Internationalization

An internationalized application is one that may be run in many different languages without having to be rewritten or recompiled. This chapter describes how to design applications to use Motif's internationalization capability. It is not a general discussion of internationalization.

Issues in Internationalized Applications

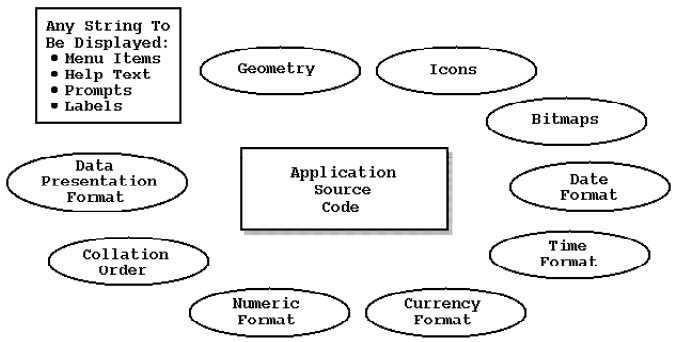
There are several important issues to keep in mind when designing an application so that it takes advantage of Motif's internationalization capabilities.

Internationalization and Localization

An internationalized application contains no code that is dependent on the user's language, the characters needed to represent that language, or any formats (such as date and currency) that the user expects to see and interact with. Motif accomplishes this by storing language and custom dependent information outside the application.

The following figure shows the kinds of information that should be external to an application to simplify internationalization.

Figure 1 Information External to the Application



Because the language and culture dependent information is separate from the application source code, the application does not need to be rewritten or recompiled to be marketed in a different countries. Instead, the only requirement is for the external information to be *localized* to accommodate local language and custom.

Localizing the application includes the process of translating certain parts of the external information into the appropriate language and storing the translated information in files that are then accessed by the application. In addition, the application may be told the format to use to display time, date, and the other language or culture dependent formats shown in the previous figure.

Every language consists of a set of characters that, either individually or in combination, represents meaningful words or concepts in the language. The set of characters is called a *character set*. The set of binary values needed to represent all the characters in a language is called a coded character set or, more simply, a *code set*.

Several attempts were started long ago to standardize code sets and continue to this day. The most commonly used code set for English is the American National Standard Code for Information Interchange (ASCII). It originally used a 7–bit encoding scheme plus an eighth bit for error control. Using 7 bits for character representation allows 128 unique binary values. Later versions use the eighth bit as a code bit allowing 255 characters. Both are fine for English and some other alphabetic languages, but neither is suitable for ideographic languages such as Chinese, Japanese, and Korean. Ideographic languages represent a concept or an idea as a single character; consequently, there are thousands of characters in these languages, and two or more bytes are needed to represent the characters.

Other standard code sets have been developed to accommodate other languages. The ISO8859 standard is perhaps the most commonly used of these. Different versions of the ISO8859 standard exist for various areas of the world. The following table shows a typical language and code set relationship for various areas. The code sets shown generally cover many more areas than are indicated, and the Table 11–1 is merely meant as a guide. (As an example, the ISO8859–3 code set covers, in addition to the languages indicated in the table, Afrikaans, Esperanto, German, Italian, Maltese, and Turkish. You can also use it for English.)

Table 1 Areas and Typical Code Sets

Area or Language	*Code Set*
English	ASCII, ISO8859–1
Western Europe	ISO8859–1
Eastern Europe	ISO8859–2
Dutch, Catalan, Spanish	ISO8859–3
Northern Europe	ISO8859–4
Russian, Ukrainian, Serbian	ISO8859–5
Hebrew	ISO8859–6
Greek	ISO8859–7, 8, 9
Japan	Shift JIS
Japan	UJIS

See the specifications for the American National Standards Institute (ANSI) C programming language and the *X/Open Portability Guide, Issue 3* (XPG3) for more information on standards involved in internationalization.

Obtaining Input

Special considerations must be made for the user of an application to input characters in the local written language. Virtually all applications require some action on the part of the user, often asking for input in one form or another. For example, an application can ask the user to input information in text form, such as name, home address, and so on. The user must then enter this information by typing it on the keyboard in the normal manner. This is done with relative ease in an English–based application but can become more complex when text in another language is desired.

Motif uses Xlib functions to provide the basic support for obtaining input in the Text widget.

The Problems

Many languages are expressed by means of an alphabet made up of characters or letters. The letters are arranged in groups to form meaningful words. A keyboard suitable for the language contains all the letters of the alphabet, plus the standard numerals and punctuation marks. The first problem arises when, as in English, standard spelling and usage requires *two* characters for each letter of the alphabet, while the standard keyboard contains only *one* key for each letter. The solution to this problem is a `Shift` key, which, when pushed in combination with another key, changes the character that key produces.

A somewhat more serious problem arises when the keyboard does *not* have all the alphabet characters. This can happen when a German user is using an English–based keyboard and needs a German character such as "[beta]."

A far more involved example is the case of defining a keyboard to use for the ideographic languages. Because thousands of characters are needed to represent an ideographic language, no reasonable keyboard can be constructed with a single key for each character.

