL files.
- Makefile – A makefile for the entire project.

The names of the first two files are derived from the name of the project file. For example, if you use GXV to generate source code for a project file named myproject.P, GXV creates the files myproject.c, myproject.h and Makefile.

Compiling and Testing Interface Code

To compile the interface code generated by GXV, use the Code Generator Tool (see “Using the Code Generator Tool” on page 8) or type the following after you have generated the code:

```
% make
```

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

Integrating Application Code with GXV Interface Code

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

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

Referencing Interface Objects in GXV

Each top level object (windows and menus) and all objects it contains are declared as C data structures inside the `_ui.h` file. Each of these objects is initialized and assigned to a global variable in the `_ui.c` file.

As an example, suppose you have created a base window that contains a text pane. The C structure declaration for these objects follows.

```
    typedef struct {
             XV_opaque    window1;     /* window's Object Name is window1 */
             XV_opaque    textpane1;   /* text pane's Object Name is textpane1 */
} my_app_window1_objects;/* interface name is my_app.G */
```

The window object's global variable definition in the `_stubs.c` file is shown below.

```
my_app_window1_objects*My_app_window1;
```

You can reference the text pane and display the word `Message` inside the pane by using the following piece of code.

```
xv_set(My_app_window1->textpane1, TEXTSW_CONTENTS,“Message”,
    NULL);
```

Regenerating Code for a Modified Interface

GXV and 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 `gxv` or `make` (`make` calls `gxv`) on the altered GIL file to regenerate the code.

Each time you run it, `gxv` 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. Adds any code that you inserted in the original `_stubs.c` to 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, `gxv` performs these steps for each of the GIL files in the project. It treats the `<projectname>.c` and `<project_name>.h` files like `_stubs.c` files, creating `.BAK` and `.delta` files, and merging any code you added into the new files.

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, 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 Devguide provides for the objects. The interface should appear like the one displayed in Figure 2-4.

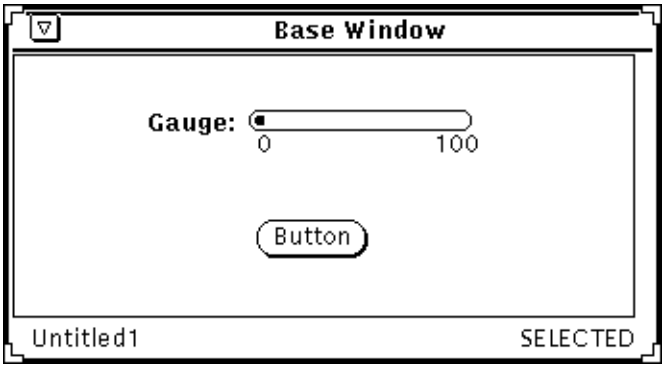


Figure 2-4 Example Interface as it Appears in Devguide

2. **Create a connection for the button in the Connections Manager Window.**
   Make sure that the button is the Source and the Target for the connection. Select Notify from the When menu and CallFunction from the Action menu. Enter `incr_gauge` on the Arg text field as the function name and click SELECT on the Connect button (GXV does not provide a built-in action to increment the gauge whenever the button is pressed). After you have created the connection, the Connections Manager window should look something like the one shown in Figure 2-5 on page 19.

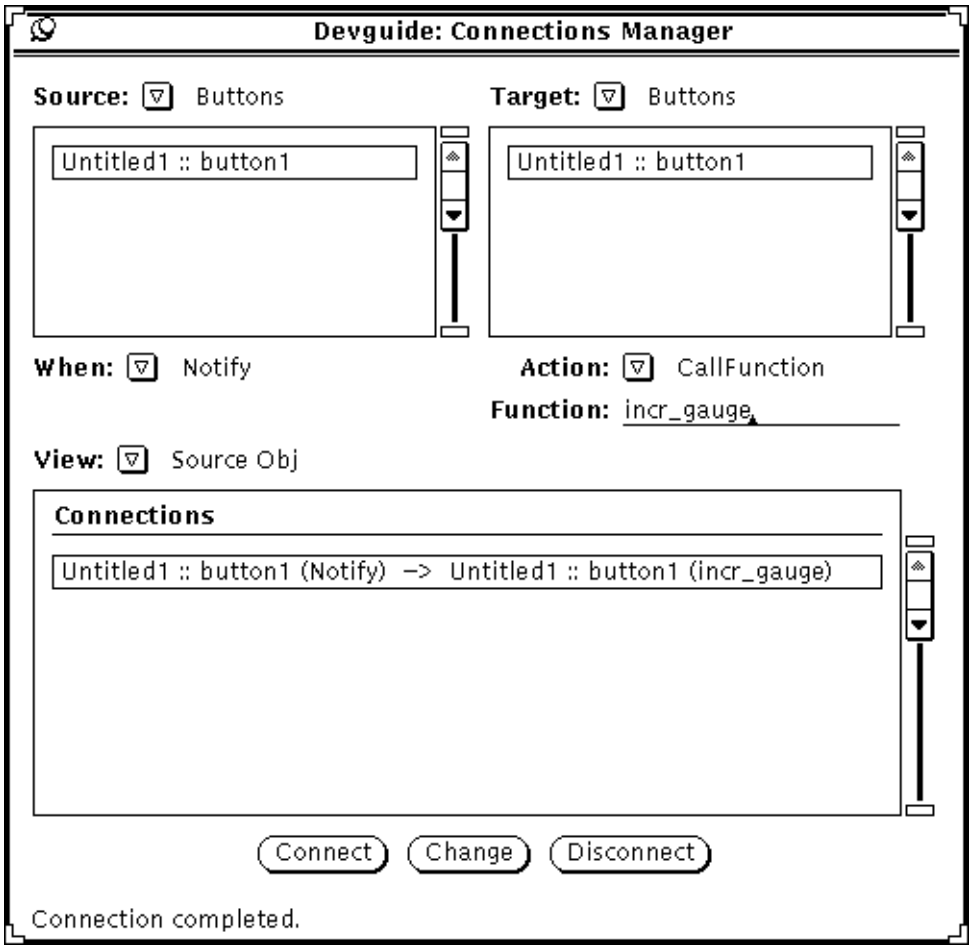


Figure 2-5 CallFunction Connection for Button

3. **Save the interface to a GIL file and close Devguide.**
   Choose Save As from Devguide's File menu and save the file as `example1`. Devguide appends the `.G`, to make the GIL file's name `example1.G`. Closing the Devguide window makes the user interface you created disappear.

4. **Use the Code Generator Tool (see "Using the Code Generator Tool" on page 8) or type** `gxv example1` **to generate the code.**
   GXV generates the interface code and the `Makefile`.

5. **Type** `make` **to build a test copy of the interface.**
   `make` builds an executable file named `example1`.

6. **Modify the** `example1_stubs.c` **file to include your application code.**
   The file originally generated by GXV should look like the one in Code Example 2-1. You only need to change the `incr_gauge()` callback template. Add code to increment the gauge. When you are done, the `incr_gauge()` callback should look something like the code in Code Example 2-2 on page 22. (The added code appears in **`bold print`**. Note that an `fputs()` call has been deleted.)

7. **Type** `make` **to build the completed application.**
   GXV preserves the application code you added to `example1_stubs.c` and builds an executable file called `example1`.

8. **Type** `example1` **to run the executable file.**
   When you run `example1`, the gauge value increases by one each time you press the button.

   Each time you press the button displayed in the base window, the application prints the connection name in the shell you use. Once you have the application running correctly, you can delete the callback print statements that generate these messages.

Code Example 2-1 `example1_stubs.c`

```
/* example1_stubs.c - Notify and event callback function stubs.
* This file was generated by 'gxv' from 'example1.G'. */
#include <stdio.h>
#include <sys/param.h>
#include <sys/types.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/xv_xrect.h>
#include "example1_ui.h"
/* Global object definitions. */
example1_window1_objects*Example1_window1;
```

Code Example 2-1 `example1_stubs.c` (Continued)

```
#ifdef MAIN
/* Instance XV_KEY_DATA key. An instance is a set of related
* user interface objects. A pointer to an object's instance
* is stored under this key in every object. This must be a global variable. */
Attr_attribute          INSTANCE;

main(
    int    argc,
    char   **argv)
 {  /* Initialize XView.*/
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    INSTANCE = xv_unique_key();
     /* Initialize user interface components.*/
    Example1_window1 = example1_window1_objects_initialize(NULL, NULL);
     /* Turn control over to XView.*/
    xv_main_loop(Example1_window1->window1);
    exit(0);
 }
#endif
 /* Notify callback function for 'button1'.*/
 void incr_gauge(
           Panel_item    item,
           Event         *event
           )
{   example1_window1_objects *ip = (example1_window1_objects *) xv_get(item, XV_KEY_DATA,
       INSTANCE);
    fputs("example1: incr_gauge\n", stderr);
    /* gxv_start_connections DO NOT EDIT THIS SECTION */
    /* gxv_end_connections */}
```

Code Example 2-2 Notify callback for button1

```
/* Notify callback function for 'button1'.*/
 void incr_gauge(
           Panel_item     item,
           Event          *event)
{     example1_window1_objects *ip = (example1_window1_objects *)
                xv_get(item,XV_KEY_DATA,INSTANCE);

      static int     gauge_value;

      if (gauge_value < 100)
           xv_set(ip->gauge1, PANEL_VALUE, gauge_value++, NULL);

      /* gxv_start_connections DO NOT EDIT THIS SECTION */
      /* gxv_end_connections */
```

GXV Tutorial 3

This chapter demonstrates how to create a complete application with Devguide and GXV. 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 consists of a text pane which you can enter text into and controls that modify the appearance of the text. More specifically, it consists of a base window with:

- A text pane which you can enter text into.
- Two layered control areas:
  - A control area with an exclusive setting that sets the font to Roman or Lucida.
  - A control area with a checkbox setting that sets the font style to bold, italic, or both.
- A third control area with two buttons.
  - A menu button that switches between the two layered control areas.
  - A button that displays a popup window.

The application provides a popup window, with:

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

The application is shown in Figure 3-1 on page 24. Note that the control area that provides the font style checkbox is not visible. Figure 3-6 on page 33 shows the base window interface with the style checkbox setting in front.

3

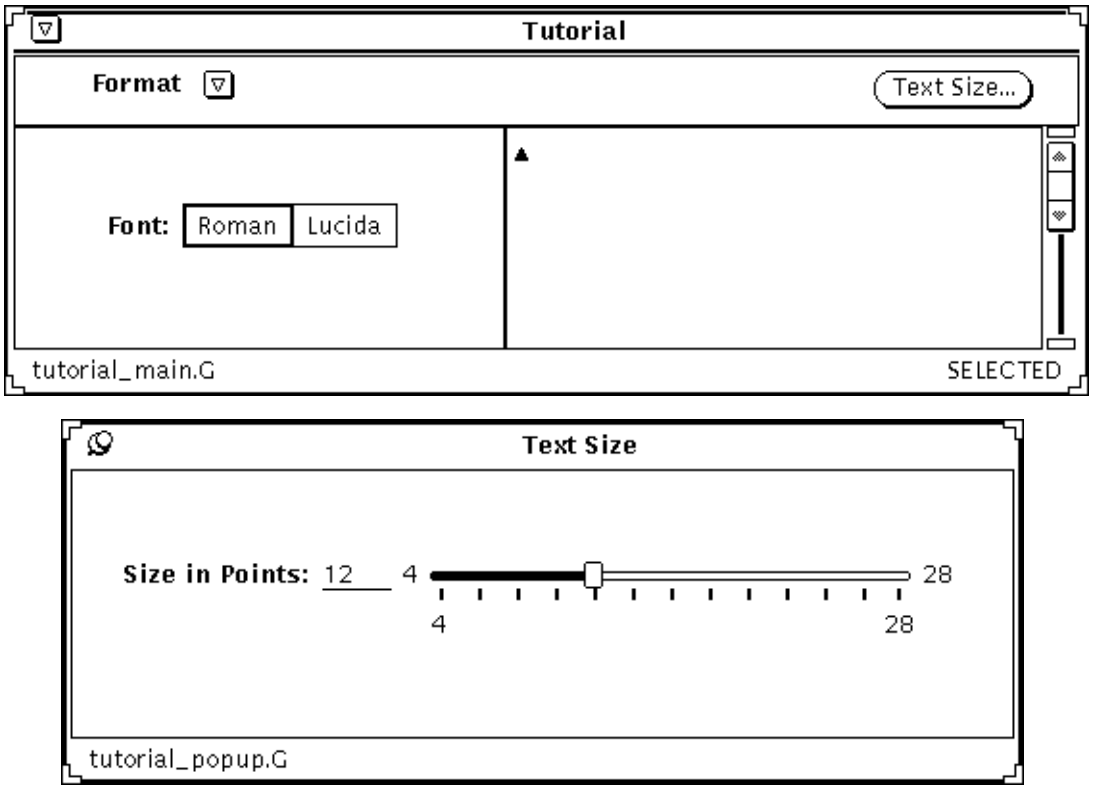


Figure 3-1 Tutorial Application

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 the Interface

To create an interface with Devguide, you drag glyphs from Devguide's palette onto the workspace. Each glyph turns into an interface element. You can lay elements down one above another or one next to another to assemble the

interface you want. You can also layer panes in your interface. To customize an element's properties, open its property window, set property values, and click SELECT on the Apply button.

