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.

4

`_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. filter_pat is an ed(3) regular expression that is used to filter out files. For example, to see only C source code, use ^.*\.c$. filter_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*.