The Solution

X and Motif solve these input problems by using an *input method*, which, in its simplest form, is a layer of mapping between the keyboard keys (or combinations of keys) that the user types and the text data that is passed to the application. For example, the Danish user with an English keyboard who needs the letter "" must enter a *combination* of keystrokes (this varies among vendors but could be `Extend char O /` as an example) rather than just one keystroke. This is very similar to the act of using the `Shift` key to access uppercase letters.

An ideographic language's input method is often based on the language's phonetics, but there are also input methods based on a common graphics property of certain characters. The graphics method involves defining a key to map to a common graphic symbol that is the basis for multiple characters. The phonetic method is more commonly used. It requires a phonetic (alphabet–based) writing system. The number of phonetic signs or characters is few enough that a unique key is assigned to each phoneme. Characters are entered by pressing the appropriate phonetic keys.

Note that the full definition of an input method actually includes the manner in which text is typed as well as the simple keyboard mapping. In one form of input method, text is simply typed at the spot where it is to appear. In another method, often used in languages where every character requires more than one keystroke, preliminary text appears in some secondary window on the screen until enough has been typed to uniquely specify a new character, which is then passed to the application. In several popular input methods, the user types a phonetic representation of a spoken word and the input method determines which characters are pronounced that way. If only one character meets this criterion, it is displayed. If more than one character meets the criterion, a list of all characters found is displayed and the user chooses the desired one. It is then passed to the application. See Section 11.4.1 for more information on input methods.

Displaying Output

Displaying the output produced by an application intended for international use also requires some consideration. In order to display text, it must have the appropriate content, encoding and fonts. For example, many languages, especially ideographic ones, require more than one font. Bitmaps and pixmaps must be localized as well. An icon that is an appropriate or meaningful symbol in one country may be totally inappropriate or meaningless in another.

Locales and Localization

A locale is the language environment determined by the application at run time. The *X Portability Guide* defines locale as a means of specifying three characteristics of a language environment that may be needed for localization: language, territory, and code set. Motif supports only one locale per application; that is, an application can set the locale only once, at start–up time.

Motif uses the locale to help find:

- Resource files
- UID files
- Bitmap files
- Fonts used to display text and labels
- Text input method
- Character size

The ANSI C method of setting the locale in an application is to use the function *setlocale*. How *setlocale* obtains a language when the language is not explicitly referenced in the call to *setlocale* is system dependent. For example, on POSIX systems, the environment variable **LANG** is used. The locale name is also used to establish a path to the localized files of information. How this is actually accomplished is explained in Section 11.2.

Localizing Applications

An internationalized application can be tailored to operate in many areas of the world, each with its own requirements for the language and customs to be used. This section explains some methods for localizing an application.

The following section describes how the user and the application developer (and perhaps the system administrator) establish the language environment of an application. It then discusses two general approaches to localizing applications. Succeeding sections focus on four aspects of localizing information in Motif programs:

- Resource files
- UID files
- Message catalogs
- X bitmap files

Many aspects of localization depend on the particular operating system, Motif implementation, and user environment in which the application runs. The following must all cooperate for correct localization to occur:

- The operating system's locale mechanism, if any
- The Motif implementation
- The application itself
- The user's system administrator
- The user's language environment

Techniques for Localization

Although there are different methods for localizing an application, there are some common considerations:

- The application should not explicitly code any language−dependent information in the application. This includes strings, fonts, and language −dependent pixmaps.
- The application should isolate text, fonts, and pixmaps, and translate them into the languages needed. Usually this information is stored in separate directories by language.

Establishing the Language Environment

The term *language environment* refers to the set of localized data that the application needs in order to run correctly in the user−specified locale. A language environment supplies the rules associated with a specific language. In addition, the language environment consists of any externally stored data, such as localized strings or text used by the application. For example, the menu items displayed by an application might be stored in separate files for each language supported by the application. This type of data can be stored in resource files, UID files, or, on XPG3−compliant systems, message catalogs.

A single language environment is established when an application executes. The actual language environment in which an application operates is specified by the application user, often either by setting an environment variable (**LANG** on POSIX−based systems) or by setting the *xnlLanguage* resource. The application then sets the language environment based on the user's specification. The application can do this either by using *setlocale* in a language procedure established by **XtSetLanguageProc**, or by using a method that does not call *setlocale*. In either case, Xt caches a per−display language string that is used by **XtResolvePathname** to find resource, bitmap, and UIL files.

An application that supplies a language procedure may either provide its own or use an Xt default procedure. In either case, the application establishes the language procedure by calling **XtSetLanguageProc** before calling **XtAppInitialize**. When a language procedure is installed, Xt calls it in the process of constructing the initial resource database. Xt uses the value returned by the language procedure as its per−display language string.

The default language procedure performs the following tasks:

- Sets the locale. On ANSI C−based systems, this is done by using the following code:

```
setlocale(LC_ALL, language );
```

  where *language* is the value of *xnlLanguage* or the empty string ("") if *xnlLanguage* is not set. When *xnlLanguage* is not set, the locale is generally derived from an environment variable (**LANG** on POSIX−based systems).
- Calls **XSupportsLocale** to verify that the locale just set is supported. If not, a warning message is issued and the locale is set to "C."
- Calls **XSetLocaleModifiers** specifying the empty string.

