Internationalizing OpenWindows 5

For the first time in Solaris, Asian OpenWindows has been integrated into mainstream OpenWindows 3.3. As a result, international software no longer requires separate releases for Asia; it can be released as a single binary. Both the XView and OLIT toolkits support level-3 and level-4 internationalization, providing both messaging and multibyte characters.

For information on international XView, see the internationalization portions of the *XView Developer's Notes*.

For information on international OLIT, see the internationalization chapter of the *OLIT Reference Manual*.

Note that 8-bit characters for western European languages no longer appear by default; `LC_CTYPE` must be set to `iso_8859_1` first. East Asian languages all require their own Feature Package, which includes appropriate font sets, input manager server, and translated messages.

You may want to develop applications with OpenWindows Developer's Guide 3.3, a window tool designed specifically for building user interfaces. Usually DevGuide is available about one month after the associated release of OpenWindows. DevGuide 3.3 offers many aids for localizing as well as internationalizing applications. Specific approaches are discussed in the Internationalization appendix of the *DevGuide 3.3 User's Manual*. Briefly, GXV— DevGuide's XView code generator—helps you to internationalize your application by placing `dgettext()` function calls around strings associated with the user interface. Furthermore, GXV has an option for adding the object size and position attributes discussed in the next section.

Window systems have to take care of three things that operating systems don't have to worry about: object layout, input method, and font handling. The following sections describe how applications might deal with these issues.

Window Object Layout

Window objects include push-pins, popup menus, control buttons, and scrollbars. Layout of objects containing strings (such as menus and buttons) can change because the dimensions of those strings often change when switching languages. For example, a control button in English is probably shorter than the equivalent in German. Also, a control button in Japanese is probably taller than the equivalent in English. Even in the same language, control buttons can grow and overlap when the font size increases. In XView, here's how you would determine the length and height of a text string:

```
extern Xv_Font font;
Font_string_dims dims;
(void) xv_get(font, FONT_STRING_DIMS, "string", &dims);
```

The `dims` structure now contains the string's width as its first element, and the string's height as its second element.

XView application layout may be *implicit* or *explicit*. Implicit layout relies on the toolkit to position objects. Explicit layout means that the programmer specifies pixel values (or row and column values) both for position (x, y) and dimension (width, height, and gaps).

If you create a series of controls without specifying location information, the toolkit will position items based on its built-in spacing rules. Implicit layout will very likely make localization easier, since control objects will often be repositioned for you as required. So if you have time, try using implicit layout whenever possible. DeskSet 3.3 tools use a custom library for implicit layout.

The XView toolkit provides a limited implicit layout facility. Basically, control area objects such as buttons are positioned based on default spacing rules. However, if button width increases so that the right-most button is outside the control area, the toolkit wraps that button to the next line, rather than expanding the control area. DeskSet 3.3 tools such as Mail Tool and Calendar Manager re-layout gracefully because developers have added code to properly center and size objects, based on current window size as well as font size.

Object Size and Positioning

International XView contains attributes to allow run time configuration of the size and position of objects making up an application's interface. Size and positioning information is retrieved from an Xwindows-style resource database. The attributes associated with sizing and positioning objects are:

`XV_INSTANCE_NAME`
: Provides a mechanism to give each XView object a unique name, so that it can be queried from the database.

`XV_USE_DB`
: Allows you to specify a null-terminated list of attributes value pairs. Each attribute's value will be looked up in the database; if not found, the specified value is used as the default.

`XV_LOCALE_DIR`
: An `xv_init()` attribute, tells XView where the `app-defaults` directory is located. The resource database file resides in a directory called `app-defaults`. Like the scheme used by the message object files, the parent of the `app-defaults` directory is the name of the locale. The name of the resource database file is the name of the application with a `.db` suffix appended to it.