You can store your application in one or more GIL files. Create a project file if you use more than one window; store each window in a separate GIL file. Note that the term *interface* refers to both the portion of your application stored in each GIL file, as well as to the entire application itself.

In this example, you will create a project with two interface (GIL) files: one for the base window and one for the pop-up window. You create the first interface by simply dragging glyphs onto the workspace. To create an additional interface file, choose New Interface... from Devguide's File menu; then assemble the interface.

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 gxv in that directory.

1. **Create a new subdirectory of your home directory and name it** `Tutorial`**.**

```
% mkdir Tutorial
```

   You can delete this subdirectory when you are done with the tutorial. `Tutorial` should be empty when you start.

2. **Change directories from your home directory into** `Tutorial`**.**

```
% cd Tutorial
```

3. **Start Devguide.**

```
% devguide &
```

3

Creating the Base Window Interface

All applications have one main base window that is visible (or iconic) when you first run an application.

Figure 3-2 shows the completed base window interface as it appears in the Interface Browser. (To view this browser after you have saved the interface, choose View from Devguide's File menu, then choose the interface file name from the Interfaces submenu.)

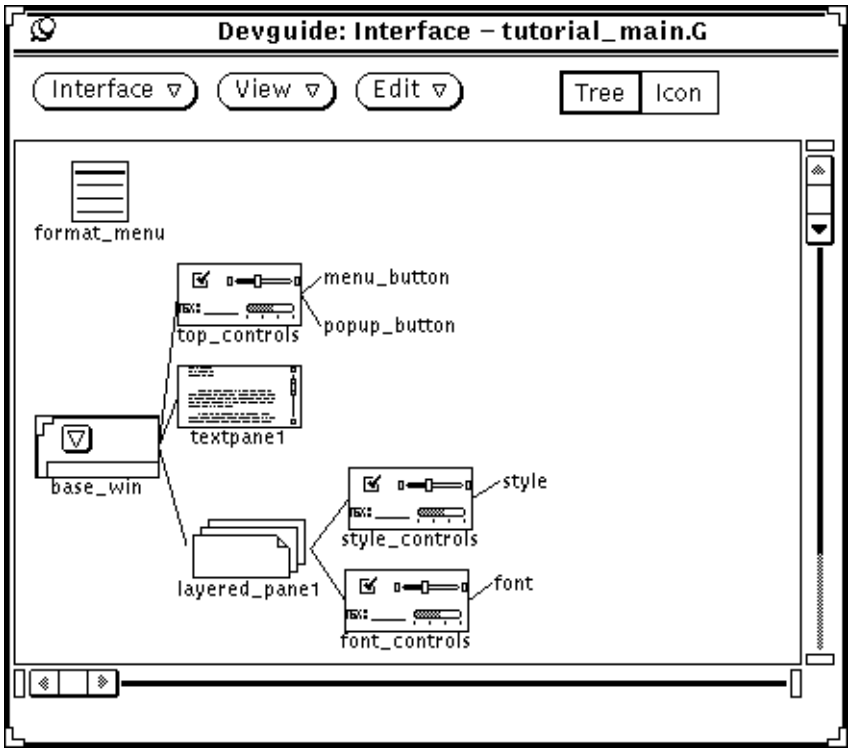


Figure 3-2 Base Window Interface Browser

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 the interface 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.

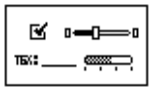
2. **Customize the base window's properties.**
   Double-click SELECT on the base window's interior or footer to open its property window. Then, change the values in the fields.

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

Note – The changes you make on a property 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 (e.g. text panes and control areas) directly on a base window or popup window. However, you *must* place control elements, such as buttons and sliders, on a control area. You can place more than one control area in a window.



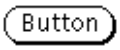
1. **Create a control area by dragging the Control Area glyph onto the base window.**
   Place and resize the control area so it looks like the area in the top of the base window in Figure 3-3 on page 29. This control area will contain the application's Format and Text Size buttons.

2. Customize the control area's properties.

Object	Property Field	Value
Control area (controls1)	Object Name	`top_controls`

Adding Buttons to the top_controls Control Area

You create two different kinds of buttons for this sample application: an abbreviated menu button and a button to display a popup 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.

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 (button1)	Object Name	`menu_button`
Left button (button1)	Label	`Format`
Left button (button1)	Type	Abbreviated Menu
Right button (button2)	Object Name	`popup_button`
Right button (button2)	Label	`Text Size...`

Later on, you will create a menu and attach it to the left button. Figure 3-3 shows how the interface should appear when you have completed these steps.

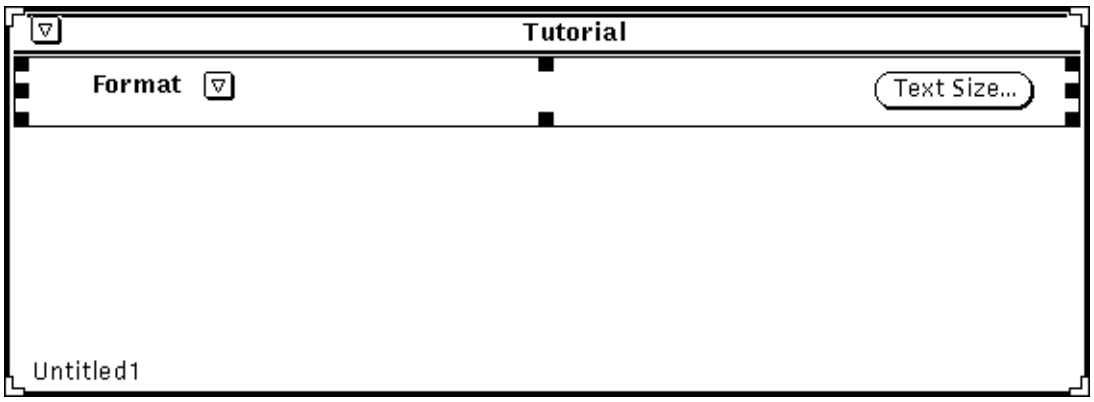


Figure 3-3 Base Window Interface with Buttons

Adding a Text Pane to the Tutorial Base Window

Devguide provides three objects that you can display and write text in: text fields, multiline text fields, and text panes.

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

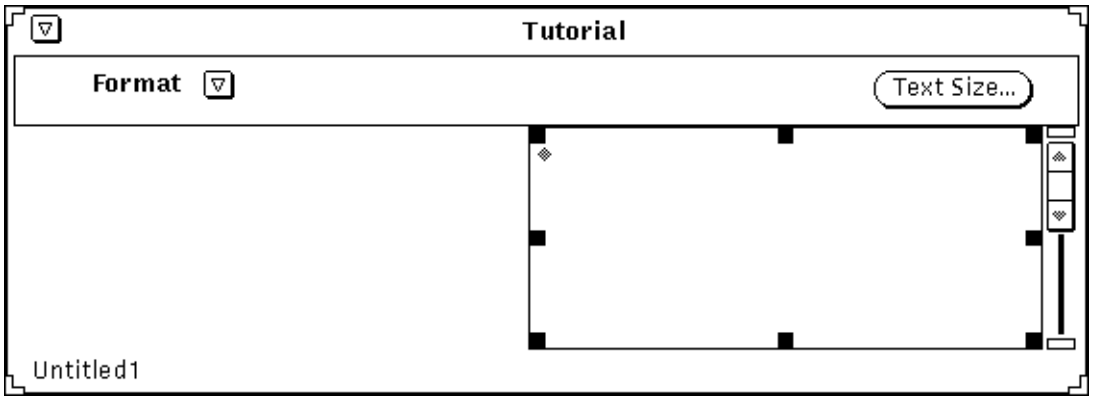


Figure 3-4 Base Window Interface with Text Pane

Creating Layered Control Areas

Devguide allows you to add layered panes to your application. You can layer control areas and canvas, text, and term panes. In the completed application, only one pane will be visible at a time. The application must include a connection—to make the pane visible—for each pane that you layer. 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 31.
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`**

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

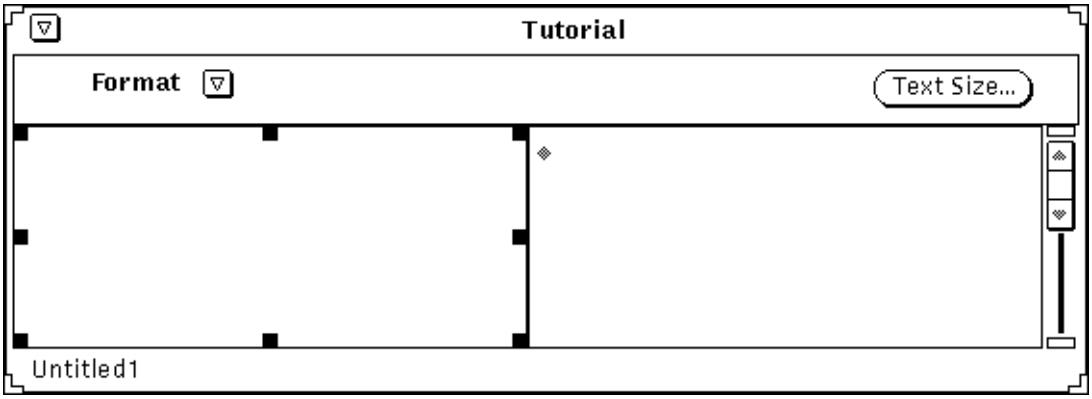


Figure 3-5 Base Window Interface with Layered Control Areas

Adding Controls to the Layered Control Areas

In this section you will add controls that control the text pane's font and style to the layered control areas. You will add an exclusive setting for the font to one control area and a checkbox setting for the style to the other.

To add an exclusive setting to the `font_controls` control area:

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

   Customize the setting's properties as follows:

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

2. **Customize the setting items' (Choices) properties.**
   The Choices scrolling list contains two choices.

Object	Property Field	Value
First Choice	Label	`Roman`
Second Choice	Label	`Lucida`

Devguide automatically sets the first choice as the default choice in the exclusive setting, unless you alter the choices' Not Selected / Selected property.

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 Devguide's 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:`
Checkbox Setting	Rows / Columns	(Rows) 1

3. **Customize the individual checkboxes' (Choices) properties.**
   The Choices scrolling list contains two choices.

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

Figure 3-6 shows the interface with `style_controls` in front; Figure 3-1 on page 24 shows it with `font_controls` in front.

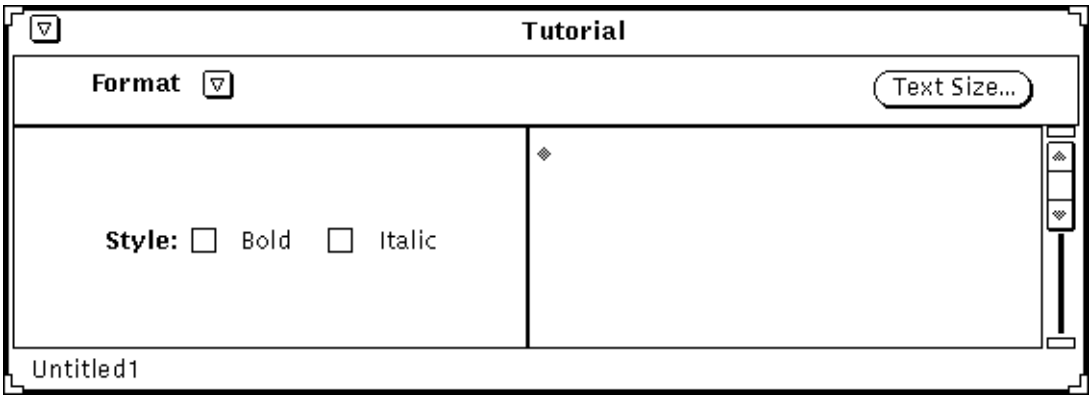


Figure 3-6 Base Window Interface with Style Controls Displayed

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.**
   Devguide appends the `.G` extension to the interface file.

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`. Customize the menu's properties.

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 of the tutorial, you create an additional interface consisting of a pop-up window, a control area, and slider. Figure 3-7 on page 35 shows the interface as it appears in the Browser.

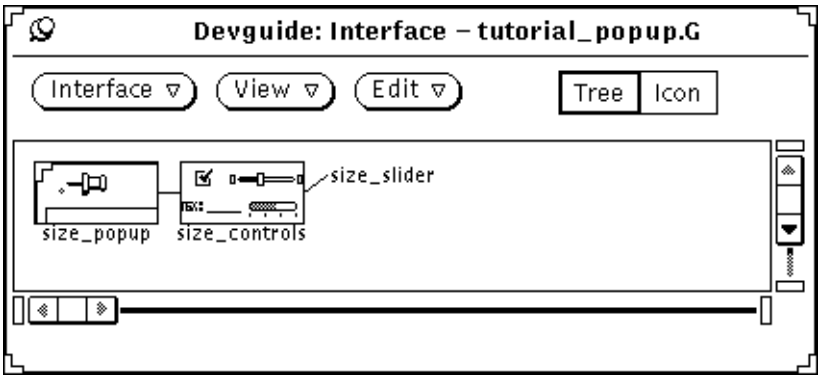
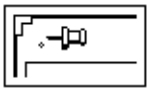


Figure 3-7 Pop-up Window Interface Browser

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	Window Parent	`base_win`
Pop-up window	Label	`Text Size`

Adding a Control Area to the Pop-up Window

Add this control area to the pop-up window the same way you added the control areas to the base window.