- Returns the value of the current locale. On ANSI C–based systems, this is the result of calling the following:
```
setlocale(LC_ALL, NULL);
```

The application can use the default language procedure by making the call to **XtSetLanguageProc** in this manner:

```
XtSetLanguageProc(NULL, NULL, NULL);
.
.
toplevel = XtAppinitialize(...);
```

By default, Xt does not install any language procedure. If the application does not call **XtSetLanguageProc**, Xt uses as its per–display language string the value of the *xnlLanguage* resource if it is set. If *xnlLanguage* is not set, Xt derives the language string from the environment. On POSIX–based systems, this is the value of the **LANG** environment variable.

It is important to note that the per–display language string that results from this process is implementation dependent and that Xt provides no public means of examining the language string once it is established. The following vary by operating system and by Motif implementation:

- The mechanism, if any, used to set the locale
- On ANSI C–based systems, the value returned by *setlocale*
- The possible values of any environment variables used to establish the language environment
- Whether or not *xnlLanguage* is used and, if so, its possible values

Furthermore, by supplying its own language procedure, an application may use any procedure it wants for setting the language string.

Using Locales

The locale provides local information to an application based on the user's language, territory, and code set. Both language and territory are needed because some languages are spoken in more than one country and more than one language may be spoken in some countries. (French is an example of the first, and Belgium, Canada, and Switzerland are examples of the second.)

Information in resource, UID, and image files can be localized and stored in separate directories by language. The Xt function **XtResolvePathname** uses the run–time locale to determine the proper directory to use.

On XPG3–compliant systems, an application can use message catalogs to localize text and messages. A message catalog file exists for each language, and each is usually stored in a separate directory by language.

The locale method of localizing compound strings and font lists consists of the following steps:

1. Establish a language procedure before calling **XtAppInitialize**. The language procedure calls *setlocale*.
2. Localize the compound strings and render tables using resource files, message catalogs, or UID files. Normally, do not specify any charset tags other than **XmFONTLIST_DEFAULT_TAG**.
3. Use font sets in resource or UID file font lists.
4. Use **XmStringGenerate** to create compound strings in the program, but only use the rendition tag **_MOTIF_DEFAULT_LOCALE**.

The run–time locale determines which fonts are used to display text. This is accomplished in the following manner:

- Motif calls **XtResolvePathname** to load resource or UID files that specify the names of fonts for font sets. **XtResolvePathname** uses a file search path that may vary depending on the display's language string.
- **XCreateFontSet** uses the locale to determine the fonts to be used from the base font name and the locale charset.

In this method, the application usually does not specify charset tags other than **XmFONTLIST_DEFAULT_TAG**. It is possible to supply explicit rendition tags with locale–dependent text. For example, text might be displayed using large and small fonts or bold and italic fonts. The application can do this with special tags in both the compound string and the render table associated with it. In the render table, match the tag with a font set specification that supplies the desired attribute (point size, for example). When the application creates the font set, the charset comes from the locale. For example, a resource file might specify a render table in the following manner to obtain fonts with a different point size:

```
*fontList:  -*-*-*-R-Normal--*-120-100-100-*-*:,\
            -*-*-*-R-Normal--*-180-100-100-*-*:BIG,\
            -*-*-*-R-Normal--*-80-100-100-*-*:SMALL
```

See Chapter 9 for more information about fonts and controlling font selection.

Localization Without Locales

In this method, the locale is not set in the program, and a language procedure is not needed. Instead, the user specifies the language environment

by using either *xnlLanguage* or an environment variable such as **LANG**. Resource, UID, and image files are localized and stored in separate directories by language, as they are when the application uses locales. **XtResolvePathname** uses the display's language string in the same way to determine the proper locations of these files.

Message catalogs are not used in this method. Also, in this case Text and TextField cannot accommodate 16–bit data.

The nonlocale method of localizing compound strings and render tables consists of these steps:
1.Localize compound strings by using UIL files. Localized render tables and font lists can appear in resource files.
2.Specify explicit rendition tags *other* than **_MOTIF_DEFAULT_LOCALE** in both compound strings and render tables.
3.Use font names with explicit charset components in resource or UIL files. Do not use font sets.
4.To create compound strings in the program, use **XmStringGenerate** with the rendition tag set to something other than **_MOTIF_DEFAULT_LOCALE**.

Resources and Localization

The resources used in an application that are subject to internationalization ought to be stored in files external to the application. These resources include

- All labels, particularly those that identify controls. Such labels are defined as type **XmString**, meaning they are compound strings.
- Text strings; that is, strings of text that are not compound strings.
- Render tables.
- Font lists.

Initial Resource Database

The information in the external resource files is used when Xt builds the initial resource database. The **XtDisplayInitialize** function loads the resource database by merging in resources from the following sources, in order of precedence (that is, each component takes precedence over the following components):

- The application command line
- A per–host user environment resource file on the local host
- Screen–specific resources for the default screen of the display
- A resource property on the server or user preference resource file on the local host
- An application–specific user resource file on the local host
- An application–specific class resource file on the local host

Localization applies to two components of the initial resource database––the application–specific user and class resources. Localized resources that are controlled by the programmer are in the application class resource file, and localized resources that are controlled by the user are in the user resource file. Note that the user resources take precedence over the application class resources.

