22
Internationalization

In this chapter, I discuss the features in XView that allow applications to be internationalized. For additional information on internationalization for XView Version 3.2, refer to the *XView 3.2 Developer's Notes*, Part II "Internationalization," distributed by Sun Microsystems, Inc. It describes the internationalization modifications by package as well as XView's wide character support. Internationalized applications do not have to be redesigned or recompiled to run in a different language or *locale*. The goal is to make a single object version of an application support any number of languages. The attribute `XV_USE_LOCALE` enables the internationalization features. All internationalized applications must set this attribute to `TRUE`.

The basic idea in internationalization is to separate language-specific data from the rest of the application code. Adapting the application to any supported language, or *localization*, then requires modification of only the language-specific data. The task of internationalization includes translation of application-specific strings and modification of layout files. All other aspects of the user interface are handled by XView.

The internationalization features discussed in this chapter are:

Locale Setting
: Before running an internationalized application, a user must choose what language to run in. This is done by the locale setting mechanism of XView.

Localized Text Handling
: Developers need to be able to write application strings (error messages, menu labels, etc.) in their native language and have those strings retrieved in the language specified by the locale. This process is described in Section 22.2, "Localized Text Handling."

Object Layout and Customization
: When an application is run in a non-native language, the dimensions of its strings may change (among other things). This may cause the dimensions of objects with text strings in them, such as buttons and panels, to change. Object Layout is the mechanism by which the screen location of objects is modified to accommodate these kinds of changes. Section 22.3, "Object Layout and Customization," describes how an application can be customized to accommodate various object layouts.

22.1 Locale Setting

This section describes the following internationalization topics:

- Locale definition.
- How to enable internationalization features in XView.
- How to switch the locale from the OpenWindows Localization Properties Sheet.
- How the locale is set in XView using locale attributes, command-line options, XView locale resources, and the ANSI-C/POSIX `setlocale()` function.
- Locale implementation limits and restrictions.

22.1.1 Locale Definition

In an internationalized program, the *locale* specifies the language and cultural conventions used in a program. Locale setting affects the display and manipulation of language-dependent features.

XView defines a program's locale and the categories within each locale. These categories refer to the language-dependent features defined by ANSI-C and OPEN LOOK. These features include the display characteristics of monetary, numeric, and time fields.

XView offers several methods of setting locale. In order of priority, locale is set as follows:

1. XView locale attributes.
2. Locale command-line options.
3. XView locale resources (using the OpenWindows Workspace Property Sheet or an entry in the X resource database).

22.1.2 Enabling Internationalization

The attribute `XV_USE_LOCALE` enables the internationalization features in XView. For internationalized applications to work properly, this attribute must be set to `TRUE`. This attribute is only valid for `xv_init()`. The following is an example:

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv,
          XV_USE_LOCALE, TRUE,
       NULL);
```

22.1.2.1 Setting path of locale-specific files

The attribute XV_LOCALE_DIR specifies the location of an application's locale-specific files. The directory structure referenced by XV_LOCALE_DIR is:

```
<XV_LOCALE_DIR>/locale/LC_MESSAGES
<XV_LOCALE_DIR>/locale/app-defaults
<XV_LOCALE_DIR>/locale/app-defaults/<application name>
```

where *locale* is expanded differently depending on which internationalized feature is using XV_LOCALE_DIR. For a discussion of the specific use of XV_LOCALE_DIR, refer to Section 22.2, "Localized Text Handling," and to Section 22.3, "Object Layout and Customization."

22.1.3 OpenWindows Localization Properties Sheet

The OpenWindows Localization Properties Sheet, shown in Figure 22-1, provides the OPEN LOOK user interface for setting locale. To set locale, bring up the OpenWindows Workspace Property sheet and choose "Localization" from the menu. The system then allows you to specify the following categories within the Localization Menu:

Basic Setting
: Specifies the country of the user interface. Once the Basic Setting is specified, the user can set the specific settings. The specific settings define properties for each locale.

Display Language
: Specifies the language in which labels, messages, menu items, help text, and so on, are shown.

Input Language
: Specifies the language used for input.

Time Format
: Specifies the format for time and date displays.

Numeric Format
: Defines the numeric format.

Supplementary Settings
: Provides a selection of additional settings for any of the Specific Settings. That is, if additional selections are required to specify Display Language, Input Language, Time or Numeric Format, pressing the Supplementary Settings button will display them.

  The Localization Property Sheet sets locale via XView locale resources in the X resource manager database. This process is described later in this chapter. Once a setting is made in the Localization Properties Sheet, subsequent applications will adopt the new environmental settings.

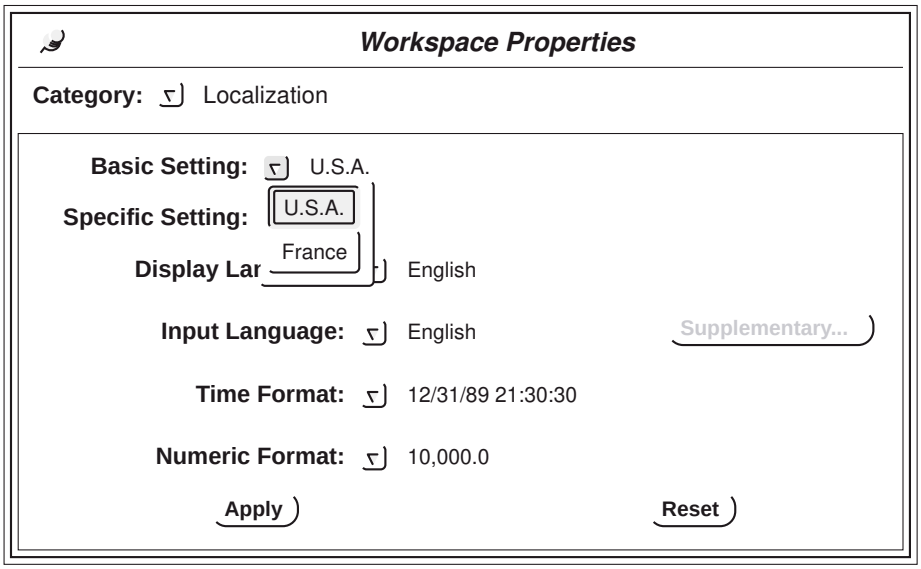

Figure 22-1. OPEN LOOK localization menu

22.1.4 XView Locale Attributes

Several attributes let you specify the locale from within an application. Each of these locale attributes corresponds to the locale parameters described for the OpenWindows property sheet. The attributes are shown in Table 22-1.

Table 22-1. XView Locale Attributes

Locale Parameter	XView Attribute
Basic Setting	`XV_LC_BASIC_LOCALE`
Display Language	`XV_LC_DISPLAY_LANG`
Input Language	`XV_LC_INPUT_LANG`
Numeric Format	`XV_LC_NUMERIC`
Time Format	`XV_LC_TIME_FORMAT`

These attributes specify the different OPEN LOOK locale categories. They can only be set at creation time through `xv_init()`, but can be queried at any time with `xv_get()` on any object that is a subclass of `WINDOW` or `SERVER`. The value that is set using the above attributes must be a valid locale name in the system.*

Locale attributes are used in situations when an application developer needs to force a program to operate in a specific locale. For example, if a program only works in the French locale, then the locale attributes can be used to hardwire the locale to French, preventing

*The valid locale names are dependent upon the operating system used.

anyone from switching the locale using X resources or command-line options. Each of these attributes takes a `char *` argument. Once again, these attributes are only valid for `xv_init()` and `xv_get()`. For example, set the basic locale to French with the following code:

```
xv_init(   XV_USE_LOCALE,  TRUE,
     XV_LC_BASIC_LOCALE,  "fr",
     XV_INIT_ARGC_PTR_ARGV,     &argc, argv,
     NULL);
```

22.1.5 Command-line Options for Specifying Locale

The locale command-line options allow the locale to be set when an application is started. These options may be used to override the system environment for a specific program. The command-line options are shown in Table 22-2, and in Section 6, *Command-line Arguments and XView Resources*, in the *XView Reference Manual*.

Table 22-2. Locale Command-line Options

Locale Parameter	Command-line Option
Basic Setting	`-lc_basiclocale`
Display Language	`-lc_displaylang`
Input Language	`-lc_inputlang`
Numeric Format	`-lc_numeric`
Time Format	`-lc_timeformat`

An example of these command-line options follows:

```
% textedit -lc_basiclocale fr -lc_displaylang C -lc_inputlang fr
```

This command line brings up a text editor with messages and panel labels in English. The user's input would be in French.

22.1.5.1 XView locale resources

Locale can also be set through resources written in the X11 Resource Manager database (*~/.Xdefaults*). The workspace property sheet uses the XView Locale Resources to set locale parameters. Resources are shown in Table 22-3.

Table 22-3. XView Locale Resources

Locale Parameter	Resource Name
Basic Setting	`basicLocale`
Display Language	`displayLang`
Input Language	`inputLang`
Numeric Format	`numeric`
Time Format	`timeFormat`

The example below shows locale resource entries in the X resource database. This example shows the resources set to the value of "C." This is the default locale on most operating systems. As far as character input/output is concerned, "C" is synonymous with 7-bit ASCII.

```
*basicLocale:     C
*displayLang:     C
*inputLang:       C
*numeric:         C
*timeFormat:      C
```

This example shows the resource set to the value "C". This value specifies the English language (American).

22.1.5.2 ANSI-C/POSIX

Internally, XView uses several standard ANSI-C/POSIX functions. The `setlocale()` is an internal function that should **not** be used in application programs. If locale attributes, command-line options, or resources are not specified, XView uses environment variables to set locale information. This method follows the ANSI-C/POSIX specification for locale announcement. The environmental variables are the same as the POSIX locale categories. POSIX categories are shown in Table 22-4.

Table 22-4. POSIX Categories

OPEN LOOK Category	POSIX Category
Basic Setting	`LC_CTYPE/LANG*`
Display Language	`LC_MESSAGES`
Input Language	`n/a`
Numeric Format	`LC_NUMERIC`
Time Format	`LC_TIME`
n/a	`LC_MONETARY`
n/a	`LC_COLLATE`

*`LANG` (Basic Setting above) is not one of the POSIX categories, but is a convenience environment variable used to set the default for other locales. Basic Locale sets `LC_CTYPE`, `LC_MONETARY`, and `LC_COLLATE`. Input Language will be the same as `LC_CTYPE`.

22.1.6 Limits and Restrictions

The Basic Locale setting determines the character set used by XView. The other locale categories can differ from the basic setting, but they cannot require a different character set from the Basic Locale. For example, setting the Basic Locale to “fr” (French) and the Display Language to “ja” (Japanese) will not work because the French and Japanese locales require different character sets.

The following restrictions thus apply:

1. If basic locale setting is the “C” locale, then all other locale categories must be in the “C” locale.
2. If the Basic Locale is set to a locale other than the “C” locale, then all other locale categories must be set either to a locale that uses the same character set as the basic setting, or to the “C” locale.

22.2 Localized Text Handling

When creating an internationalized application, developers need to write text strings in their native language (error messages, menu labels, button labels) and have those strings appear in foreign languages based on the locale established in the environment. Note that the term “foreign language” refers to non-native languages. XView allows you to define any language as the native language and any other language as the foreign language.

Localized text handling allows a developer to write text strings in the native language and have those strings retrieved in a foreign language. To implement localized text handling in an XView program, the developer must perform the following tasks:

1. For all displayable text strings, use the string as an argument in the `gettext()` or `dgettext()` functions. These functions accept native language strings as arguments, and return the equivalent foreign language strings.
2. Extract the native language text strings used with `gettext()` and `dgettext()` from the source and store them in a portable message file. This extraction can be done by hand, or through the source filter `xgettext`.
3. Provide the foreign language equivalent for each text string, and put it in the portable message file. The portable message file can be edited using any plain text editor which supports both the native and foreign language.
4. Run the `msgfmt` program on the portable message file to create a *text domain*. Here is an explanation on domains. The complete list of messages that exists in a system can be divided in two different ways: Those that belong to a particular language can be grouped together in a *locale*.

   Those that logically belong to a “function” (for lack of a better word), “package,” “library,” etc., can be grouped together in a “text domain.” Messages are grouped in domains for convenience and maintenance. A domain contains messages from its

"group" in all the valid languages of the system. Likewise, a locale contains all the messages in a specific language (this covers all domains of the system). Example of domains are, "panel_labels," "library_error_messages," "menu_labels," and so on.

So, to obtain the correct translation of a message string, one needs to know what language it is to be displayed in (locale - French, Japanese, etc), and which logical group it belongs to (domain - error messages, button labels, etc.) The text domain is in binary file format and is not interchangeable between CPU architectures. This file cannot be edited using an editor; it must be created using `msgfmt`.

XView follows the UniForum Messaging Proposal (1003.1b) specification of *gettext, dgettext, text domain,* and *portable source file format for messages* (PSFFM).

22.2.1 Localized Text Handling – Application Programmer Interface

Several functions support localized text handling. They are described in this section.

22.2.1.1 gettext()

The procedure gettext( ) retrieves the foreign language string for the passed string. The format of gettext( ) is:

```
char *
gettext(msg_id)
    char   *msg_id;
```

where "msg_id" is the native language string for which the foreign language equivalent is required. The locale used for retrieval is the current setting of the Display Language category (`LC_MESSAGES`). The domain used is the "current domain," or the domain set by the last call to "textdomain( )." If a call to textdomain( ) has not been made, the default domain is used.

If gettext( ) cannot find the foreign language equivalent for "msg_id," then "msg_id" itself is used.

The example below searches the current text domain for the foreign language string that is identified by the key "wrongbutton." The foreign language string is then assigned to "message:"

```
message = gettext("wrongbutton");
```

22.2.1.2 dgettext()

Another function, `dgettext()`, allows you to specify the text domain from which to retrieve the foreign language string without changing the current text domain. Its format is:

```
char *
dgettext (domain_name, string)
   char *domain_name;
   char *string;
```

Where *domain_name* is the text domain containing the desired foreign language equivalent, and *string* is the native language string for which the foreign language equivalent is desired.

The following example searches for the foreign language equivalent `wrongbutton` in the text domain `library_error_strings`, then assigns the result to `message`. The current text domain, which was active before this call, is not changed.

```
message = dgettext("library_error_strings",
      "wrongbutton");
```

22.2.1.3 textdomain()

Use `textdomain()` to specify the current text domain. It has the following format:

```
char *
textdomain(domain_name)
   char *domain_name;
```

Where *domain_name* is the desired current text domain.

The following example sets the current text domain to `library_error_strings`. All subsequent `gettext()` calls automatically search `library_error_strings` for the foreign language equivalent.

```
textdomain("library_error_strings");
```

The value of the current text domain can be queried as follows:

```
current_domain = textdomain(NULL);
```

The domain may be set to the implementation-defined default domain by calling `textdomain()` with a domain name set to an empty string. Such a call to `textdomain()` will also return the implementation-defined default domain. (Refer to the man page on `textdomain` for specific information on the default domain for your operating system.) For example:

```
default_domain = textdomain("");
```

Multiple text domains can be used within the same application. This allows different types of strings to be grouped into separate text domains. For example, one text domain might contain error messages from the XView library, a second text domain might contain XView menu and button text, and a third might contain strings from the application.

22.2.1.4 bindtextdomain()

The current Uniforum messaging proposal does not specify a method for identifying the location of text domains in the file system. The function `bindtextdomain()` performs this function. The function defines the location of the text domain files to the library. The format of the `bindtextdomain()` call is:

```
char *
bindtextdomain(domain_name, binding)
   char *domain_name;
   char *binding;
```

The following example sets the binding for the text domain `library_error_strings` to */home/myapp/lib/locale*:

```
bindtextdomain("library_error_strings", "/home/myapp/lib/locale");
```

All subsequent gettext( ) and dgettext( ) calls for the text domain library_error_strings will look in:

```
/home/myapp/lib/locale/<Display Language Setting>/LC_MESSAGES
```

for the message file library_error_strings.mo.

The `bindtextdomain()` function also has a default domain binding. The default domain binding is applied to any text domains that haven't been bound explicitly by a `bindtextdomain()` call. Unlike the other bindings, the default binding is actually a list of directories that are searched in order. There is always at least one directory in the list: */usr/lib/locale*.

Additional directories can be added to the list as follows:

```
bindtextdomain("", "/home/myapp/lib/locale");
```

Directories are searched in the order in which they are added to the list. In the above example, */usr/lib/locale* would be searched first followed by */home/myapp/lib/locale*.

An opaque snapshot of the state of the default binding list can be taken with the following:

```
(char *) state = bindtextdomain("", NULL);
```

Note that this is similar in principle to using `setlocale()`:

```
setlocale(LC_ALL, NULL);
```

The state can be restored at a later time using the following:

```
bindtextdomain("", state);
```

22.2.1.5 Examples

The following three examples retrieve the same strings, but have different effects on the text domain. The first example does not change the current text domain. The second and third examples change the current text domain to `library_error_strings`, then retrieve the foreign language string of `wrongbutton`.