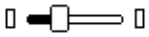
1. **Drag the Control Area glyph onto the pop-up window and resize the control area.**
   Resize the control area to fill the pop-up window.

2. **Customize the control area's properties.**

Object	Property Field	Value
Control area (controls1)	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.**
   Position the slider as shown in Figure 3-8 on page 37.

2. **Customize the slider's properties.**

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	Ticks	`13`
Slider	Tick String Min	`4`
Slider	Tick String Max	`28`
Slider	Initial Value	`12`

Figure 3-8 shows the completed pop-up window interface (after it has been saved).

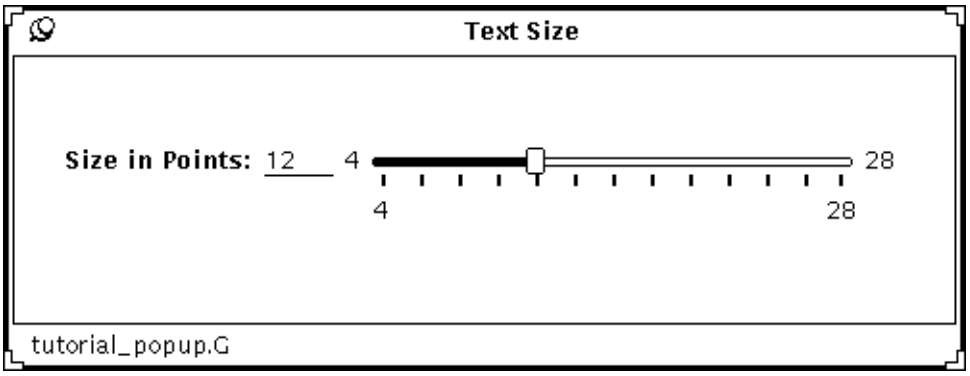


Figure 3-8 Completed Pop-Up Window Interface

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, you can create a project file. 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; in particular, icons 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`. The `tutorial` project consists of `tutorial_main.G` and `tutorial_popup.G`.

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 must insert custom code into

In this section you establish 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 is pressed.

You also set up 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. Then click SELECT on the Connect button to complete 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 entire 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 Devguide's Properties menu.
2. **Click SELECT on Connections... in the upper half of the Menu Editor window.**
   The Connections Manager window appears.
3. **Choose Buttons from the Connections Manager's 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.

You can also attach a menu to a button by using the Menu button on the Buttons property window. See the Reference chapter of the *OpenWindows Developer's Guide: User's Guide* for details.

Creating Connections to Switch between Layered Control Areas

To switch between the layered control areas, you need to create two connections: one for each menu item in the Format menu.

- Show `style_controls` when the user chooses Style.
- Show `font_controls` when the user chooses Font.

To create the connections (in the Connections 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.

Create the following connections

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

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

Use drag and link to create this connection. Select the Text Size... button. Press the Meta key and drag the pointer from the button to the pop-up window. The pointer turns into a cord with a plug at the end.

Note – Use the key marked “Alt” if there is no Meta key (usually marked as a diamond) on your keyboard.

When you release the SELECT button, the Connections Manager 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

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. Specify the connection as follows.

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

Creating Connections to Set Font and Style

To set up connections to set the text's font and style, choose Setting Items from the Source and Target menus. The item names displayed in the scrolling lists are prefixed with the setting name; for example, font.Lucida. Specify the connection as follows.

Source	Target	When	Action	Arg
Lucida	Lucida	Notify	CallFunction	`set_lucida`
Roman	Roman	Notify	CallFunction	`set_roman`
Bold	Bold	Notify	CallFunction	`set_bold`
Italic	Italic	Notify	CallFunction	`set_italic`

Saving

Save your application in order to save the connections.

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 behave. First choose Test mode. Then you can display the Format menu, scroll through the layered panes, and display the Text Size pop-up window.

1. **Choose Test mode by clicking SELECT on Test in Devguide's base window controls area.**
   The Test choice is highlighted. All the predefined connections in your application will work. The only connections that do not work are CallFunction and ExecuteCode connections.

2. **Put the pointer over the text pane you created and enter some sample text.**
   Notice that the scroll bar is activated if you enter more text than can appear in the text pane at one time. *The text you enter is not saved.* The text pane will be empty when you start up the completed tutorial. *The font of the text you enter in Test mode will be Devguide's default font.*

3. **Click 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.

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 font size in Test mode.

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

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

Generating Interface Source Code

1. **Use the Code Generator Tool or run the `gxv` command on your project file.**
   Invoke `gxv` with the `-p` option because `tutorial` is a project file. GXV lists the files it is currently reading and writing:

```
% gxv -p tutorial
 gxv: reading tutorial.P
 gxv: reading tutorial_basewindow.G
 gxv: writing tutorial_basewindow_ui.h
 gxv: writing tutorial_basewindow_ui.c
 gxv: writing tutorial_basewindow_stubs.c
 gxv: writing tutorial_basewindow.info
 gxv: reading tutorial_popup.G
 gxv: writing tutorial_popup_ui.h
 gxv: writing tutorial_popup_ui.c
 gxv: writing tutorial_popup_stubs.c
 gxv: writing tutorial_popup.info
 gxv: writing tutorial.c
 gxv: writing tutorial.h
 gxv: writing Makefile
```

Creating Custom Code Using GXV Files

In this section you alter four GXV-generated files to make the CallFunction connections work. The callback functions GXV generates for these connections contain print statements that you replace with custom ANSI C code. You also need to define global variables to register the user's choice of Bold and/or Italic, and Roman or Lucida.

You insert a common function called `change_font()` inside the `set_roman()`, `set_lucida()`, `set_italic()`, `set_bold()`, and `set_textsize()` callbacks to change the text's font, style, and size.

Open the files in your favorite text editor and save all changes.

Customizing the `tutorial.c` *File*

1. **Insert these global variables into the file before the definition of** `main()`**.** The text in the text pane when you first run the application is twelve point Roman regular font.

```
Attr_attribute INSTANCE;
int Bold = 0;
int Italic = 0;
int Lucida = 0;
int Size = 12;

/*
 * main for project tutorial
*/
void
main(int argc,char **argv)
```

2. **Insert a call to** `change_font()` **right before the call to** `xv_main_loop()` **inside** `main()`**.**
   This initializes the text's style and font to the values set above. `change_font()` must be called after all UI objects are initialized and before XView polls for events in `xv_main_loop()`.

```
            change_font();

            /*
            * Turn control over to XView.
            */

xv_main_loop(tutorial_main_base_win->base_win);
```

Customizing the `tutorial.h` *File*

1. **Add extern declarations and define the constants TRUE and FALSE.**
   Include the extern declarations so both `_stubs.c` files can access the variables and `change_font()` function. Redefine 1 and 0 as TRUE and FALSE, respectively, to make the code more readable and boolean variables more obvious.

```
#define TRUE 1
#define FALSE 0

extern int Bold;
extern int Italic;
extern int Lucida;
extern int Size;

extern void change_font();
```

Customizing the `tutorial_popup_stubs.c` *File*

1. **Replace the** `fprintf()` **statement in** `set_textsize()` **with code to read the slider value and scale the font.**
   First retrieve the user's choice for text size, and then set the font size accordingly.

```
Size = value;
change_font();
```

Customizing the `tutorial_main_stubs.c` *File*

You created CallFunction connections for both checkbox and exclusive settings earlier in this tutorial. GXV generates callbacks for each individual item connection you create, as well as a wrapper function that executes item callbacks as appropriate.

For example, in this tutorial GXV generates `tutorial_main_base_win_font_notify_callback()`, which calls `set_roman()` and `set_lucida()`. (GXV constructs names for callback wrapper functions by concatenating together the names of certain objects with the name of the interface containing the function.)

The name of the callback that calls `set_bold` and `set_italic` is given in Step 4.

1. **Add the following line before the statement** `#include "tutorial.h"`**.**
   This provides access to XView's font library.

```
#include <xview/xv_xrect.h>
#include <xview/font.h>
#include "tutorial.h"
```

2. **Replace the call to** `fputs()` **in** `set_roman()` **with code to set the font to Roman.**
   First record the font choice as Roman and then set the font accordingly.

```
Lucida = FALSE;
change_font();
```

3. **Replace the call to `fputs()` in `set_lucida()` with code to set the font to Lucida.**
   First record the font choice as Lucida and then set the font accordingly.

```
Lucida = TRUE;
change_font();
```

4. **Insert the following right before the `for` loop in `tutorial_main_base_win_style_notify_callback`.**
   Initially clear the Bold and Italic values; function calls within the `for` loop reset these booleans to current user input values.

```
Bold = Italic = FALSE;
```

5. **Replace the call to `fputs()` in `set_bold()` with code to set the font type to bold.**
   First record the text style choice as Bold and then set the style accordingly.

```
Bold = TRUE;
change_font();
```

6. **Replace the call to `fputs()` in `set_italic()` with code to set the font type to italic.**
   First record the text style choice as Italic and then set the style accordingly.

```
Italic = TRUE;
change_font();
```

7. **Insert the code for** `change_font()` **at the end of the** `tutorial_main_stubs.c` **file.**

```
void
change_font()
{
        Xv_opaque  text_window;
        Xv_opaque  base_window;
        Xv_opaque  font;
        char       *style;

        text_window =
        Tutorial_main_base_win -> textpane1;

        base_window =
        Tutorial_main_base_win -> base_win;

        if (Bold && Italic)
                    style = FONT_STYLE_BOLD_ITALIC;
        else if (Bold)
                    style = FONT_STYLE_BOLD;
        else if (Italic)
                    style = FONT_STYLE_ITALIC;
        else
                    style = FONT_STYLE_NORMAL;

        /* Set the window busy; finding the font the first time can take a
                    while */
        xv_set(base_window, FRAME_BUSY, TRUE, NULL);

        /* Find the font */
        font = xv_find(text_window, FONT,
                    FONT_FAMILY,
                    (Lucida ? FONT_FAMILY_LUCIDA : FONT_FAMILY_ROMAN),
                    FONT_STYLE, style,
                    FONT_SIZE, Size,
                    NULL);

        xv_set(text_window, WIN_FONT, font, NULL);
        xv_set(base_window, FRAME_BUSY, FALSE, NULL);
}
```

3

Compiling Code

To compile code for your sample application:

- Use the Code Generator Tool, or
- Enter `make` in the shell tool. `make` references the GXV-generated `Makefile` as it compiles your code.

If you build an application containing any `.c` files that aren't generated by Devguide, you should include the files' names in the `SOURCES.C` line in the `Makefile`. This tutorial application only contains Devguide-generated files.

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

```
% make
```

   An executable file named `tutorial` is produced, assuming you've made no errors.

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

```
% tutorial
```

You now have a working application. The base window appears, with the Format menu button, Text Size... button, Font exclusive setting, and text pane visible. You can enter text into the text pane. Try changing the font, style, and size of the text.

This completes the tutorial. The remainder of this programming guide discusses generating GXV code, describes GXV's libraries, and provides some programming tips.

GXV Functionality in Detail 4

Using Connections to Handle Events

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

Use the Connections Manager window to define a connection (an example is shown in Figure 2-5 on page 19). Connections are discussed in “Handling Events Using Connections” in the *OpenWindows Developer’s Guide: User’s Guide*.

When Menu Items

The When menu items that appear for a chosen Source object represent events recognized by the object. The menu items are listed below, along with a brief description of their functionality.

- Select – OPEN LOOK mouse SELECT button or the user’s system equivalent is pressed.
- Adjust - user clicks OPEN LOOK mouse ADJUST (see Note below)
- Menu – OPEN LOOK mouse MENU button or the user’s system equivalent is pressed.
- AnyEvent – Refers to any keyboard or mouse event.
- DoubleClick – OPEN LOOK mouse SELECT button, or the user’s system equivalent, is pressed twice quickly.
- Enter – Pointer enters a window or pane.
- Exit – Pointer exits a window or pane.

- Resize – User resizes a window or pane.
- Keypress – User presses any key on the keyboard.
- Repaint – User repaints a window or pane.
- Notify – User selects a control UI element.
- DraggedFrom – User drags data from a drag and drop target.
- DroppedUpon – User drops data onto a drop site.

Note – If you have a two-button mouse, the ADJUST function is performed by holding down the Shift key and clicking the SELECT (left) button.

Action Menu Items

The Action menu items that appear for a chosen Target object represent actions that can be taken by the object. The menu items are listed below, along with a brief description of their functionality.

- Show – Target object is displayed.
- Hide – Target object becomes invisible.
- Enable – Target object becomes enabled.
- Disable – Target object becomes disabled.
- SetLabel – Target object's label is set.
- GetLabel – Target object's label value is returned.
- SetSelected – Scrolling list item is selected.
- SetLeftFooter – Target object's left footer is set.
- GetLeftFooter – Target object's left footer value is returned.
- SetRightFooter – Target object's right footer is set.
- GetRightFooter – Target object's right footer value is returned.
- SetValueNumber – Target object's numeric field value is set.
- GetValueNumber – Target object's numeric field value is returned.
- LoadTextFile – Text file whose name you specify is loaded.
- CallFunction – Function whose name you specify is called.
- ExecuteCode – Code you specify in the Devguide pop-up window is executed.
- SetValueString – Target object's string field value is set.
- GetValueString – Target object's string field value is returned.