Resource File Locations

XtDisplayInitialize calls **XtResolvePathname** to load both the user and the class resources.

To load the user's application resource file, **XtDisplayInitialize** uses the value of the **XUSERFILESEARCHPATH** environment variable as the search path. If that variable is not set or if the search path fails to find the file, and if the environment variable **XAPPLRESDIR** is defined, **XtDisplayInitialize** next tries an implementation–dependent search path with a number of entries that include **XAPPLRESDIR** and the user's home directory. If **XAPPLRESDIR** is not set or if that search path fails, **XtDisplayInitialize** tries another implementation–dependent search path with a number of entries that include the user's home directory.

To load the application–specific class resource file, **XtDisplayInitialize** uses the value of the **XFILESEARCHPATH** environment variable as the search path. If that variable is not set or if the search path fails to find the file, **XtDisplayInitialize** tries an implementation–dependent search path.

The search paths for both resource files may contain any substitutions recognized by **XtResolvePathname**. That routine substitutes the display's language string for *%L*. In an implementation–dependent manner, it substitutes the language, territory, and code set components of the language string for *%l*, *%t*, and *%c*, respectively. This mechanism allows Xt to load different resource files for different languages, as specified by the display's language string.

The display's language string is determined by the application's language procedure, if present, or else by the value of the *xnlLanguage* resource

or by the environment. The language string associated with any particular language and the search paths used to find the resource files depend on the system vendor, the Motif vendor, the application, and the user's system administrator. Determining the actual directories in which localized resource files reside requires coordination among all these sources.

In general, an application developer prepares a set of localized application class resource files, one for each language the application supports. The developer may also need to supply a language procedure appropriate for one or more of the systems on which the application will run. The application vendor must arrange for the resource files to be installed in the correct directories, depending on the operating system and the Motif implementation on which the application will run.

An Example

Following is an example of an application class defaults file for a simple program that creates a MainWindow with a Text widget. Because the render table specification includes a single rendition with a default tag, this resource file would be appropriate for an application that uses locales.

```
*renderTable.fontName:                  -*-*-*-R-Normal--*-180-100-100-*-*
*renderTable.fontType:                   XmFONT_IS_FONTSET
*Text1.value:\
Hier ist etwas Text fur das Text Widget.\n\
Gemischter 8-und 16-bit Text.
*version_box.messageString:     Dies ist i18n Demo Version
*version_box.okLabelString:     Schliessen
*version_box.dialogTitle:       I18n Demo Version
*pgm_ver_btn.labelString:       I18n Demo Version
*events_btn.labelString:        Aktionen
*help_btn_menu.labelString:     Hilfe
*help_btn_cascade.labelString:  Hilfe
*help_box.messageString:        Leider ist keine Hilfe hier.
*help_box.okLabelString:        Schliessen
*help_box.dialogTitle:          i18n Demo Hilfe
*stop_btn.labelString:          Enden
```

UIL and Localization

The general models for localizing applications that use UIL are the same as those for applications that do not use UIL. An application developer creates separate UIL files, each containing string and resource values for a particular language. UIL files can also be used in conjuction with localized resource and pixmap files. As with localization of resource files, there are two basic approaches to localizing UIL files: one that uses locales and one that does not.

Preparing Localized UID Files

When using locales with UIL, an application developer should follow these rules:

- Do not use a *character_set* declaration for the module.
- When creating compound strings in a UIL file, use double quotes and no character set specification for the text.
- When creating render tables or font lists in a UIL file, use font sets, not fonts. Do not specify character sets for the font sets.
- Before compiling a UIL file via the **uil** command, set up any environment variables (such as **LANG**) or other mechanisms the system vendor recommends to establish the locale that is appropriate for the UIL file to be compiled. Invoke the **uil** command with the *-s* option. This enables the UIL compiler to set the locale and parse double–quoted strings without explicit character sets in the locale's encoding. It also ensures that localized compound strings and font list entries are created with font list element tags of **XmFONTLIST_DEFAULT_TAG**.
- Before using the *Uil* function to compile a UIL file, set the locale that is appropriate for the UIL file to be compiled. In the *Uil_command_type* structure that is the first argument to the *Uil* function, set the *use_setlocale_flag* member to 1. This has the same effect as invoking the **uil** command with the *-s* option.

When localizing UIL files without using locales, an application developer should follow these rules:

- When using single quotes for the text of compound strings, supply a *character_set* declaration for the module.
- When using double quotes for the text of compound strings, supply an explicit character set for each text component.
- When creating font lists in a UIL file, use fonts, not font sets. Specify an explicit character set for each font.
- When compiling a UIL file via the **uil** command, do not invoke the command with the *-s* option. The UIL compiler does not set the locale, and it parses each string by using rules derived from the explicitly specified character set for that string.
- When compiling a UIL file via the *Uil* function, set the *use_setlocale_flag* member of the *Uil_command_type* structure to 0. This has the same effect as invoking the **uil** command without the *-s* option.

The UIL compiler processes a single source file for each invocation of the **uil** command or the *Uil* function. However, UIL has an *include file* directive that is similar to the C preprocessor's *#include* directive. If the file argument for this directive is not an absolute pathname, the compiler searches for the file in a series of directories. These include the directory of the main UIL source file and any directories specified via the *-I* option to the **uil** command or the *include_dir* member of the *Uil_command_type* structure for the *Uil* function.