```
message = dgettext("library_error_strings",
          "wrongbutton");

textdomain("library_error_strings");
message = gettext("wrongbutton");

textdomain("library_error_strings");
message = dgettext("", "wrongbutton"); \\* "" = current domain *\\
```

22.2.1.6 XV_LOCALE_DIR

If `XV_LOCALE_DIR` is set in `xv_init()`, XView calls `bindtextdomain()` on the application's behalf to prepend the pathname specified by `XV_LOCALE_DIR` to the default domain path list. All unbound text domains would be located in the directory with the general format:

```
<XV_LOCALE_DIR>/<Display Language Setting>/LC_MESSAGES/domain
```

"domain" is the name of the text domains. The default value of `XV_LOCALE_DIR` is *$OPENWINHOME/lib/locale.*
An example, using `XV_LOCALE_DIR` follows:

```
xv_init(XV_INIT_ARGS, argc, argv,
         XV_LOCALE_DIR, "/app_home/lib/locale",
         NULL);
```


22.2.2 Creating a Text Domain

Once the program is written using `gettext()` and `dgettext()`, text domains are created using the following procedures:

1. For example, you could run `xgettext` on the following source file fragment (see the man page for `xgettext` for proper usage):

```
message1 = dgettext("library_error_string", "Save");
message2 = dgettext("library_error_string", "File");
message3 = dgettext("library_error_string", "Edit");
```

This filter extracts the strings from the source files, and places them in a portable message file. The format of the portable message file produced is shown below:

```
domain "library_error_strings
msgid "File"
msgstr
msgid "Save"
msgstr
msgid "Edit"
msgstr
```

The `xgettext` filter currently does not expand `#define` commands. If you have `#define` strings which need to be extracted, then you need to run your application through the C preprocessor as follows:

```
cc -P file.c
```

This runs the source file through the C preprocessor only, and outputs to a file with a suffix, `.i`. You should then run this `.i` file through `xgettext` to extract all strings:

`xgettext` *xgettext options* `file.i`

This will produce a file called *library_error_string.po.*

2. Add the foreign language equivalents for all the native language strings in the portable message file between quotes next to `msgstr`. Use a plain text editor that supports the character set/encoding for the language.

```
domain "library_error_strings
msgid "File"
msgstr "<french for File>"
msgid "Save"
msgstr "<french for Save>"
msgid "Edit"
msgstr "<french for Edit>"
```

If you wish to create multiple text domains, you may separate the related text strings into separate files. Be sure that the first line of every file consists of a domain line at the beginning of each file. You may also have multiple text domains within the same file. To do this, simply start each new domain with the line:

`domain "`*domain_name*`"`

where *domain_name* is the name of the text domain. Note that the message identifiers and strings of a particular domain must be in the same file under the same domain name.

3. Run the `msgfmt` program on each portable message file to produce the text domain file:

```
msgfmt library_error_string.po
```

This produces the text domain file *library_error_string.mo*, which contains the messages compiled in an internal format. The text domain file is in binary format, and is not interchangeable between CPU architectures. It cannot be edited using an editor, and must be created using `msgfmt`.

4. Move each text domain to the directory under the path specified as follows:

`<XV_LOCALE_DIR>/`*locale*`/LC_MESSAGES`

As mentioned previously, `gettext()` and `dgettext()` search the text domain for the native language text or *key* value, then extract the corresponding foreign language text. The key is a unique name that identifies the string to be retrieved. The key does not have to be the native language string, but the native language string is a good choice since it tends to be self-documenting.

22.2.3 New and Enhanced XView Attributes for gettext()

`NOTICE_MESSAGE_STRING` specifies text to print in a notice. `NOTICE_MESSAGE_STRING` is necessary when using `gettext()` to get a notice message. The value of this attribute is a single `NULL`-terminated string, which may contain "\n" as a line break. `NOTICE_MESSAGE_STRINGS` specifies the text to print in a notice. The value is `NULL`-terminated list of strings, which may contain "\n" as a line break. `NOTICE_MESSAGE_STRINGS_ARRAY_PTR` specifies the text to print in a notice. The value is a variable pointing to a `NULL`-terminated array of strings, which may contain "\n" as a line break.

This example shows how notices are displayed in XView Version 2.