Hints on Using Connections

GXV generates code in the `_stubs.c` file to implement each connection specified in Devguide. For most connections, the code corresponds to `xv_set` function calls. These `xv_set` calls are written in the callbacks for the source objects. The type of callbacks used depends on the “When” part of the connection.

For example, the connection:

Source: button1
Target: popup1
When: Notify
Action: Show

generates a notify callback for button1. Inside that callback is the `xv_set` call to show popup1. GXV automatically names the callback as the concatenation of the interface name, the name of the object’s owner, the object’s name, and the type of the callback. For example, the notify callback generated for the above connection is `itf_popup1_button1_notify_callback()`, assuming `itf.G` is the name of the interface.

For each ExecuteCode connection, GXV generates a callback and puts the code fragment as it is specified in the callback.

For each CallFunction connection, GXV generates a stubs function in the `_stubs.c` file. If the connection is a self-connected one (the source and target are the same), the function will be installed as a callback to the object. For example, the connection:

Source: canvas1
Target: canvas1
When: Repaint
Action: CallFunction `repaint_proc`

specifies that `repaint_proc` is the repaint callback for canvas1. If the CallFunction connection is *not* a self-connected one, GXV will again generate a callback and the function specified will be called from within this generated callback.

A mixture of these different types of connections can be specified and GXV will generate the code according to the scheme described above.

4

Using the gxv *Command*

To run GXV, use the Code Generator Tool (see “Using the Code Generator Tool” on page 8) or enter the following command after a prompt in an open shell.

```
% gxv filename
```

where *filename* is the file name of the GIL file. GIL file names end in `.G` so you can easily identify them. When you enter the file name following `gxv`, you need not add the `.G` extension; GXV automatically adds it. For example, to generate source code for the GIL file `display.G`, use the command `gxv display`. GXV then generates source code for the file `display.G`. If you use the command `gxv display.G`, GXV still generates source code for `display.G`.

The command is similar for project files. In this case, however, you must use the `-p` option. For example:

```
% gxv -p myfile
```

and

```
% gxv -p myfile.P
```

generate code for the project file `myfile`.

Note that if GXV and the GIL (`.G`) or project (`.P`) file don’t reside in the same directory or you aren’t located in their directory when you generate source code, you need to provide a path name for GXV or the file to generate code.

Using Flags with GXV

GXV provides several command line options (or flags) to enhance or customize code generation. These are

- `-s (-silent)` – Silent mode; no trace messages are generated.

- `-v (-verbose)` – Verbose mode; a message is generated as each new UI element is encountered.
- `-n (-nomerge)` – Prohibits merging of existing and new `_stubs.c` files.
- `-k (-kandr)` – Specifies that K&R C code be generated.
- `-p` *project* – Generates code for a project. You must specify a project name. See "Options to GXV" on page 101 for details on using the `-p` option for internationalization.
- `-m (-main)` – Generates source code for *projname*`.c` and *projname*`.h` files. Only works with `-p` option present. Use the `-m` option if you have already run GXV on `.G` files comprising a project.
- `-h (-helpfile)` – Generates only the help text file (the `.info` file).
- `-i (-il8n)` – Generates code for internationalization; combines the `-g` and `-r` options discussed below.
- `-g (-gettext)` – Generates `gettext` string wrappers for internationalization.
- `-d domain` – Overwrites the default domain name of *filename*_labels generated by the `dgettext()` function for internationalization. You must specify a new domain name for its replacement.
- `-r` – Generates XView `XV_USE_DB` attributes for internationalization.
- `-sb` – Adds the `-xsb` switch to the compiler options in the makefile if you are using ANSI C. Adds the `-sb` switch if you are using K&R C. The resultant compiled code can then be used with Source Browser in SPARCworks or ProWorks.
- `-x (-xdb)` – Creates an X resource database for internationalization (the `.db` file).
- `-? (-help)` – Prints out a help message explaining all GXV options. Note that -? *does not* work in the C shell.

You can use as many of these options as you want when you run GXV. To do so, type:

```
% gxv [options] [<filename>]
```

For more detailed information on using options to assist with internationalization, see “Options to GXV” on page 101. An example of using the `-h` option for help text localization can be found in “Options to GXV” on page 105.

Generated Files

Once you create an interface and save it in one or more GIL files, you generate source code by using the `gxv` command followed either by a GIL file name or the `-p` option with a project file name. GIL files end in `.G`, and project files end in `.P`. GXV generates four or more source code files and a `Makefile` you can use with the `make` command to compile the source code files. GXV generates ANSI C code by default.

GXV names its generated files after the original GIL file name, stripping off the `.G` extension and adding new extensions to identify each file. The four file name extensions are:

- `_ui.c` – Graphical User Interface (GUI) code to implement your application
- `_ui.h` – GUI data declarations
- `_stubs.c` – `main()` plus callback functions
- `.info` – XView help file

As an example, if you use GXV to generate source code for a GIL file named `newt.G`, GXV generates the files `newt_ui.c`, `newt_ui.h`, `newt_stubs.c`, `newt.info`, and `Makefile`.

If you run GXV on a project file, the above files are produced for each `.G` file in the project, as well as files with extensions:

- `.c` – `main()` is taken out of `_stubs.c` and put here, along with any callbacks that are shared among `.G` files in a project
- `.h` – extern declarations for any shared callbacks

If a `_stubs.c` file for your `.G` file already exists, running GXV again (for example, after modifications to the `.G` file) will also produce:

- `_stubs.c.BAK` – the previous stubs file
- `_stubs.c.delta` – a log of additions made to your previous stubs file

If you use the options to internationalize your application (see Appendix B, “Internationalization” for details), GXV produces files with extensions:

- .po – strings extracted from your application are placed in this *portable message object file.*
- .db – an X resource database

`_ui.c` *File*

The `_ui.c` file is the primary source code file. It creates the elements of the user interface. It begins with include statements, and follows immediately with one initialization function per instance; an instance is initialized for each window in the GIL file. The instance initialization function is named:

<*interface*>_<*window*>`_objects_initialize()`

where <*interface*> is the name of the user interface and <*window*> is the name of the window being initialized. For example, GXV initializes an instance for the window `financials` in the interface `report` using the function `report_financials_objects_initialize()`.

`Following` the instance initialization function (or functions) are creation functions, one per interface element. Each element creation function is named:

<*interface*>_<*window*>_<*element*>_create()

where <*interface*> is the name of the user interface, <*window*> is the name of the window in which the element is located (which is the window itself if the function creates a window), and <*element*> is the name of the element. For example, a creation function creating a control area named `controls1` in a window named `win` in an interface named `intrfce` would be created by the function `intrfce_win_cntrlarea_create()`.

Both instance initialization functions and element creation functions are public and not static to allow flexibility in their calling order.

Do *not* alter the `_ui.c` file; any changes you make to this file will be written over by GXV as soon as you use `make`, and you will lose all your work. If you want to make changes to objects in your interface, you can make them after the objects have been instantiated by function calls in the `main()` function in the `_stubs.c` file.

`_ui.h` *File*

The `_ui.h` file contains declarations. It has a `typedef` for each window instance structure. The `typedef` structure contains handles to the window and each object within the window.

This file also contains C externs for each initialization and creation function in the `_ui.c file`. The instance initialization functions within the externs create a new copy of an instance and create each object in the instance by calling the object creation function.

Do *not* alter this file; any changes you make will be written over by GXV as soon as you use `make`. Save custom declarations in your own `.h` file so they won't be lost during the compile.

`_stubs.c` *File*

The `_stubs.c` file contains code for each of the connections contained in the GIL file. Functions are generated for the ExecuteCode and CallFunction connections. Callbacks are automatically generated if an object does not already have a handler. Trace `printf` statements are generated to write to `stderr` when the functions are called so you can receive confirmation of the call.

The `_stubs.c` file also contains a `main()` function with calls to the various XView and Devguide initialization functions. You can add to it, or you can use parts of it to add to your own `main()` function.

GXV will not overwrite an existing `_stubs.c` file. Instead, it first generates a new `_stubs.c` file; then it merges the new `_stubs.c` file with the existing one. This preserves the user-added code in the existing `_stubs.c` file.

For example, suppose you have added application code into `myApp_stubs.c`. You then use Devguide to modify your interface; this generates a new `myApp.G` file. If you now run GXV on `myApp.G`, a new `myApp_stubs.c` file is produced, with all your application code intact.

`.info` *File*

When you add help text to elements in a user interface, GXV puts that text in the `.info` file.

You should *not* alter this file because any changes you make will be lost when you compile. Use the help text editor in Devguide to change the help text.

`Makefile` *File*

When you run GXV, it checks the current directory to see if a `Makefile` already exists. If there is no `Makefile`, it generates a new `Makefile`. If it detects a `Makefile`, it does not generate a new `Makefile`. This feature protects a custom `Makefile`, insuring that GXV doesn't write over or modify it. If you need to add new libraries or flags you must do so by hand.

Once GXV generates its own `Makefile`, you can easily customize it to use your own custom code files as source files during compilation. 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. If your application contains more than one GIL file but you haven't created a project file, you must insert the GIL file names here.
- `STUBS.G`, which lists the names of GIL files without customized callbacks (that is, those files whose functions haven't been copied to a custom `.c` file listed in `SOURCES.c`). Any files listed here will cause `make` to compile and link in the associated _stubs.c files.

Note – When you run GXV on a project file, the `SOURCES.G` and `STUBS.G` parameters are not hard-coded into the `Makefile`. Instead, they are included from the `.make` file Devguide generates. The line `include` *projname*`.make` appears in the `Makefile` to do this.

When GXV first generates the `Makefile`, it shows no file names in the `SOURCES.c` and `SOURCES.h` parameters. To add your own source code files to the Parameters list, you must add their file names. For example, if you have several C source files you use for the engine of the application, you must include their names in the `SOURCES.c` parameter.

Note that if you add a file to `SOURCES.c` that includes function callbacks from the GXV-generated `_stubs.c` file for an interface, you must remove the GIL file name of that interface from the `STUBS.G` parameter. You cannot list files with the same callbacks in both the `SOURCES.c` and the `STUBS.G` parameters. If you do, you'll get multiply defined symbols.

`.c` *File*

When you make a project file, GXV produces a file named *projname*`.c` (for a project named *projname*`.P`). This file contains the `main()` function for your program. It also contains all functions that are common to more than one `.G` file in your project. These C declarations will not appear in the corresponding `_stubs.c` files.

For example, suppose you create a project which includes two menus saved in separate `.G` files. Both menus have a Save As... menu item. GXV will store the code for Save As... created by a CallFunction connection in the `.c` file. This code will not appear in the `_stubs.c` file for either menu.

`.h` *File*

When you make a project, GXV also produces a file named *projname*`.h`. This file contains the C declarations for all the interface objects in the project. These C declarations will not appear in the corresponding `_ui.h` file.

For example, suppose you create a project consisting of two base windows saved in separate `.G` files. The `_ui.h` file for an individual `.G` file in the project will include the typedef for its window, and external declarations for initialization and creation functions. *projname*`.h` contains the external declarations for all windows.

`_stubs.c.BAK` *File*

Every time GXV generates a new `_stubs.c` file, it also produces a backup of the existing `_stubs.c` file. This backup file has the extension `_stubs.c.BAK`. The next time you run GXV, it overwrites this backup file with a new one.

`_stubs.c.delta` *File*

This file describes the changes made to the `_stubs.c` file since the last run of GXV. That is, it logs the differences between the new `_stubs.c` file and `_stubs.c.BAK`.

`.po` *File*

If you use GXV's `-i` option (for internationalization), the XView utility `xgettext` produces the `.po` file. This file contains all text strings used by the application. (These strings will be translated into another language by the localizer.) For example, UI element labels appear in the `.po` file. Once translated, `msgfmt` is used to compile this file and rename it with a .mo extension. That is, `msgfmt` compiles *filename*`.po` and produces *filename*`.mo`.

`.db` *File*

The `.db` file is generated during the internationalization process. The Devguide utility `gmomerge` takes a `.mo` file (mentioned above) and a `.G` file, and produces an internationalized `.G` file. The localizer then repositions the UI elements so that the interface appears as it should, and saves it. Running GXV with the `-x` option on this new `.G` file generates the `.db` file. The `.db` file contains the size and position data for the UI elements of the localized interface in an X resource database. This database is used by the XView toolkit if you don't use relative layout.

Compiling Your Application

After you've created an interface with Devguide, generated source code files with GXV, and created your own custom source code files, you should be sure that you're set to compile.

Note – You can use the Code Generator Tool instead.

You must have the environment variables `GUIDEHOME` and `OPENWINHOME` set to point to the Devguide and window system home directories, respectively.