One strategy for maintaining localized UIL source files is to place only language−independent information in the main UIL source file and to put all language−dependent information in included files that are in separate directories for each language. Then a developer can compile the UIL files for different languages without editing any UIL files. When using locales, a developer first sets up the environment for the intended locale. Whether using locales or not, the developer then invokes the UIL compiler with the proper include directory for the intended language.

In general, a developer can mix localized UIL files with localized resource files. For example, the developer might specify compound strings in UIL files and render tables in resource files. Note one exception: it is not practical to use resource files to localize compound strings without using locales. This is because no resource file syntax exists for supplying an explicit charset/locale tag for a compound string.

For resource values that the user may override, the developer must use resource files or fallback resources, or must in some way ensure that the user's resource settings can override the developer's settings from the UIL file.

MRM and Localized UID Files

Once the developer has generated localized UID files, the vendor and the user's system administrator must arrange for these files to be installed in the appropriate directories for the system where the program is to run. As with resource files, these directories depend on configurations established by the operating system vendor, the Motif vendor, and the system administrator.

MrmOpenHierarchyPerDisplay takes as an argument a list of names of UID files. It calls **XtResolvePathname** to find each file the list. If a filename is an absolute pathname, that pathname is the search path for **XtResolvePathname**. Otherwise, **MrmOpenHierarchyPerDisplay** constructs a search path in the following way:

- If the environment variable **UIDPATH** is set, the value of that variable is the search path.
- If **UIDPATH** is not set, but **XAPPLRESDIR** is set, **MrmOpenHierarchyPerDisplay** uses a default search path with entries that include **$XAPPLRESDIR**, the user's home directory, and vendor−dependent system directories.
- If neither **UIDPATH** nor **XAPPLRESDIR** is set, **MrmOpenHierarchyPerDisplay** uses a default search path with entries that include the user's home directory and vendor−dependent system directories.

These paths may include the substitution field *%U*. In each call to **XtResolvePathname**, **MrmOpenHierarchyPerDisplay** substitutes the current filename from the list of UID files for *%U*. The paths may also include other substitution fields accepted by **XtResolvePathname**. In particular, **XtResolvePathname** substitutes the display's language string for *%L*, and it substitutes the components of the display's language string (in a vendor−dependent way) for *%l*, *%t*, and *%c*. If necessary **MrmOpenHierarchyPerDisplay** searches the path twice, first with *%S* mapped to *.uid* and then with *%S* mapped to NULL. The substitution field *%T* is always mapped to *uid*.

The usual mechanism for employing localized UID files is to use a search path that contains one of the substitutions derived from the display's language string. As with resource files, the vendor and system administrator must ensure that the directories where the localized UID files reside match the display's language string (or the appropriate component of the language string).

Message Catalogs and Localization

On an XPG3−compliant system, an application can use message catalogs to localize text. The format of message catalogs is implementation dependent, and the application must take steps to coordinate the locations of the message catalogs with the locations of resource, UID, and image files. Use of message catalogs requires the following steps:

1. Using an implementation−dependent method, prepare a separate message catalog containing text to be localized for each language.
2. Arrange to have the message catalogs installed in the appropriate directories on the systems on which the application will run.
3. Arrange for the user's environment to be set up correctly so that the application can read the message catalog appropriate to the language.
4. In the program, use the *catopen* function to open a message catalog and the *catclose* function to close it.
5. Use the *catgets* function to read text from an open message catalog.
6. If necessary, convert the text to the target format (such as a compound string) and, for resources, supply the text in the appropriate widget creation argument list or call to **XtSetValues**.

The *catopen* function takes as an argument the name of the message catalog file. If this is an absolute pathname, *catopen* opens that file. Otherwise, *catopen* uses the value of the **NLSPATH** environment variable as a search path. This path can contain a number of substitution fields. The filename passed to *catopen* is substituted for *%N*. The value of the **LANG** environment variable is substituted for *%L*, and its language, territory, and code set components are substituted for *%l*, *%t*, and *%c*, respectively.

Note that these values may not be the same as the display's language string or its components. An application and software vendor that use message catalogs must coordinate the locations of message catalogs with those of localized resource, UID, and image files, which usually depend on the display's language string. One possible strategy is to call *catopen* with an absolute pathname constructed by calling **XtResolvePathname** with the value of **NLSPATH** as the search path argument. **XtResolvePathname** substitutes the display's language string and its components for *%L*, *%l*, *%t*, and *%c* in **$NLSPATH**. In this way, the application can use a single mechanism, the display's language string, to distinguish file locations by language. The software vendor must still arrange for the user's system administrator to install the message catalogs in the correct locations and to ensure that **NLSPATH** is appropriately set in the user's environment.

Images, Pixmaps, and Localization

A pixmap is a screen image that is stored in memory so that it can be recalled and displayed when needed. Motif has a number of pixmap resources that allow the application to supply pixmaps for backgrounds, borders, shadows, label and button faces, drag icons, and other uses. As with text, some pixmaps may be specific to particular language environments; these pixmaps need to be localized.