```
result = notice_prompt( panel , NULL,
     NOTICE_BUTTON,           "Update changes" ,    101,
     NOTICE_BUTTON,           "Ignore changes" ,    102,
     NOTICE_BUTTON,           "Destroy everything" ,103,
     NOTICE_MESSAGE_STRINGS,  "Press button,"
                             "then go home" ,
     NULL,
     NULL);
```

This example uses `NOTICE_MESSAGE_STRING` and makes the same notice window as the previous example.

```
result = notice_prompt( panel , NULL,
     NOTICE_BUTTON,           "Update changes", 101,
     NOTICE_BUTTON,           "Ignore changes", 102,
     NOTICE_BUTTON,           "Destroy everything", 103,
     NOTICE_MESSAGE_STRING,   "Press button,\nthen go home" ,
     NULL);
```

This example shows how to embed `gettext()`.

```
result = notice_prompt( panel , NULL,
    NOTICE_BUTTON,   gettext("Update changes" ),     101,
    NOTICE_BUTTON,   gettext("Ignore changes" ),     102,
    NOTICE_BUTTON,   gettext(" Destroy everything" ),     103,
    NOTICE_MESSAGE_STRING, gettext(" Press button,\n then go home" ),
    NULL);
```

22.3 Object Layout and Customization

The layout of screen objects (buttons, panel items, etc.) needs to change when the dimensions of the strings within those objects change. For example, a button string in French might be much longer than the same button string in English, thus requiring other buttons to be repositioned. Another example of object layout changes are objects that have to be repositioned to accommodate locale-specific representations such as date or time. A mechanism is required to alter the layout during localization.

22.3.1 Implicit and Explicit

Object layout of an XView application may be implicit or explicit. Using explicit layout, the application specifies *x* and *y* coordinates for the objects in the application. Implicit layout relies on certain defaults that are built into the toolkit. Thus, if a series of panel items is created, and no layout information is explicitly specified, the panel package will position the items based on default spacing rules. For example, one rule is to start a new row when a panel item extends past the edge of a panel.

Implicit positioning may be all that is required to localize an application. When the locale is switched, the new strings are picked up and the panel package automatically adjusts the layout of the panel items based on the dimensions of the new strings.

Other scenarios may exist, however, where explicit layout is required. One situation where implicit positioning does not provide a satisfactory layout is when switching locales. Another scenario requiring explicit repositioning is when accommodating changes in format when locales are switched. For example, the date representation in the United States is mm/dd/yy, while in India the date is represented as dd/mm/yy. Reliance on such explicit formats requires modification during localization.

The attributes, `XV_USE_DB` and `XV_INSTANCE_NAME`, allow developers to specify customizable attributes. These attributes can be used to specify explicit object layout.

22.3.2 Layout and Customization API

22.3.2.1 XV_LOCALE_DIR

Each application may have one or more locale-specific application defaults files. These files provide layout information and any other data stored under the X Resource Manager that is specific to a given locale.

The directory structure for the locale-specific defaults files used by the layout scheme is:

```
<XV_LOCALE_DIR>/locale/app-defaults/<appname>.db (read first)
<XV_LOCALE_DIR>/locale/app-defaults/<appname>/*.db
```

where *locale* is the current setting of Basic Locale.

Typically an application would have its locale-specific application defaults file (one file) in:

```
<XV_LOCALE_DIR>/locale/app-defaults/<appname>.db
```

If the application wants to group its application defaults into multiple files, they can be placed in the directory:

```
<XV_LOCALE_DIR>/locale/app-defaults/<appname>/*.db
```

22.3.2.2 XV_USE_DB

`XV_USE_DB` can be used to specify a set of attributes that are to be searched in the X11 Resource Manager database. `XV_USE_DB` takes a `NULL`-terminated list of attribute value pairs as its values. During program execution, each attribute in this `NULL`-terminated list is looked up in the X11 Resource Manager database. If the attribute is not found in the database, then the value specified in the attribute-value pair is used as the default value.

22.3.2.3 XV_INSTANCE_NAME

The attribute `XV_INSTANCE_NAME` is used to associate an instance name with an XView object. The instance name is used to construct the key value (resource name) used by the resource manager to perform the lookup. The key value is constructed by concatenating the instance names of all objects in the current object's lineage, starting with the name of the application or whatever was passed in with the `-name` command-line option. The XView attribute name remains in lowercase.

An example using `XV_INSTANCE_NAME` is shown below. Assume that the application name is "app." The key value is constructed by concatenating "app," "frame1," and "panel1." Thus, resource files containing entries such as the following would be specified as in Example 22-1:

```
app.frame1.panel1.xv_x: 20
app.frame1.panel1.xv_y: 20
```

Example 22-1. Using XV_INSTANCE_NAME

```
(void)xv_init(XV_USE_LOCALE, TRUE,
          XV_INIT_ARGC_PTR_ARGV, &argc, argv,
          ...
          NULL);

frame = (Frame)xv_create(0, FRAME,
          XV_INSTANCE_NAME, "frame1" ,
          NULL);


panel = (Panel)xv_create(frame, PANEL,
          XV_INSTANCE_NAME, "panel1" ,
          XV_USE_DB,
          XV_X, 10,
          XV_Y, 15,
          NULL,
          NULL);
```

If entries corresponding to the attributes XV_X and XV_Y were not specified in the resource file, the default values specified in the attribute-value pair will be used (XV_X will default to 10 and XV_Y will default to 15).

For more detailed information on the X resource manager, refer to Volume One, *Xlib Programming Manual.*

The list of attributes that are customizable is shown below:

```
CANVAS_HEIGHT                    CANVAS_MIN_PAINT_HEIGHT
CANVAS_MIN_PAINT_WIDTH           CANVAS_WIDTH

PANEL_CHOICE_NCOLS               PANEL_CHOICE_NROWS
PANEL_DROP_HEIGHT                PANEL_DROP_WIDTH
PANEL_EXTRA_PAINT_HEIGHT         PANEL_EXTRA_PAINT_WIDTH
PANEL_GAUGE_WIDTH                PANEL_ITEM_X
PANEL_ITEM_X_GAP                 PANEL_ITEM_Y
PANEL_ITEM_Y_GAP                 PANEL_JUMP_DELTA
PANEL_LABEL_WIDTH                PANEL_LABEL_X
PANEL_LABEL_Y                    PANEL_LIST_DISPLAY_ROWS
PANEL_LIST_ROW_HEIGHT            PANEL_LIST_WIDTH
PANEL_MAX_VALUE                  PANEL_MIN_VALUE
PANEL_NEXT_COL                   PANEL_NEXT_ROW
PANEL_SLIDER_WIDTH               PANEL_TICKS
PANEL_VALUE_DISPLAY_LENGTH       PANEL_VALUE_DISPLAY_WIDTH
PANEL_VALUE_STORED_LENGTH        PANEL_VALUE_X
PANEL_VALUE_Y

WIN_COLUMNS                      WIN_DESIRED_HEIGHT
WIN_DESIRED_WIDTH                WIN_ROWS

XV_HEIGHT                        XV_WIDTH
XV_X                             XV_Y
```

22.3.3 Command-line Options

The internationalization command-line options are supported for use with the customizable attributes. The list of command-line options is available in Section 6, *Command-line Arguments and XView Resources*, in the *XView Reference Manual*. The `-name` command-line option can be used to set the application instance name. For example, if you run:

```
mytool -name big
```

all the resource names of customizable attributes within `mytool` will use:

```
big.<rest of name>
```

for resource database lookup.

22.4 Internationalization Attribute Summary

The attributes that support internationalization are shown in Table 22-5. This information is described fully in the *XView Reference Manual*.

Table 22-5. Internationalization Attributes

XV_INSTANCE_NAME
XV_LC_BASIC_LOCALE
XV_LC_DISPLAY_LANG
XV_LC_INPUT_LANG
XV_LC_NUMERIC
XV_LC_TIME_FORMAT
XV_LOCALE_DIR
XV_LOCALE_DIR
XV_USE_DB
XV_USE_LOCALE