Note – If you have any custom C source code files, you must edit the `Makefile` to include these files and header files in the Parameters section. Enter the names of all your C 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 filename
```

or

```
% make
```

where *filename* is either the name of your GIL file or your project file, without the `.G` or `.P` extension.

`make` 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 GXV to generate fresh source code files for any changed files. It finishes by compiling and linking all specified source code files.

GXV produces an object file with the file name you specified (*filename*). This is the file named in the `PROGRAM` parameter of the `Makefile`. You can run the compiled code by entering the file name in an open shell.

Once you have run GXV on your application file(s), you should run `make` to generate new files for any GIL files you subsequently modify. `make` then runs GXV on the modified `.G` files. Any saved changes you make to an interface using Devguide will show up in the executable file after the compile. You should never spend time altering GXV-generated files: when you use `make`, it

detects that files have changed. GXV will overwrite the files, except for the `_stubs` file`.c`. A new `_stubs.c` file is produced which preserves application code.

Using GXV++

GXV++ generates C++ code. It includes the same functionality as GXV. You first create an interface using Devguide, just as you would for GXV. GXV++ generates the same files as GXV, with C code replaced by C++ code.

You run GXV++ on *filename* as follows:

```
% gxv++ filename
```

The command line options are the same as those for GXV, except that GXV++ has no `-a` (ansi) option. GXV++ has two additional options:

- `-csuffix` <*suffixString*> – Default extension is `.cc`
- `-hsuffix` <*suffixString*> – Default extension is `.h`

GXV++ generates files with `.cc` and `.h` extensions. <*suffixString*> is the suffix you want to appear in place of `.cc` or `.h`. If your interface is named `simtool`, and you want GXV++ to generate `simtool.CC`, you type:

```
% gxv++ -csuffix CC simtool
```

This option is useful if your compiler uses a suffix for C++ source and header files different from GXV's default values.

Using Included Libraries

Devguide comes with two libraries located in the `lib` subdirectory in `$GUIDEHOME`:

- `libguide` – Contains useful functions that are independent of the windowing system
- `libguidexv` – Contains useful functions that are dependent on the XView toolkit

Source to these libraries is distributed as is, with no technical support. The source code is available in `$GUIDEHOME/src/{libguide,libguidexv,libgolit}`. The following sections describe five of the functions included in the libraries.

File Chooser (`gfm`)

The file chooser is a standard file dialog box that you can use for loading or saving files. A scrolling list of files and directories in your current directory is displayed when you open the file chooser. Double clicking on a directory will descend into that directory and display a list of the files in the directory. An *action button* is located at the bottom of the file chooser dialog box. It is typically labeled “Load” or “Save.” Double clicking on a file or typing in a file name and pressing the action button will call an application specified callback. It is then up to the application program to do whatever is necessary to load or save the file. The file chooser is fully resizable. It has a minimum size of eight characters by five lines in the scrolling list. Following are the functions used to access `gfm` from an application.

The `gfm_initialize` function initializes the Devguide file chooser.

```
gfm_popup_objects *
gfm_initialize(
    gfm_popup_objects       *ip,                /* Initial ip, NULL if none */
    Xv_opaque               owner,              /* Parent window */
    char                    *title              /* Initial window title */
                        )
```

You can pass in an initial instance pointer, NULL if a new file chooser is desired. The file chooser popup is parented off the window `owner`. NULL is a valid parent. You can specify a title for the window in the string `title`. NULL means no title at all.

The `gfm_initialize` function returns a pointer to the newly created file chooser. This is an “instance pointer.” Save this pointer; it will be needed later for other calls.

The `gfm_activate` function activates the file chooser by making it visible and bringing it to the top of the window stack.

```
void
gfm_activate(
        gfm_popup_objects       *ip,                    /* ip to a file chooser */
        char                    *directory,             /* Initial directory */
        char                    *filter_pat,            /* ed(3) filter pattern */
        int                    (*filter_callback)(),    /* Filter callback*/
        int                     (*callback)(),          /* Callback function */
        Xv_opaque               glyph,                  /* Custom glyph */
        GFM_MODE                mode                    /* Mode */
                              )
```

The instance pointer specifies which file chooser to activate. This is the pointer returned by `gfm_initialize`. You can have several file choosers in one application; the most common usage would be one for Load and one for Save. The file chooser is started in the directory `directory`. If NULL is specified then the file chooser is left in the same directory it was in the last time it was dismissed. f`ilter_pat` is an `ed(3)` regular expression that is used to filter out files. For example, to see only C source code, use `^.*\.c$`. f`ilter_callback` is described below. `callback` is a pointer to a function that is called when a file is selected by the user. A callback function must be specified; otherwise the file chooser is essentially worthless. `glyph` can specify a 16x16 XView SERVER_IMAGE that is displayed for any custom file (that is, for files that match the filter patterns). If NULL is specified then the default document glyph is used. `mode` tells the file chooser which label to put on the button. Possible values are `GFM_LOAD`, `GFM_SAVE`, `GFM_CREATE`, and `GFM_DEFAULT`. `GFM_DEFAULT` leaves the button label as it is.

The `filter_callback` function is notified every time a file is about to be put into a scrolling list.

```
int
filter_callback (
       gfm_popup_objects     *ip,       /* ip to a file chooser */
       char                  *path      /* Absolute path to file */
   )
```

It is passed two parameters: an instance pointer to the file chooser that is filling a list and the current file that is being processed. If `filter_callback` returns TRUE, the file will be added to the list. If `filter_callback` returns FALSE, the file will not be added to the list. Only files that make it past `filter_pat` are passed to this routine.

The application specified `callback` is notified when a file has been selected.

```
int
callback(
        gfm_popup_objects       *ip,         /* ip to a file chooser */
        char                    *directory,  /* Directory selected */
        char                    *file        /* File selected */
       )
```

`callback` is passed three parameters: an instance pointer to the calling file chooser, and the directory and file that were selected by the user. For example, suppose the user selects the file `/usr/guide/lib/templates/sample.G`. Then `directory` is assigned the value `/usr/guide/lib/templates`, and `file` is assigned the value `sample.G`. The instance pointer will be a pointer that was returned by `gfm_initialize`. If `callback` returns `GFM_OK`, the file chooser will be dismissed if it is not pinned. If `callback` returns `GFM_ERROR`, the file chooser will remain on the workspace.

The `gfm_show_dotfiles` function tells the file chooser referred to by the instance pointer whether or not to display "." files; for example, `.login`.

```
void
gfm_show_dotfiles(
        gfm_popup_objects     *ip,    /* ip to a file chooser */
        int                   flag    /* Boolean flag */
       )
```

The instance pointer should be a valid pointer that is returned by `gfm_initialize`. `flag` is a boolean variable. A TRUE value for `flag` means to show all hidden "." files except `"."`.

The `gfm_set_action` function sets the label on an action button.

```
void
gfm_set_action (
    gfm_popup_objects         *ip,    /* ip to a file chooser */
    char                      *label /* Label for action button */
    )
```

The instance pointer points to the file chooser whose dialog box contains this action button. The instance pointer should be a valid pointer that is returned by `gfm_initialize`. `label` is a string that will be used for the button label. This is useful if you want to have a custom label on the button.

Color Chooser (`gcc`*)*

The color chooser is a simple interface for choosing a color from the standard list of X11 colors; you use it whenever you set an element color in Devguide. The functions that follow initialize, activate, deactivate, and suspend the color chooser.

The `gcc_initialize` function initializes the color chooser.

```
void
gcc_initialize(
       Xv_opaque          owner,     /* Owner to parent gcc from */
       char               *title     /* Window frame title */
       )
```

`owner` gives the parent of the window. NULL is a valid parent. `title` specifies a title for the window; NULL means no title at all.

The gcc-activate function activates the color chooser, making it visible and bringing it to the top of the window stack.

```
void
gcc_activate(
    char       *left,               /* Left footer string */
    char       *right,              /* Right footer string */
    void       (*callback)(),       /* Callback function */
    caddr_t    client_data,         /* Client specific data passed to func */
    char       *color_name          /* Initial color */
    )
```

The left and right pop-up window footers are set to the strings in left and right respectively; NULL means no footer for that side. func is a pointer to a function that is called when the user chooses the Apply button on the color chooser. client_data contains client specific information that is passed back to the application when func is called. color_name is a string containing the initial color selected in the color chooser.

The function callback is called as follows:

```
callback(
    char           *color_name,
    caddr_t        client_data
    )
```

color_name is a string containing the name of the color that was chosen by the user. client_data is the same as the data passed to gcc_activate. It can be used to store client specific information.

The gcc_deactivate function deactivates the color chooser.

```
void
gcc_deactivate()
```

It makes the pop-up window inactive by graying out pertinent items and resetting the callback function to NULL.

The gcc_suspend function suspends the color chooser depending on the flag.

```
void
gcc_suspend(
    int        suspend            /* Boolean flag TRUE | FALSE */
    )
```

Suspending the color chooser unpins the window so that it can't be used. This is useful when an application wishes to disable the color chooser temporarily.

*Colormap Handler (*gcm*)*

The colormap handler is a simple module that allows applications to load in the standard X11 colors.

The gcm_initialize_colors function sets up the standard Devguide GXV colormap (all of the standard X11 color names) for window and all inherited items of that window.

```
void
gcm_initialize_colors(
           Xv_opaque          window,              /* Base window */
           char               *bg_color,           /* Initial background color */
           char               *fg_color            /* Initial foreground color */
           )
```

If bg_color or fg_color are not NULL, it sets that color as the initial background or foreground color respectively.

The gcm_color_index function returns the colormap index for the color name specified in color_name.

```
int
gcm_color_index(
           char    *color_name           /* Name of color to look up */
           )
```

It returns -1 if the color is not found. Please note that the Devguide GXV colormap includes all of the standard X11 color names found in the O'Reilly X books. This function is case insensitive.

The `gcm_colormap_name` function returns the name of Devguide's GXV colormap segment.

```
char
*gcm_colormap_name()
```

*Drag and Drop Interface (*gdd*)*

The drag and drop module allows an application to get information about a drop onto the application and also provides a mechanism to be the source of a drag operation. As an example, Devguide now supports drops on the Project Organizer window and on a drag and drop target located on Devguide's main window. A *drag and drop target* has a target glyph associated with it. A *drop site* is an area of your application, such as a text pane, that can accept drops.

There are three basic ways to drag data from one application to another: dragging a glyph, dragging from a drag and drop target, and selecting text and dragging using the flying punchcard method. For instance, to load a GIL file into Devguide, a user could select one or more GIL file icons from file manager or from the Mail Tool attachments window and drop them onto Devguide's drag and drop target or Project Organizer window. They could also select the text of a GIL file from a Text Editor window or a Command Tool and drag the selected text onto Devguide.

See the *XView Programming Manual* and the *OpenWindows Version 3.1 Desktop Integration Guide* or the *Desktop Integration Guide* for information on drag and drop.

Supporting Drag and Drop Operations

Devguide supports drops over three UI elements: drag and drop targets, scrolling lists, and canvases. Devguide supports dragging from drop targets. See the associated portions of the *OpenWindows Developer's Guide: User's Guide* for an explanation on how to create drag and drop targets and drop sites. Most of the code necessary for processing drops will be automatically generated by GXV. You are responsible for interpreting and using the data. For example, you

must add code to make data dropped onto a scrolling list appear in the list. Other things you will be responsible for are: keeping the drop site regions up to date, supporting multiple drop sites (on a canvas for instance), and setting up the data that will be dragged.

Following are the functions used to access the drag and drop package from an application.

The `gdd_init_dragdrop` function initializes the drag and drop package.

```
void
gdd_init_dragdrop(
           Xv_opaque      frame             /* Frame handle */
           )
```

The frame handle is used by the drag and drop module to display status and error messages in the frame's footer area. If a Devguide-generated application has drop sites or drag and drop targets, then GXV will generate this call in the `main()` function.

The `gdd_register_drop_target` function registers a drag and drop target with the drag and drop package.

```
void
gdd_register_drop_target(
           Panel_item     drop_target,              /* Drop Target handle */
           void           (*drop_callback) ()       /* Drop callback function */
           void           (*drag_callback) ()       /* Drag callback function */
           )
```

`drop_target` is the handle of a drag and drop target created by a call to `xv_create()`. `drop_callback` is a pointer to a function that is called when the drag and drop package has successfully processed a drop. A `drop_callback` function must be specified in order for an application to receive drops. If the application does not support drops, the `drop_callback` parameter must be set to `NULL`. If a drop consists of multiple items (for example, dragging more than one glyph from File Manager) then the callback will be called multiple times, once for each item dragged. `drag_callback` is a pointer to a function that is called twice. It is called once when a drag is initiated from the drag and drop target, and again when the drag is completed.

A drag_callback function must be specified for an application to support drags. If the application does not support drags, the drag_callback parameter must be set to NULL.

If a Devguide generated application has a drag and drop target, and if a drag_callback or a drop_callback is specified, then GXV will create the drag and drop target and generate this call in the _ui.c file.

The gdd_register_drop_site function registers a drop site with the drag and drop package.

```
void
gdd_register_drop_site(
    XV_drop_site        drop_site,                  /* Drop Site handle */
    void               (*drop_callback) ()          /* Drop Callback function */
    )
```

drop_site is the handle of a drop site (an XView DROP_SITE_ITEM) created by a call to xv_create(). drop_callback is a pointer to a function that is called when the drag and drop package has successfully processed a drop. A callback function must be specified in order for an application to receive drops. If a drop consists of multiple items (for example, dragging more than one glyph from file manager) then the callback will be called multiple times, once for each item dragged. If a Devguide-generated application has a drop site, then GXV will create the drop site and generate a call to gdd_register_drop_site in the _ui.c file.

The gdd_unregister_drop_site() function is used to remove a drop site or a drop target from the list of drop sites and drop targets that were previously registered with the drag and drop package.

```
void
gdd_unregister_drop_site(
           Xv_opaque      drop_site
           )
```

The application specified drop_callback is notified once for each successfully processed drop.