Motif maintains caches of pixmaps and images. The function **XmGetPixmapByDepth** searches these caches for a requested pixmap. If the requested pixmap is not in the pixmap cache and a corresponding image is not in the image cache, **XmGetPixmapByDepth** searches for an X bitmap file whose name matches the requested image name. **XmGetPixmapByDepth** calls **XtResolvePathname** to search for the file. If the requested image name is an absolute pathname, that pathname is the search path for **XtResolvePathname**. Otherwise, **XmGetPixmapByDepth** constructs a search path in the following way:

1. If the environment variable **XBMLANGPATH** is set, the value of that variable is the search path.
2. If **XBMLANGPATH** is not set but **XAPPLRESDIR** is set, **XmGetPixmapByDepth** uses a default search path with entries that include **$XAPPLRESDIR**, the user's home directory, and vendor−dependent system directories.
3. If neither **XBMLANGPATH** nor **XAPPLRESDIR** is set, **XmGetPixmapByDepth** uses a default search path with entries that include the user's home directory and vendor−dependent system directories.

These paths may include the substitution field *%B*. In each call to **XtResolvePathname**, **XmGetPixmapByDepth** substitutes the requested image name for *%B*. The paths may also include other substitution fields accepted by **XtResolvePathname**. In particular, **XtResolvePathname** substitutes the display's language string for *%L*, and it substitutes the components of the display's language string (in a vendor−dependent way) for *%l*, *%t*, and *%c*. The substitution field *%T* is always mapped to *bitmaps*, and *%S* is always mapped to NULL.

As with resource and UID files, the usual mechanism for employing localized X bitmap files is to use a search path that contains one of the substitutions derived from the display's language string. As with resource and UID files, the vendor and system administrator must ensure that the directories where the localized X bitmap files reside match the display's language string (or the appropriate component of the language string).

See Chapter 12 for more information on images and pixmaps.

Comparing Approaches to Localization

The locale approach allows an application to use existing internationalization routines. On the other hand, the application is limited in portability to systems that support the same internationalization standards (XPG3, POSIX, or ANSI). This approach is also only applicable to applications using a single language.

The nonlocale approach only addresses the aspect of isolating information from the application and ensuring that it uses the proper localized version of this information. The disadvantage is that there is more work for the programmer and there may be nonstandard functionality. The advantages are that there is guaranteed portability across all platforms that support Motif, and that it allows handling of multiple character sets for specialized applications that require this functionality.

Layout Direction

Layout direction refers to the direction that is used to display visual elements such as widget children, widget components, and text. In general, this direction matches the direction that people use when reading or writing in a particular language. Languages such as English, French, German, and Swedish are read and written from left to right. Therefore, when users working in those languages enter characters from a computer keyboard, each new character is displayed to the right of the preceding one. These same users would also expect the layout of other visual elements to be displayed from left to right. For example, in a menu bar, the cascade buttons would be laid out from left to right so that a simple menu bar would position the "File" cascade button in the upper left corner, and the "Help" cascade button would appear in the upper right corner of the menu bar.

Languages such as Arabic and Hebrew are read and written from right to left. To display text correctly in these languages on the screen, each successive character that a user enters must appear to the left of the preceding character. Using the example above for layout of other visual elements, these users would expect a menu bar to lay out cascade buttons from right to left. The result would typically position the "File" cascade button in the upper right corner and the "Help" cascade button in the upper left corner of the menu bar.

There are several reasons why it is helpful for programmers to be able to specify the layout direction in applications:

- Application programmers want to use the same application in a variety of locales, including those with right−to−left oriented languages. They need to be able to specify that, when using a locale whose language is either Hebrew or Arabic, menus, labels, and messages, for example, should be displayed from right to left and be right justified.
- When applications require entering numeric values, even if the application is restricted to an audience with a right−to−left locale, users need to be able to enter numbers in certain text widgets so that they display from left to right while still entering text that displays from right to left in other text widgets.

You can use the **XmNlayoutDirection** resource to set the default layout direction for your entire application. This resource specifies the default layout direction for all widgets that are affected by it. In turn, the **XmNlayoutDirection** resource sets a default rendering direction for any compound string (**XmString**) that does not have a component specifying the direction for that string. A widget that needs to render such a string should use this resource value to substitute for the missing direction indicator.

The following two examples clarify the use of the **XmNlayoutDirection** resource.

Suppose your application contains only unidirectional compound strings; that is, every **XmString** in the application is either left−to−right or right −to−left. To set the layout direction, all you need to do is set the appropriate value for the **XmNlayoutDirection** resource. You do not need to create compound strings with specific direction components. When the application renders an **XmString**, it should look to see if the string was created with an explicit direction (**XmStringDirection**). If there is no direction component, the application should check the value of the **XmNlayoutDirection** resource for the current widget and use that value as the default rendering direction for the **XmString**.

Another more complex example involves an application that runs in a locale with right−to−left languages but includes text widgets for entering numbers, which need to be displayed from left to right. In this example, the application needs to set the **XmNlayoutDirection** resource to right −to−left for the entire application and then explicitly reset the **XmNlayoutDirection** resource to left−to−right only for those widgets that display numerical values. You still do not need to set any direction components for the compound strings themselves.