To illustrate the use of these attributes, see how the following portion of code from the file `my_app.c` is modified:

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);
frame = xv_create(XV_NULL, FRAME, XV_NULL);
control = xv_create(frame, PANEL, XV_NULL);
button = xv_create(control, PANEL_BUTTON,
        PANEL_LABEL_STRING, "Quit", XV_X, 10, XV_NULL);
```

Assuming the resource database (`.db`) file for the Spanish locale resides in `/home/locale/spanish/app-defaults`, here's how to modify the code:

```
bindtextdomain("my_app_strings", "/home/locale");
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv,
        XV_LOCALE_DIR, "/home/locale",
        XV_USE_LOCALE, TRUE,
0);
frame = xv_create(XV_NULL, FRAME,
        XV_INSTANCE_NAME, "base_frame",
XV_NULL);
control = xv_create(frame, PANEL,
        XV_INSTANCE_NAME, "control1",
XV_NULL);
button = xv_create(control, PANEL_BUTTON,
        XV_INSTANCE_NAME, "button1",
        PANEL_LABEL_STRING, dgettext("my_app_strings", "Quit")
        XV_USE_DB,
        XV_X, 10, XV_Y, 10, XV_NULL,
XV_NULL);
```

When the application runs, it builds a path using the directory pointed to by `XV_LOCALE_DIR`, the name of the current locale, and `app-defaults`. This path is used to search for the resource database (`.db`) file. Any attribute specified in the `XV_USE_DB` attribute list, whose parent is given a name with `XV_INSTANCE_NAME`, will have its valued looked up in the file. If the attribute is found, the value in the database is used; if the attribute isn't found, the value specified in the source code is used.

For the example above, the entries in the resource database file `my_app.db` would look like this:

```
my_app.base_frame.control1.button1.xv_x: 10
my_app.base_frame.control1.button1.xv_y: 10
```

If the layout has changed for a specific locale, an interface object may need to be repositioned. For example, let's say that for the Spanish locale, the interface looks more presentable if the above button is positioned at position 20,11 rather than position 10,10. All the developer needs to do is to change the *x* and *y* values in the `.db` file and rerun the application (without recompiling!). In this case, the above database lines would be changed to:

```
my_app.base_frame.control1.button1.xv_x: 20
my_app.base_frame.control1.button1.xv_y: 11
```

The button would then appear at position 20,11.

Input Method

Phonetic-based languages such as English are easy to type on a keyboard. However, typing is much more complicated for East Asian languages based on ideograms. When entering Japanese, for example, the typist must select between Kanji, Hiragana, Katakana, and Romaji (Roman letters used phonetically to produce Japanese words). There must also be a way to convert phonetic characters into Kanji. Input is further complicated because a given syllable can frequently be represented by several different Kanji characters—the user must select the Kanji character that is appropriate for the specific context. Here's how you'd type Japanese under OpenWindows with a Japanese keyboard.

1. Set the `LANG` environment to Japanese.
2. The initial input mode simulates a Type-4 or a PC-AT101 US keyboard. To switch input modes, press the Nihongo (Japanese) key.
3. Type words using phonetic values in Romaji, or press the Romaji/Kana toggle key and type words in Hiragana. Japanese engineers often prefer typing in Romaji, but most clerks prefer Hiragana since it requires fewer keystrokes. With either method, the system echoes Hiragana letters in reverse video.
4. If that Hiragana word is the one you want, press the Commit key.
5. To convert the word to Kanji, press the Conversion key. The system shows a Kanji spelling in the Pre-edit Region. If that's the spelling you want, press the Commit key. If not, press the Conversion key again to see the next choice.

5

Font Handling

In OpenWindows 3.3, all 56 OpenFonts® in all sizes include ISO Latin-1 characters. Most other OpenWindows fonts do not.

Solaris supports EUC encoding, as did releases of Asian OpenWindows. Refer to Chapter 3 for more information on EUC. To support languages with multiple character fonts, XView applications can open a *font set*. Here's an example of a call to open the Minchou font with Internationalized XView. This is intended only as an example—the actual interface for Japanese and Asian OpenWindows might be different:

```
Xv_Font_Set font_set;
font_set = (Xv_Font_Set) xv_find(frame, FONT_SET,
        FONT_FAMILY, FONT_FAMILY_MINCHOU,
        FONT_STYLE, FONT_STYLE_NORMAL,
        FONT_LOCALE, xv_get(server, XV_LC_DISPLAY_LANG),
NULL);
```

Additionally, there are two new text functions to handle wide characters: `XwcDrawString` is the wide character equivalent of `XDrawString`, and `XwcTextWidth` is the wide character equivalent of `XTextWidth`.

Note – Earlier releases of OpenWindows displayed ISO Latin-1 characters by default, even if the `LC_CTYPE` environment variable was not set. Because this release of OpenWindows supports multibyte characters and EUC, Latin-1 characters are not displayed unless the `LC_CTYPE` environment variable is set to `iso_8859_1`.