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.**

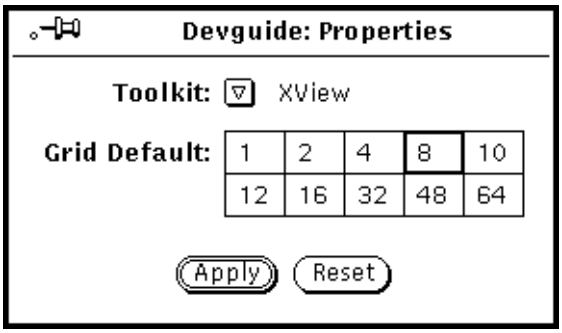

Figure 2-1 The Devguide Properties Window

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.

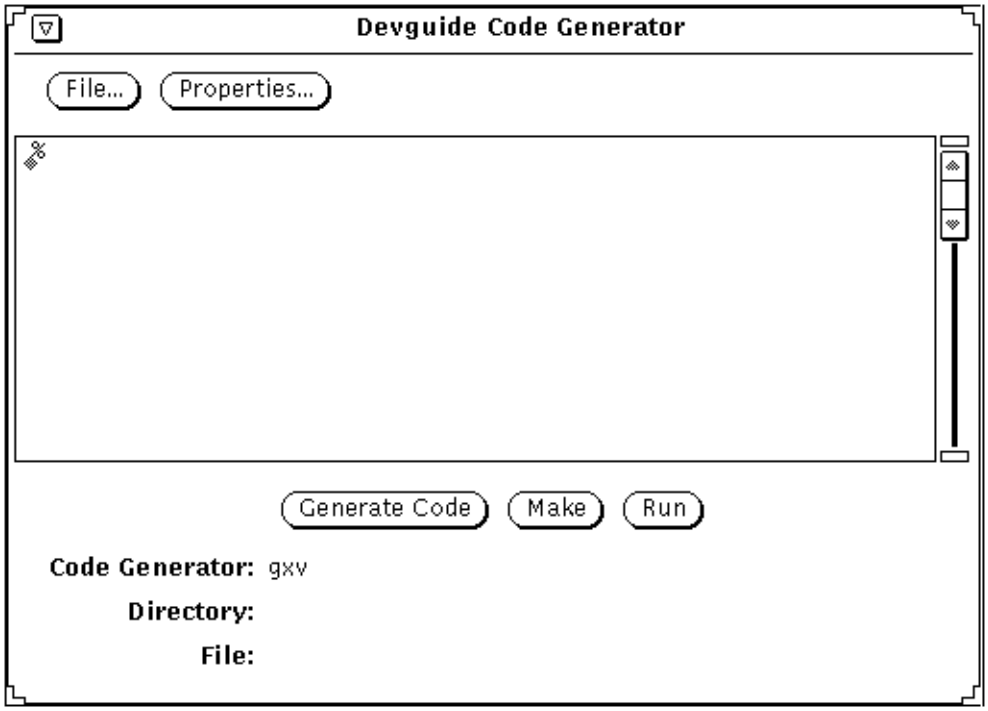

Figure 2-2 The Code Generator Tool

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.