In Motif applications, you can set the layout direction by using the **XmNlayoutDirection** resource from the **VendorShell** or *MenuShell*. Manager and Primitive widgets (as well as Gadgets) also have an **XmNlayoutDirection** resource. The default value is inherited from the closest ancestor that has the same resource.

The layout direction resource affects some or all of the subwidgets of the following three widget classes:

- **XmGadget**
- **XmPrimitive**
- **XmManager**

The specific effects of the **XmNlayoutDirection** resource vary with the widget in question. The following three sections outline these effects for the widgets in the three listed classes.

XmLabelGadget and Related Widgets

The following list describes display situations with elements of the **XmLabelGadget** class that are dependent on the layout direction.

- **XmCascadeButtonGadget**
  - Positioning of cascade graphics

- Positioning of menu popup

- **XmLabelGadget**
  - Meaning of **XmNalignment** resource values
  - Default **XmNstringDirection**
  - Positioning of accelerator text
- **XmPushButtonGadget**, **XmTabButtonGadget**
  - Positioning of accelerator text
  - Meaning of **XmNalignment** resource values
- **XmToggleButtonGadget**
  - Positioning of accelerator text
  - Meaning of **XmNalignment** resource values
  - Positioning of toggle graphic

XmPrimitive and Related Widgets

The widgets **XmCascadeButton**, **XmLabel**, **XmPushButton**, **XmTabButton**, and **XmToggleButton** use the **XmNlayoutDirection** resource in the same manner as their corresponding gadget outlined in the previous section.

Note that the arrow keys `osfRight` and `osfLeft` refer to absolute directions within a row, and do not change their meaning when a widget's layout direction changes. Similarly, `osfUp` and `osfDown` refer to absolute directions within a column, and do not change their meaning when a widget's layout direction changes. However, a widget's layout direction does affect the interpretation of the arrow keys when the pointer is at the end of a row or column. In other words, a widget's layout direction will affect the way in which the widget "wraps" when it reaches the end of a row or column.

The following list describes display situations with elements of the **XmPrimitive** class that are dependent on the layout direction.

- **XmDrawnButton**
  - Resizing edge default
- **XmList**
  - **XmNstringDirection** default value
  - Meaning of alignment resources
  - MARQUEE behavior during selection
- **XmScrollBar**
  - Default value of **XmNprocessingDirection**
- **XmText**
  - Text writing direction only for **XmTOP_TO_BOTTOM** is supported.

XmManager and Related Widgets

The following list describes display situations with elements of the **XmManager** class that are dependent on the layout direction.

- **XmBulletinBoard**
  - The **XmNdialogTitle** direction is set from **XmNstringDirection**
- **XmComboBox**
  - Layout of arrows with regard to text
  - Direction in which List can be displayed
  - Resize direction for List
- **XmCommand**
  - Positioning of prompt string
- **XmContainer**
  - Default layout of contained objects
  - Positioning of label with regard to pixmap for icons
  - MARQUEE selection behavior
  - Position of header
- **XmDrawingArea**
  - Resizing edge default
- **XmFileSelectionBox**
  - Placement of scrollbars

- **XmForm**
  - Meaning of left and right in resource values
  - Default side for attachments
- **XmFrame**
  - Layout of children
  - Meaning of **XmNchild*Alignment** resources
- **XmMainWindow**
  - Layout of children
- *MenuBar*
  - Layout of children
- **XmMessageBox**
  - Layout of buttons
  - Positioning of pixmap
  - Default alignment of labels
- **XmNoteBook**
  - Default layout of children and book visuals
  - Meaning of "left" and "right" for arrows
- *OptionMenu*
  - Alignment of label
  - Positioning of bar graphic
  - Positioning of pulldown menu
- **XmPanedWindow**
  - Positioning of sash
  - Meaning of **XmNsashIndent**
- *PopupMenu*
  - Location of hotspot
  - Positioning of menu
- *PulldownMenu*
  - Alignment of edge of menu with regard to parent cascade button
- **XmRowColumn**
  - Layout of children, including menu bar cascades
  - Meaning of **XmNentryAlignment**
  - Meaning of **XmNentryVerticalAlignment**
  - Meaning of **XmNisAligned**
- **XmScale**
  - Positioning of text string
  - Positioning of value if shown
  - Default value of **XmNprocessingDirection**
  - Positioning of tick marks.
- **XmScrolledWindow**
  - Default positioning of scrollbar
- **XmSelectionBox**
  - Layout of buttons
  - Alignment of labels
  - Interpretation of **XmNchildPlacement** resource
- **XmSpinBox**
  - Default layout of text with regard to arrows
  - Meaning of left and right arrows

Internationalization and Text Input

An application subject to internationalization presents some unique problems when it deals with text input. The application must be able to correctly interpret and process text input in any language. This section explains how an application accomplishes this.

Input Method

Although there are many different keyboards in use, sometimes certain characters in an alphabetic language are not directly available on any keyboard. In this case, the user must type a combination of keys to input the desired character. In English, for example, the capital letters are produced by pressing the <Shift> key in combination with a letter key. Other alphabetic languages with larger alphabets than English may use slightly more complex combinations of keystrokes to describe their entire alphabet.

This problem, however, is compounded many times in the case of ideographic languages, which may require thousands of different characters for basic text. This far exceeds the capability of any keyboard and makes it impossible to have a keyboard with all of the language's symbols. An *input method* can be used to overcome this difficulty.