```
void
drop_callback(
    Xv_opaque                   item,
    Event                       *event,
    GDD_DROP_INFO                   *drop_info
    )
```

drop_callback is passed three parameters: the handle of the item, a pointer to the event, and a pointer to a data structure containing information about the drop. The drag and drop target handle is used for drag and drop targets, the panel handle for scrolling lists, and the canvas handle for canvases.

The structure will contain only information that the drag and drop package could determine from the drop; not all of the fields will be filled in and some may contain useless information. If a Devguide-generated application has a drag and drop target or a drop site, then GXV will generate this function in the _stubs.c file.

The GDD_DROP_INFO data structure is shown in Code Example 4-2 on page 73, with a description of each field provided on page 67.

Experiment with several drops from a variety of sources, such as File Manager and Mail Tool, under different situations, such as local versus remote. GXV generates a call to gdd_print_drop_info within the drop_callback function in the _stubs.c file. gdd_print_drop_info is called each time you perform a drop. This helps you determine which fields are filled into the Drop_info data structure. You can then decide how to interpret the data for use in your application. The most common use of the information received in

the drop_callback is to check for a tmpfile first, and if that does not exist, then to use the filename. See Code Example 4-1 for an example of how to use this information.

Code Example 4-1 Sample Pseudocode Using drop_callback Information

```
if (drop_info -> tmpfile && *(drop_info -> tmpfile))
{
        use the tmpfile
}
else if (drop_info -> filename && *(drop_info -> filename))
{
        use the filename
}
```

The application specified drag_callback is notified twice; once when a drag is initiated from the drag and drop target, and again after a successful drag and drop operation.

```
void
drag_callback (
        Xv_opaque               item,
        Event                   *event,
        GDD_DROP_INFO           *drop_info,
        int                     drag_state
        )
```

Drag_callback is passed four parameters: the handle of the drag and drop target, a pointer to the event, a pointer to a data structure to be filled in by the user (see Code Example 4-1), and a flag. The fields that can be filled in for dragging are filename, data_label, app_name, and data. The drag_state flag can have two values, GDD_DRAG_STARTED and GDD_DRAG_COMPLETED. The flag is used to determine if drag_callback is being called before or after the drag.

There are several ways that information can be passed to another application through a drag and drop operation. Either a file name can be used, or both a file name and a chunk of data can be used. If a file name is available, fill in the filename field of the data structure. For example:

```
drop_info -> filename = "/home/username/docs/Monthly_report"
```

Note that the file name you provide must include the file's complete path name.

To pass a chunk of data, use the data structure's `data` field. The `data_label` field can be used to assign a label to the data. The destination of the drop can then use this label to give a name to the data that it received. File Manager would use this label as the name of the new file, while Mail Tool would use the label as the name of the Mail Tool attachment. An example of using this method is:

```
drop_info -> data_label = “Monthly_report”;
drop_info -> data = Buf;              /* Buf points to data */
drop_info -> length = strlen(Buf);
```

Note – The drag and drop package frees the following structure and unlinks the temporary file after returning from the `drop_callback` function. You must copy any information that you wish to keep. After a drag operation, free any data that was filled in the `Drop_info` data structure (shown in Code Example 4-2) if it is no longer needed.

Code Example 4-2 `GDD_DROP_INFO` Data Structure

```
typedef struct Drop_info {
          char          *app_name;
          char          *data_label;
          char          *source_host;
          char          *filename;
          int            length;
          char          *data;
          char          *tmpfile;
} GDD_DROP_INFO;
```

Following is a description of the `Drop_info` variables (see Code Example 4-2):

- `app_name` – The name of the application from which the drag originated. Fill this in with your application name for a drag operation.

- `data_label` – An object's label. For instance, File Manager icons and Mail Tool attachments have data labels. This is useful for preserving the dragged data's file name. For a drag operation, fill this in to give a label to a chunk of data being passed in the data field.
- `source_host` – The host name of the machine that owns the file or data; automatically set by the drag and drop package for a drag operation.
- `filename` – For a drop, this field is set if the corresponding drag's source provides a file name. For a drag operation, fill this in if you have a file name available.
- `length` – For a drop, this is set if the data field is filled in. For a drag operation, you must fill this in with the length (in bytes) of the data, if the data field is set.
- `data` – For a drop, this is set to a pointer to the ASCII data (only needed if the dragged object does not have a file associated with it, for example, a flying punchcard drag). For a drag operation, fill this in with a pointer to any ASCII or non-ASCII data type.
- `tmpfile` – A temporary file for data that does not have a file associated with it, or is not ascii. This file will have the same information as the data field for ascii drops up to a certain size. Large drops or non-ascii drops will use the `tmpfile` exclusively.

The `gdd_activate_drop_site` function activates a drop site and assigns it its region.

```
void
gdd_activate_drop_site(
           Xv_drop_site   drop_site,                    /* A drop site handle */
           Rect           *rectlist                     /* A list of Rect structs */
)
```

A drop site is a valid region for a drop. A drop site must be registered and activated to allow drops in a particular region. If `NULL` is specified for the region, then the drop site is inactivated. The `rectlist` must be terminated by a `Rect` that has its width and/or height set to 0 (zero). The upper left hand corner of the region is relative to the owner of the drop site. That is, for scrolling lists, it is relative to the panel; while for canvases, it is relative to the canvas itself. The drop site handle is stored as `XV_KEY_DATA` on the scrolling list or canvas handle. The key is `DROP_SITE_ITEM`. If an application contains

a drop site, then GXV will generate the code necessary to create the drop site and the call to `gdd_activate_drop_site()` in the `_ui.c` file. If a panel or canvas is resizable, then `gdd_activate_drop_site()` must be called from the respective resize procedure to update the drop site region.

Note – Drag and drop targets do not need to be activated; this function is only valid for drop sites.

`gdd_activate_drop_site` would be commonly called from a canvas resize procedure. For example, if the whole canvas is a drop site, then the code would look something like:

```
Rect                  rectlist[2];
Xv_drop_site          drop_site;

drop_site = (Xv_drop_site) xv_get(canvas_paint_window(canvas),
                      XV_KEY_DATA, DROP_SITE_ITEM);

rectlist[0] = *(Rect *) xv_get(obj, XV_RECT);
rectlist[0].r_left = 0;
rectlist[0].r_top = 0;
rectlist[1].r_width = 0;
rectlist[1].r_height = 0;
gdd_activate_drop_site(drop_site, rectlist);
```

The `gdd_drop_target_notify_proc` function is attached to the drag and drop target in the `_ui.c` file automatically by GXV. It is called after the drag and drop target has been created.

```
int
gdd_drop_target_notify_proc(
    Panel_item            item,
    unsigned int          value,
    Event                 *event
    )
```

The gdd_load_event_proc function is attached to a scrolling list's panel if the list accepts drops and to the canvas if the canvas accepts drops.

```
Notify_value
gdd_load_event_proc(
    Xv_opaque             window,
    Event                 *event,
    Notify_arg            arg,
    Notify_event_type     type
    )
```

The function is attached to the panel or canvas in the ui_.c file automatically by GXV.

The gdd_print_drop_info is a convenience function to be used during development. It prints out the fields of the GDD_DROP_INFO data structure.

```
void
gdd_print_drop_info(
        GDD_DROP_INFO   *drop_info
        )
```

Using a Drag and Drop Target: An Example

This example uses a drag and drop target to drag the contents of a text pane to another application. In the text pane's event procedure, the following code keeps the drag and drop target status up-to-date:

```
int     length;

length = xv_get(ip -> textpane1, TEXTSW_LENGTH);

if (length > 0)
    xv_set(ip -> drop_target1, PANEL_DROP_FULL, TRUE, NULL);
else
    xv_set(ip_drop_target1, PANEL_DROP_FULL, FALSE, NULL);
```

Insert the following code into `drag_callback` to set up the application `name` field, the `data_label` field, and the `data` and `length` fields.

```
int     length;
char    *buf;

switch (drag_state) {
case GDD_DRAG_STARTED:

    length = (int) xv_get(ip -> textpane1, TEXTSW_LENGTH);

    if (buf)
            free(buf);

    buf = (char *) malloc((length + 1) *sizeof(char));

    xv_get(ip -> textpane1, TEXTSW_CONTENTS, 0, buf, length);

    buf[length] = '\0';
    drop_info -> app_name = "my_app_name";
    drop_info -> data = buf;
    drop_info -> data_label = "Monthly_report";
    drop_info -> length = length;

    break;

case GDD_DRAG_COMPLETED:
    break;
}
```

Note – A drag and drop target cannot be dragged from until the application has data to be dragged and `PANEL_DROP_FULL` is set to `TRUE`. The drag and drop target's Normal Glyph is not displayed unless `PANEL_DROP_FULL` is set to `TRUE`.

Group Package

The Group package is an XView extension that supports a high level layout by means of a constraint-based container. This package supports the new Devguide 3.0 grouping features on top of XView. A group is a transparent object that has members. The members of a group can be Panel_items or other

groups. You can think of the group as a constraint manager, where you specify constraints to apply to the members of the group. The group package will lay out the members of the group to satisfy those constraints.

For example, you might have a set of three buttons that you always want to be equally spaced apart. To do this, first create the three buttons. The initial position of each button isn't important because the group positions them correctly later. Then create a group with the constraints "keep these in a row, horizontally centered, horizontally spaced 10 pixels apart." The group you create lays out the buttons according to the rules you specify. The first button is kept in its current position, the second is placed 10 pixels to the right of the first, horizontally centered, and so on.

Some of the rules affect only the contents of the group. Use the anchoring feature of groups to create an external relationship between the group and some other object.

By using groups with layout constraints you are performing relative layout. That is, one object is placed in relation to another object. Under certain conditions an object's size or position can change. For example, the font or the strings on a button may change because you are running the application in another language. Relative layout eliminates problems associated with these changes. If you use absolute positions for objects then they can easily overlap, resulting in an unusable interface.

Note that the current implementation only lays out the members once, by default, at create time. If something about a member or an external anchor object changes it will not re-layout the group. In other words, this package does not support dynamic resize behavior. The group package can, however, help an application programmer achieve this goal. This will be discussed later.

A group is an XView object, created as follows:

```
Group group;

group = xv_create(panel, GROUP,
    ...
        NULL);
```

Once you have the handle to a group, call `xv_set` to change attributes or `xv_get` to retrieve values for attributes. A group can also be destroyed by calling `xv_destroy`. A call to `xv_destroy` does not call `xv_destroy` on each of the individual members of the group. You must do this before or after you destroy the group.

The group package supports four basic layout types: As Is (GROUP_NONE), Row (GROUP_ROW), Column (GROUP_COLUMN), and matrix (GROUP_ROWCOLUMN). As Is leaves the x and y locations of the members alone with respect to the upper left corner of the group. A row group will place its members in a horizontal row equally spaced apart and horizontally aligned. A column group will place its members in a vertical column equally spaced apart and vertically aligned.

Recall the example mentioned above with three buttons, horizontally centered and equally spaced. If the buttons have already been created with handles button1, button2, and button3, you can place them in a group as follows:

```
Group   button_group;

button_group = xv_create(panel, GROUP,
        GROUP_TYPE, GROUP_ROW,
        GROUP_MEMBERS,
                    button1,
                    button2,
                    button3,
                    NULL,
    GROUP_ROW_ALIGNMENT, GROUP_HORIZONTAL_CENTERS,
    GROUP_HORIZONTAL_SPACING, 10,
    NULL);
```

As another example, consider the Apply and Reset buttons that appear at the bottom of most OPEN LOOK property windows. Use the anchoring facility to center these buttons as they should appear. Create a group consisting of the two buttons. Choose the South point of the group's bounding box as the reference point, `GROUP_REFERENCE_POINT`. (A group's bounding box includes nine compass points. South is the group's bottom center point.)

Choose the South point on the panel as the anchor point. The following code places the bottom center of the Apply/Reset group 10 pixels above the panel in which it is located.

