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.