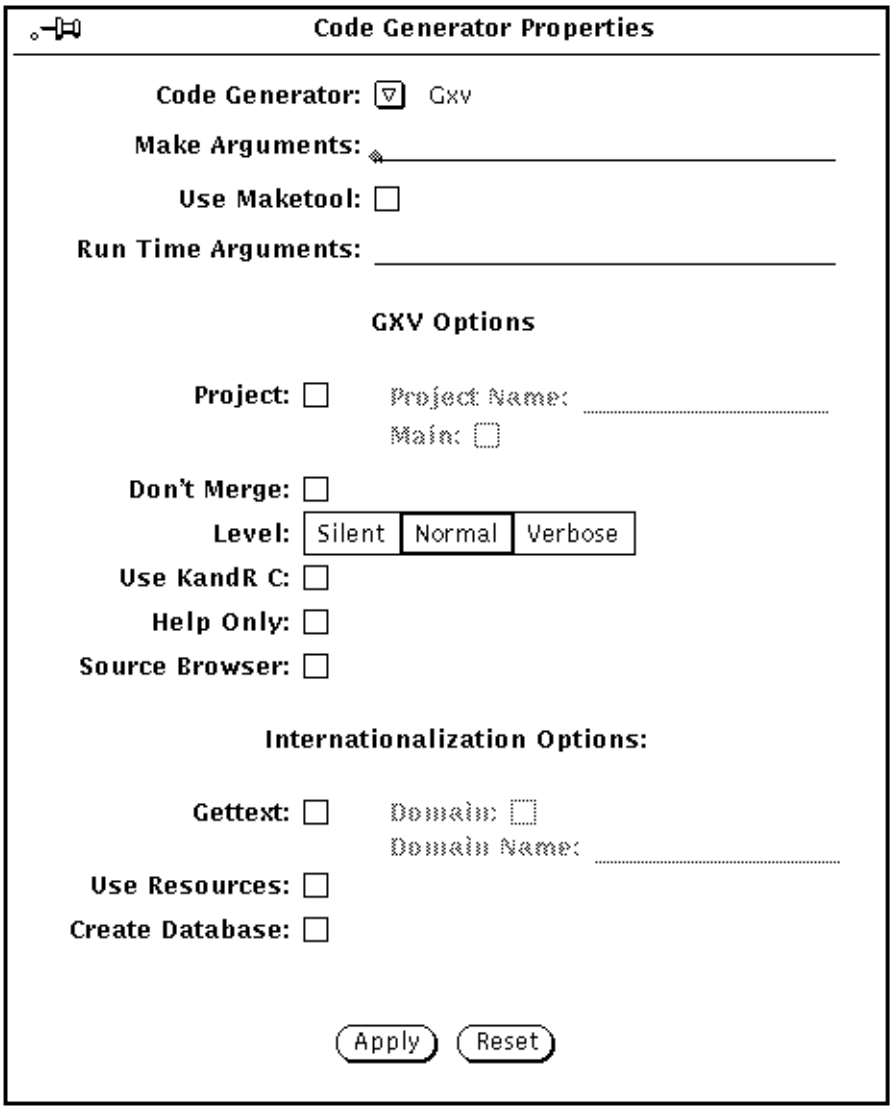

Figure 2-3 GXV Code Generator Properites window

- **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.

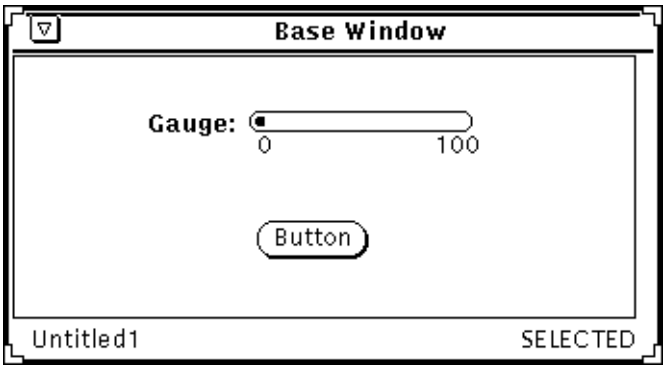

Figure 2-4 Example Interface as it Appears in Devguide

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.

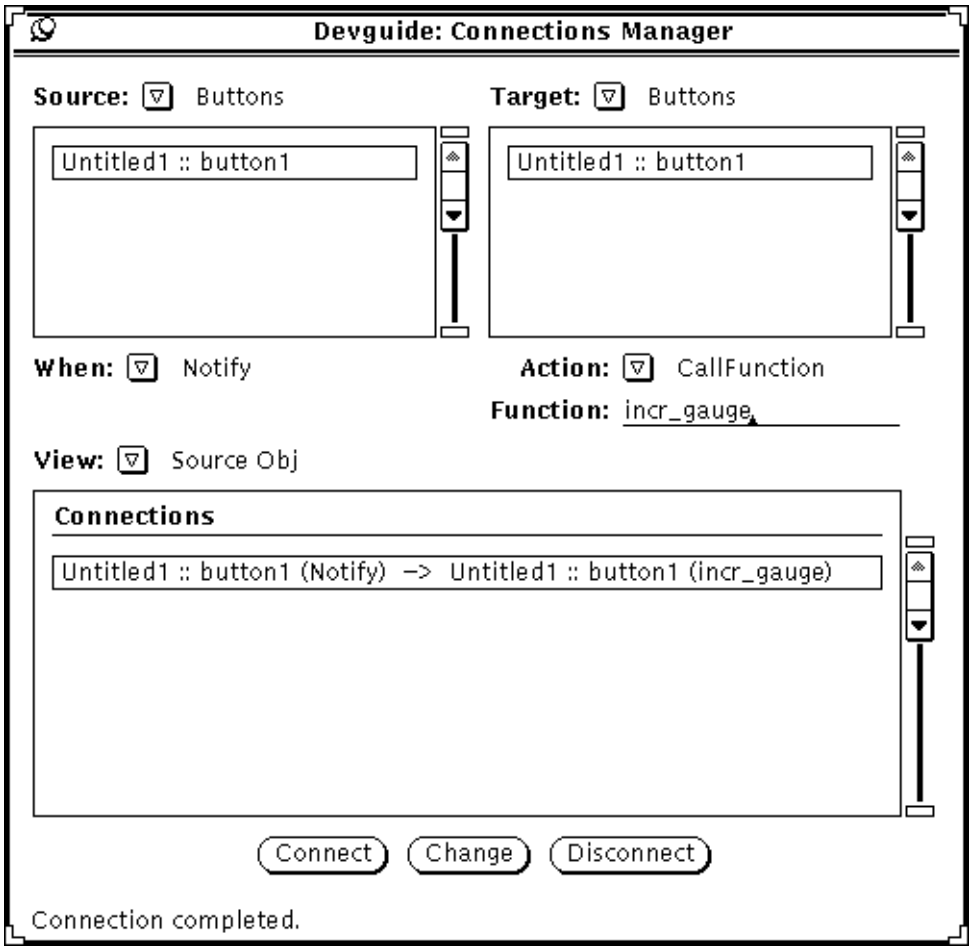

Figure 2-5 CallFunction Connection for Button

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 */
```