```
Group   apply_reset_group;

apply_reset_group = xv_create(panel, GROUP,
        GROUP_TYPE, GROUP_ROW,
        GROUP_MEMBERS,
                    apply,
                    reset,
                    NULL,
    GROUP_ROW_ALIGNMENT, GROUP_HORIZONTAL_CENTERS,
    GROUP_HORIZONTAL_SPACING, 10,
    GROUP_ANCHOR_OBJ, panel,
    GROUP_ANCHOR_POINT, GROUP_SOUTH,
    GROUP_REFERENCE_POINT, GROUP_SOUTH,
    GROUP_HORIZONTAL_OFFSET, 0,
    GROUP_VERTICAL_OFFSET, -10,
    NULL);
```

If you want relationships to stay up to date, or change on a resize, you need to attach an event handler. When you see a resize event, call the group package to re-establish the constraints of the group. In the Apply/Reset example above, put the following code inside an event handler for the window:

```
if (event_action(event) == WIN_RESIZE)
            group_anchor(apply_reset_group);
```

Similarly, if you want an object in the group to change size when the window is resized, use the following code:

```
if (event_action(event) == WIN_RESIZE) {
        /*
        * Change sizes of members, for example
        *               xv_set(some_list, PANEL_LIST_WIDTH, ..., NULL);
        */
    group_layout(some_group);
    panel_paint(panel, PANEL_NO_CLEAR);
}
```

The call to `panel_paint` ensures that all of the items inside the panel are painted correctly. Items can overlap while being moved; the damage needs to be repaired.

A good example of using groups to support dynamic resize behavior is the file chooser (`gfm`), also available in `libguidexv`. Rules define how different objects' sizes change when the window is resized. An example of a rule is to specify how a scrolling list grows or shrinks by an even number of rows as the parent window is resized. When a resize event is seen, these rules are applied, changing the sizes of objects, and a call is made to the group package to correctly re-layout the members. Source to the file chooser, as well as the Group package, is in `$GUIDEHOME/src/libguidexv/gfm*.[ch]`.

The Group package is a stand-alone XView extension. Source can be found in the files `$GUIDEHOME/src/libguidexv/group*.[ch]`.

Group Package Attribute Summary

`GROUP_TYPE`

Specifies the type of layout to be performed on this group. If GROUP_NONE is specified no layout will be performed; all members of the group are absolutely positioned.

Type: GROUP_TYPES
Default: GROUP_NONE
Procs: `create, set, get`

`GROUP_ROWS`

Specifies the number of rows in a group. Setting this attribute invalidates whatever is set for GROUP_COLUMNS. This fixes the number of rows. A row first fill is done and the number of columns is calculated based on GROUP_ROWS and the number of members.

Type: `int`
Default: none
Procs: `create, set, get`

GROUP_COLUMNS
Specifies the number of columns in a group. Setting this attribute invalidates whatever is set for GROUP_ROWS. This fixes the number of columns. A column first fill is done and the number of rows is calculated based on GROUP_COLUMNS and the number of members.

Type: `int`
Default: none
Procs: `create, set, get`

GROUP_HORIZONTAL_SPACING
Specifies the amount of horizontal space to be used between items in the group.

Type: `int`
Default: 10
Procs: `create, set, get`

GROUP_VERTICAL_SPACING
Specifies the amount of vertical space between items in the group.

Type: `int`
Default: none
Procs: `create, set, get`

GROUP_ROW_ALIGNMENT
Specifies a row (horizontal) alignment to be performed on members of the group.

Type: `GROUP_ROW_ALIGNMENTS`
Default: `GROUP_TOP_EDGES`
Procs: `create, set, get`

GROUP_COLUMN_ALIGNMENT
Specifies a column (vertical) alignment to be performed on members of the group.

Type: `GROUP_COLUMN_ALIGNMENTS`
Default: `GROUP_LEFT_EDGES`
Procs: `create, set, get`

GROUP_MEMBERS
Specifies the list of members to be in the group as a list of handles. A group may contain Panel_items or other groups. A group must contain one or more members. If only one member exists then only the attributes related to anchoring apply.

Type: List of `Xv_opaque`
Default: `NULL`
Procs: `create, set, get`

GROUP_MEMBERS_PTR
Specifies the list of members in the group specified as a pointer to an array of handles. A group may contain Panel_items or other groups. A group must contain one or more members. If only one member exists then only the attributes related to anchoring apply.

Type: `Xv_opaque *`
Default: `NULL`
Procs: `create, set, get`

GROUP_ANCHOR_OBJ
Specifies an object to anchor this group to. This can be another Group, Panel_item, or even the Panel that the group is located in.

Type: `Xv_opaque`
Default: `NULL`
Procs: `create, set, get`

GROUP_ANCHOR_POINT
Specifies the point on the anchor object to which the anchor is attached.

Type: `GROUP_COMPASS_POINTS`
Default: none
Procs: `create, set, get`

GROUP_REFERENCE_POINT
Specifies the point on the group to which the anchor is attached.

Type: `GROUP_COMPASS_POINTS`
Default: none
Procs: `create, set, get`

GROUP_HORIZONTAL_OFFSET
Specifies a horizontal offset between GROUP_ANCHOR_POINT on GROUP_ANCHOR_OBJ and GROUP_REFERENCE_POINT on the current group. This is the horizontal component of the vector between the two points. Positive integers are to the right, negative to the left.

Type: int
Default: 10
Procs: create, set, get

GROUP_VERTICAL_OFFSET
Specifies a vertical offset between GROUP_ANCHOR_POINT on GROUP_ANCHOR_OBJ and GROUP_REFERENCE_POINT on the current group. This is the vertical component of the vector between the two points. Positive integers are below, negative above.

Type: int
Default: 10
Procs: create, set, get

GROUP_PARENT
Used to retrieve the parent group for a group. Returns NULL if this is a top level group. Otherwise returns a handle to the group that is the parent.

Type: Group
Default: none
Procs: get

GROUP_LAYOUT
Specifies whether layout is performed for this group.

Type: Boolean
Default: TRUE
Procs: create, set, get

GROUP_LAYOUT can be used in two ways:

- As a form of batching. If you need to change many things about the group, you can set this to FALSE before making changes. Set it back to TRUE when you are done to force a re-layout. For example:

```
xv_set (group, GROUP_LAYOUT, FALSE, NULL);
    /* Make changes to group here */
xv_set (group, GROUP_LAYOUT, TRUE, NULL);
```

- To re-layout the group.

```
xv_set (group, GROUP_LAYOUT, TRUE, NULL);
```

Both of the above methods are typically done inside an event routine upon receiving a WIN_RESIZE.

GROUP_REPLACE_MEMBER

Replace a given member with a new member. Useful if you want the group to still behave properly when you need to destroy a member and replace it with a new version.

Type: Xv_opaque, Xv_opaque
Default: none
Procs: set

A common case here is when you destroy a scrolling list and then recreate it again with new items. If you don't tell the group package you have destroyed and replaced the list it will contain a handle to the old list that no longer exists. For example:

```
old_list = list;
xv_destroy(list);
list = xv_create(panel, PANEL_LIST, ...,NULL);
xv_set(group, GROUP_REPLACE_MEMBER, old_list,
                              list, NULL);
```

XV_X
Set the X position of the upper left corner of the group.

Type: `int`
Default: none
Procs: `create, set, get`

XV_Y
Set the Y position of the upper left corner of the group.

Type: `int`
Default: none
Procs: `create, set, get`

XV_SHOW
Show or hide all members of a group simultaneously. This can be used to easily show or hide an entire set of Panel_items with one call. To hide an entire group, call:

xv_set(group, XV_SHOW, FALSE, NULL);

Type: `Boolean`
Default: `TRUE`
Procs: `create, set, get`

PANEL_INACTIVE
Make all of the members of a group active or inactive. This can be used to easily activate or deactivate an entire set of Panel_items with one call. For example to inactivate an entire group, call:

xv_set(group, PANEL_INACTIVE, TRUE, NULL);

Type: `Boolean`
Default: `TRUE`
Procs: `create, set, get`

Procedures and Macros

`group_layout`
Layout a group with the current layout constraints. This has the same effect as setting GROUP_LAYOUT to TRUE.

```
group_layout(
        Group   group
        )
```

`group_anchor`
Re-anchor a group according to the current anchor and offset. Call this if something changes about the GROUP_ANCHOR_OBJ and you would like to have a group moved to reflect the new relationship.

```
group_anchor(
        Group   group
        )
```

Data Types

GroupHandle to an opaque structure that describes a group.

`GROUP_TYPES`	Enumeration:
	`GROUP_NONE`
	`GROUP_ROW`
	`GROUP_COLUMN`
	`GROUP_ROWCOLUMN`

`GROUP_COLUMN_ALIGNMENTS`	Enumeration:
	`GROUP_LEFT_EDGES`
	`GROUP_LABELS`
	`GROUP_VERTICAL_CENTERS`
	`GROUP_RIGHT_EDGES`

`GROUP_ROW_ALIGNMENTS`	Enumeration:
	`GROUP_TOP_EDGES`
	`GROUP_HORIZONTAL_CENTERS`
	`GROUP_BOTTOM_EDGES`

GROUP_COMPASS_POINTS	Enumeration: GROUP_NORTHWEST GROUP_NORTH GROUP_NORTHEAST GROUP_WEST GROUP_CENTER GROUP_EAST GROUP_SOUTHWEST GROUP_SOUTH GROUP_SOUTHEAST

Additional Programmer's Tips

When you customize GXV-generated source code to tie interface code together with your own custom source code, consider these tips to make your job easier.

Using Stubs Merge

Each successive run of GXV merges the user-added code from the previous version of the `_stubs.c` file to the new `_stubs.c` file. This is done so that you can add application code directly to the stubs file without having to worry about your code being overwritten by the next run of GXV.

The Mechanism

When GXV generates the `_stubs.c` file, it first checks to see if there is an existing `_stubs.c` file with the same name in your current directory. If found, it then makes a backup copy of this existing `_stubs.c` file and names it `_stubs.c.BAK`. By doing this, any code added to the original `_stubs.c` file is saved and retained in this backup file. GXV then proceeds to generate a new `_stubs.c` file. This new `_stubs.c` file reflects all the changes made to the `.G` file since the last run of GXV. If a callback already exists in the old `_stubs.c` file, it is not modified during stubs merge. In this way customized _stubs.c files are saved. After this is done, GXV merges this new `_stubs.c` file with `_stubs.c.BAK`. The result is a new version of the `_stubs.c` file with all the user code intact.

Potential Problems

GXV merges two `_stubs.c` files by inserting code from one file into the other. It does not remove or replace code from a file. This could create problems in the following two cases:

- Removing (renaming) Top Level Objects – Removing (renaming) top level objects (windows or menus) in Devguide will not cause GXV to remove (rename) the objects' definitions and initializations from the `_stubs.c` file. You must manually remove or rename definitions and initializations in the `_stubs.c` file before you compile the program.
- Removing (renaming) Callbacks – If a callback is removed (renamed) in Devguide, the callback's function definition will still remain in the `_stubs.c` file. You must alter the `_stubs.c` file accordingly.

`-n` *(No Merge) Option*

GXV's `-n` option turns off the stubs merge capability. If you are prototyping an interface and have never added code to the `_stubs.c` file, run GXV in `-n` mode. You will always obtain a new copy of the `_stubs.c` file. If you add code to your `_stubs.c` file, do not use the `-n` option. GXV merges the user-added code into the newly generated `_stubs.c` file.

Enabling and Disabling Menu Items

There are many times in a user interface where menu items must be enabled or disabled depending on conditions within the program. For example, the Cut item in the Edit menu can be disabled if there is no text selected to cut.

You can use a CallFunction connection to enable or disable a menu item. When GXV is run, it includes a function in the `_stubs.c` file for the CallFunction connection. Place your code in the CallFunction's MENU_DISPLAY case to enable or disable a menu item. This case is called after the menu is requested to be displayed, and before the it appears on the screen.

Code Example 4-3 shows a CallFunction callback filled in with four lines of custom code, under `case MENU_DISPLAY`, that disable a menu item.

Code Example 4-3 Disabling a Menu Item

```
/*
 * Menu handler for 'file_menu (Load...)'.
 */
Menu_item
g_file_load(
        Menu_item       item,
        Menu_generate   op
        )
{
    switch (op) {
    case MENU_DISPLAY:
            if (item_is_inactive)
                    xv_set(item, MENU_INACTIVE, TRUE, 0);
            else
                    xv_set(item, MENU_INACTIVE, FALSE, 0);
            break;

    case MENU_DISPLAY_DONE:
            break;

    case MENU_NOTIFY:
            break;

    case MENU_NOTIFY_DONE:
            break;
    }
   return item;
}
```

Tying a Pop-Up Window to a Button

You can tie a pop-up window to a button very easily using connections; you do not need to add any application code. Establish a connection with the button as Source and pop-up as Target. Choose Notify from the When menu and Show from the Action menu. For more information on connections, see "Handing Events Using Connections" in the *OpenWindows Developer's Guide: User's Guide.*

Files Provided with GXV A

The Devguide software includes a set of files useful to Devguide and GXV users. These files are installed, along with Devguide and the code generators, on your workstation or server. You'll find the files in separate subdirectories in Devguide's home directory.

`bin` *Subdirectory*

The `$GUIDEHOME/bin` subdirectory contains the executable files for Devguide, GXV, GXV++, and `gmomerge` (as well as for GOLIT).

`demo` *Subdirectory*

The `$GUIDEHOME/demo` subdirectory has an `xview` subdirectory, that contains three files:

- `rasview` – A sample program to display images from raster files
- `tree` – A fractal tree drawing program
- `psdraw` – A PostScript drawing program that uses `-lxvps`

The `$GUIDEHOME/demo`/gnt subdirectory contains the following subdirectories:

- `doc` – Documentation in PostScript format
- `man` – `gnt man` page

A

`include` *Subdirectory*

The `$GUIDEHOME/include` subdirectory contains header files that can be included in the C source code for a window-based program using GXV. These files offer different useful components of Devguide's own interface for programmers to use in their own programs. They are:

- `gcc.h` – Color chooser used to select colors for UI elements
- `gcm.h` – Color manager used to keep track of colors set for different UI elements
- `gdd.h` – Drag and drop controller used to display the text contents of icons dropped into a window
- `gfm.h` – File chooser header file utility routines
- `gfm_ui.h` – File chooser header file
- `gio.h` – Input/output routines used to read and write GIL files
- `group.h` – Header file for the XView group extension
- `guide.h` – Routines that define elements and functions
- `gutil.h` – Double-click support

These files are all supplied "as is" and are not supported.

`lib` *Subdirectory*

The `$GUIDEHOME/lib` subdirectory contains two libraries of C modules that you can link into other C programs. The `libguide.a` library contains routines that are independent of the window system and the `libguidexv.a` library contains routines that depend on the XView toolkit. The modules in these libraries correspond to the components offered in the `include` subdirectory.

`libgolit.a` is the runtime library for GOLIT that provides facilities for you to create user interfaces described by GOLIT-generated C code.

In addition, `lib` contains the `XView.config`, and `OLIT.config` files. These files are used to configure the Connections Manager window for the three toolkits. They list the events, actions, and targets that are permitted in the XView, and OLIT toolkits, respectively, for each possible source object.

lib also contains a help subdirectory that includes help files for Devguide, a locale subdirectory to be used for localization, and a templates subdirectory described below.

lib/templates *Subdirectory*

The templates subdirectory contains sample GIL files. An OPEN LOOK property window and OPEN LOOK application window created in properties.G and sample.G can be used as templates in your application to get you started. The sample application window contains File, View, and Edit menu buttons. controls.G contains examples of UI elements you can create using Devguide.

man *Subdirectory*

The $GUIDEHOME/ man subdirectory contains man pages for Devguide's accompanying XView code generator, GXV. To see them when you use the man command, you can append $GUIDEHOME/man to the MANPATH variable to look in this directory when you use the man command. You can also use man -M path to view man pages that are not in the standard path. The -M option overrides MANPATH. For more information, see the man(1) man page.

src *Subdirectory*

The $GUIDEHOME/src subdirectory contains source code for the libraries included in the lib subdirectory. This source code is supplied "as is" and is not supported. You can port the source to other platforms.

doc *Subdirectory*

The $GUIDEHOME/doc subdirectory contains miscellaneous documentation about Devguide. The GIL syntax file gil_syntax.doc and project syntax file project_syntax.doc can be found in this subdirectory.

A

Internationalization B

Thinking about software from an international perspective means considering a larger set of options and constraints than you may be used to.

To internationalize your software means to include nothing specific to your language and culture and to provide features that facilitate translation of text into other languages. *Internationalization* is the process of making software portable between languages or regions, while *localization* is the process of adapting software for specific languages or regions. Internationalization and localization go hand-in-hand. Internationalization is usually performed by the software manufacturer; localization is usually performed by software experts familiar with the specific language or region, called a *locale*.

Devguide provides some assistance in internationalizing and localizing applications.

This appendix covers:

- Levels of Internationalization
- Basic Internationalization Tools
- Internationalization Using Devguide
- Localization Using Devguide
- Other Internationalization Assistance

B

Levels of Internationalization

The four levels of software internationalization discussed below are in increasing order of complexity, that is, software that complies with the specifications in level 1 is easiest to achieve; level 4 is the most difficult to achieve. However, compliance at a higher level does not mean automatic compliance at a lower level. In fact, compliance at level 4 usually means that you lose level 1 compliance (see the following descriptions of the levels).

Level 1—Text and Codesets

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 and, historically, programmers may have used the eighth bit to store information about the character. The ISO Latin-1 codeset requires all 8 bits for the character.

Level 2—Formats and Collation

Many different formats are employed 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.

For additional information on internationalization, refer to the *Internationalized XView Programming Manual a*nd the *Developer's Guide to Internationalization*.

Basic Internationalization Tools

Text Databases (Text Domains)

To facilitate internationalization of text, a process exists (see `gettext()`, later in this section) to collect all text visible to the user into a file called a *portable object* file. The portable object file contains the native language strings from a program and placeholders for a localizer to put each string's translation.

When translation is completed, the localizer runs a utility against the portable object file, producing a binary *message object* file, also called a *text domain*. An application scans this message object file, or text domain, to retrieve text in the local language.

It is possible to specify more than one text domain. For example, a developer may want to organize an application's error messages in one text domain and its interface labels in another text domain. The developer gives each domain a unique name. If you want more than one text domain, you create more than one portable object file. Each portable object file contains strings from one text domain.

B

`gettext()` *and* `dgettext()` *Routines*

Two similar routines are available to a developer for retrieving translated text. One, `gettext()`, assumes a text domain has already been specified by a call to a function called `textdomain()`. For example:

```
textdomain ("domain_name");
.
.
gettext("Message_1\n");
gettext("Message_2\n");
```

The second, `dgettext()`, allows the developer to specify the domain as one of the parameters to the call. For example:

```
dgettext("domain_name", "Message_1\n");
dgettext("domain_name", "Message_2\n");
```

In both of these examples, `domain_name` is the name of a message object file and `Message_1` and `Message_2` are the native language strings for which a translation will be retrieved.

Portable Object and Message Object Files

A portable object file is a file containing all the text from an application that needs translating for users in other locales. Examples of text that *do* need translating are object labels and error messages. Examples of text that *do not* need translating are commands (like `sort`), file names, and messages used internal to a program for debugging. Portable object files are shipped to localizers for translation into local languages.

An example portable object file before translation might look like this:

```
domain "guide_notices"
msgid "Cannot open %s: %s\n"
msgstr
msgid "Continue"
msgstr
msgid "Connect"
msgstr
```

After translation (localization) the same file would look like this:

```
domain "guide_notices"
msgid "Cannot open %s: %s\n"
msgstr "No es posible abrir %s: %s\n"
msgid "Continue"
msgstr "Sigue"
msgid "Connect"
msgstr "Conectar"
```

A message object file, which contains application text strings and their translations, is a binary version of a portable object file.

`xgettext` *and* `msgfmt` *Utilities*

Once Devguide or a developer has inserted `gettext()` function calls around all user-visible text in an application, `xgettext` can be run on the source files to produce the portable object files.

When the portable object files have been created and contain the translations, the `msgfmt` utility converts portable object files (*filename*.po) into message object files (*filename*.mo).

Internationalization Using Devguide

Devguide aids the developer's internationalization efforts in two important ways: by collecting interface text strings for later translation and by providing a tool for automatic resizing and repositioning of user interface elements after their text has been translated.

B

Text Translation

When you create a user interface using Devguide, Devguide saves all interface element specifications in one or more `.G` files.

If you then use the GXV code generator to generate the C source code, the interface's text strings get linked with the `gettext()` or `dgettext()` function call. These calls are used to look up the strings in other languages, or locales.

Once the internationalization "hooks" are in the source code, you run a utility (`xgettext`) against the source to produce a specially-formatted ASCII file (the portable object or `.po` file). This ASCII file contains the original text strings and placeholders for their translations. The localizer translates the strings into the local language, then, using another utility (`msgfmt`), creates a binary message object file (also called a text database or text domain) from the portable object file.

Size and Position of Elements

Two mechanisms exist for positioning objects on the screen. The most general method is to use Devguide's relative layout or "grouping" feature, in which object positions are based on other objects.

Devguide's grouping feature enables an interface designer to select objects and designate them as a group. A group is a collection of user interface objects that is treated as a single unit. A group can be copied, moved, or deleted as a single entity. Objects included in a group maintain a physical relationship with other objects in that group. When the group is moved, or if an object in the group changes size, the relative positioning of the individual objects remains the same. This is especially important for localizing applications when button sizes may change due to different font sizes or languages.

Another method for positioning objects is available if you choose to explicitly position and size the objects making up your interface. This method uses an X windows-style resource database. The application retrieves the size and position information of the objects making up the application's interface from the resource database file (`.db` file).

If you use a relative layout scheme in which objects are positioned relative to other objects, you shouldn't need to use a size and position database. If you explicitly specify *x,y* coordinates for your interface objects, you will need to use the size and position database.

The code generator generates function calls and attributes for the interface that allow an application to access the locale-specific databases. The code generator can also generate the X resource database (containing size and position information) for the developer, if desired.

Options to GXV

Several options to GXV assist the developer with internationalization. These are `-g`, `-d`, `-h`, `-r`, `-p`, and `-x`. You can find the full `GXV(1)` man page in `$GUIDEHOME/man/man1`. See "Using Flags with GXV" on page 52 for further discussion of GXV options.

`-g`

Enables `gettext()` use. GXV-generated code will have `dgettext()` wrappers around all XView object labels (for example, `XV_LABEL` and `PANEL_CHOICE_STRINGS`) and a call to `bindtextdomain()`. One of the parameters to `dgettext()` and `bindtextdomain()` is the domain name. GXV creates the domain name by using either the interface name stripped of the `.G` suffix or the name specified with the `-p` option (see below). In either case, `_labels` is appended to the string to create the domain name.

`-d` *domainname*

Enables you to specify a domain name for the `dgettext()` function. Using the `-d` option overrides the default domain name of *interface*`_labels`.

domainname with `_labels` appended to it will be used as the domain name in `dgettext()` and `bindtextdomain()` calls. For example:

```
dgettext("domainname_labels", "Hello World");
```

or

```
bindtextdomain("domainname_labels", "pathname");
```

Both of these functions are useful for applications that consist of several GIL files. The user could organize all text strings in the same domain and place all resource information in a single file.

-h

Sets GXV to generate only the help text (`.info`) files. When used in conjunction with any of the `-s`, `-v`, `-p`, and `-x` options, GXV generates help text files but no C source code. If `-h` is specified along with any GXV option besides those just mentioned, help text files are generated along with the rest of the source code. This option is helpful for localization.

-r

Enables resource database access. GXV-generated code will include the XView attributes `XV_USE_LOCALE`, `XV_LOCALE_DIR`, `XV_INSTANCE_NAME`, and `XV_USE_DB`. These attributes are set by GXV, but an application uses them only if it uses explicit positioning of user interface objects. The attribute values are set as follows:

`XV_USE_LOCALE`, `TRUE`,

Set in `xv_init()`. `XV_USE_LOCALE` tells XView that international character strings will be displayed and/or used as input.

`XV_LOCALE_DIR`, `"."`,

Set in `xv_init()`. XView looks in the current directory for the path leading to the resource database (`.db`) file, that is, `./`<*current_locale*>`/app-defaults`. For example, for the Japanese locale, the path would be `./japanese/app-defaults/filename.db`. The current locale is determined by various means (see the section on "Locale Setting" in the *Internationalized XView Programming Manual*). In this appendix, the term <*current_locale*> refers to the locale, as set by any method.

XV_INSTANCE_NAME

Give each object a unique name.

XV_USE_DB

Lists the attributes for each object which are to be looked up in the resource database. See the *Internationalized XView Programming Manual* for a list of customizable attributes.

`-p` *app_name*

The `-p` option to GXV performs several functions. One result of using `-p` is that *app_name* is used as the first component of each line in the resource database, when used in conjunction with the `-x` option.

Note – If you run `gxv` using the `-x` and `-p` options together, *app_name* must be the name specified in the `PROGRAM` parameter of the `Makefile`. This name serves as a prefix for resources in the `.db` file.

For example, if an interface called `timezone.G` were part of a larger application, say `clock`, running:

```
example% gxv -x -p clock timezone.G
```

would produce in the `.db` file:

```
clock.*.win1.xv_width: 612
```

Note that `clock` is the `PROGRAM` name. If the interface name were simply `timezone.G` instead of `clock`, then running:

```
example% gxv -x timezone.G
```

(without the `-p` option) would generate lines in the `.db` file something like:

```
timezone.*.win1.xv_width: 612
```

Note – You can use the Code Generator Tool instead of typing the command-line options shown here.

-x

Create an X Windows-style resource database file. The file will have the name of the application with `.db` appended to it. If the project is composed of multiple GIL files, it is up to the developer to append all of the `.db` files associated with each GIL file into one single `.db` file. This can be done by adding the appropriate statement to the GXV-generated `Makefile`.

This option is needed only if the application uses explicit positioning of objects.

From Internationalization to Localization

Most of the time the internationalizer passes to the localizer:

- Application in binary form
- Application's GIL file(s)
- Portable object file(s)
- Resource database file (if used)
- Help text files

Localization Using Devguide

Devguide aids the developer's localization efforts by providing separate files containing application text strings for easy translation. The portable object files contain the application's native language text strings and placeholders for their translations.

Spot help text files are created automatically by GXV, one help file for each `.G` file. For each user interface object that a developer writes help for, GXV creates an entry in the help (`.info`) file. A developer can either write extensive help text or deliver a help file to a writer who can write and edit help text in the native language. The localizer can then localize the spot help text.

Options to GXV

GXV provides the `-h` option to generate only the help files. The help text can be translated by the localizer, without searching through the application source code. The following is an example of how to generate the help files:

```
example% gxv -h my_tool.G
gxv: reading my_tool.G
gxv: writing my_tool.info
example%
```

The help text can be translated using any text editor that supports both the native and foreign language.

Other Internationalization Assistance

Positioning Objects Explicitly

If you position user interface objects explicitly (by specifying *x*,*y* coordinates on the screen), you need to modify your application to access a size and position database (a `.db` file). When you translate the application's text for a different locale, you would need to manually experiment with new sizes and positions of the objects after translation, then re-layout your interface using the new object sizes and positions.

`gmomerge` *Utility*

If you do not use the grouping feature of Devguide, after the strings have been translated, you run a utility called `gmomerge`, in `$GUIDEHOME/bin`, to merge the translated text back into an existing English `.G` file. This merged `.G` file can be read into Devguide (displayed in the new language) and the interface can be laid out according to the new element sizes and positions. Once the new interface is laid out properly, it can be written out as a new `.G` file. This new file now contains the translated strings as well as the size and position information. It is this new `.G` file that is used to build the size and position database for a specified locale. This process allows the application to be shipped in local languages. Only one binary is needed; locale-specific databases are shipped for strings and, possibly, size and position information.

B

Conclusions

For Devguide to assist your internationalization work, use the group feature to lay out user interface objects relative to one another. Also use the internationalization option to GXV (`gxv -g`) to insert `dgettext()` function calls into your source code to facilitate string translation.

Index

D

E

F

G

H

I

K

L

M

O