An input method is simply the mechanism that is used to map between the keys pressed by a user and the resulting characters that are input to an application. A common feature of many input methods is that the application user may press combinations of keys to create a single character. Creating characters from keystrokes is called *pre−editing*.

Input methods may require several areas to display the actual keystrokes.

- The *status area* is an output−only window that identifies the style of input (phonetic, numeric, stroke and radical, and so on) and the current status of an input method interaction.
- The *pre−edit area* displays the intermediate text for languages that are composed before the application acts on the data. There are several possible locations for the pre−edit area:

  Over−the−spot

  Displays the data in an input method window that is placed over the point of insertion.

  Off−the−spot

  Displays the pre−edit window inside the application window (usually at the bottom) but not at the point of insertion.

  On−the−Spot

  Displays the pre−edit string in the text widget window.

  Root−window

  Uses a pre−edit window that is a child of the root window.

  A VendorShell resource, **XmNpreeditType** determines which style is used for a Text or TextField input method. The syntax, possible values, and default value of this resource are implementation dependent.
- The *auxiliary area* is used for popup menus and customizing dialogs that some input methods use.

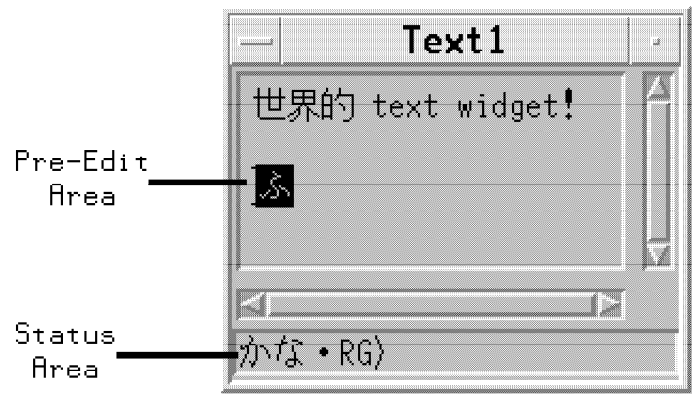
Input methods are supplied by vendors and are implementation dependent. The VendorShell resource **XmNinputMethod** is an implementation −dependent string that specifies the input method portion of the locale modifiers. If a value is supplied for this resource, Motif uses it to set the locale modifiers before opening an input method for Text or TextField.

Figure 11−2 shows one possible program window with a Text widget using over−the−spot interaction for Japanese text input. The status area indicates that phonetic input is in use and insert mode is enabled. The pre−edit area shows that the letter "H" has been entered. Since there is no Hiragana phonetic equivalent, the "H" appears in the pre−edit window.

Figure 1 Text Widget Pre−Edit and Status Areas Using Over−the−Spot

Figure 11–3 shows the same window after a "u" has been entered following the "H" shown in Figure 11–2.

Figure 2 Text Widget Pre–Edit Area After Next Character Entry



Here the pre–edit area is displaying the phonetic equivalent of the English letters "hu" in Hiragana.

When on–the–spot input style is used, a pre–edit string is displayed in the text widget window. This preedit string is considered part of the text widget value, and its integrity is ensured by the verify callbacks of the text widget. If the verify callbacks of the text widget do not accept any part of the preedit buffer, the preedit string is committed. The following actions also cause the text widget to commit the preedit string before performing the specified action:

Table 1 Highlight Modes

Commit Actions
cut
paste
selection
cursor movement
commit key

Note: Note: Cursor movement may be interpreted by the input method as cursor movement within the preedit buffer. If this is the case, the preedit buffer may not be committed. This behavior is completely dependent upon the implementation of the input method.

In the case of a shared XIC, the widgets that share the XIC shall retain the preedit buffer, if any, and preedit state when focus is switched between the widgets that share the XIC.

When the preedit buffer is active, it may be highlighted. This highlight value can be set by the input method server. The following mapping table relates Text widget highlighting modes to the input method highlighting feedbacks:

Table 2 Highlight Modes

XIMFeedback	**XmHighlightMode**
XIMReverse	XmHIGHLIGHT_SELECTED
XIMUnderline	XmHIGHLIGHT_SECONDARY_SELECTED
XIMHighlight	XmHIGHLIGHT_NORMAL
XIMPrimary	XmHIGHLIGHT_SELECTED
XIMSecondary	XmHIGHLIGHT_SECONDARY_SELECTED
XIMTertiary	XmHIGHLIGHT_SELECTED

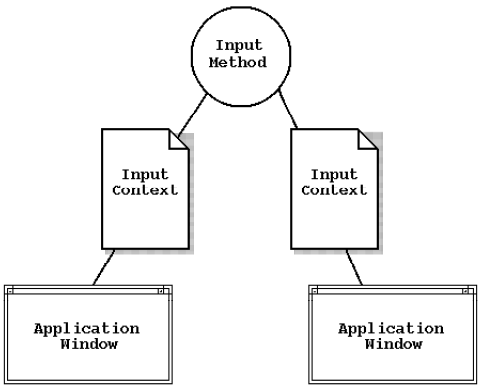
In normal insertion mode, original data in the text widget is shifted out to give way for the preedit string at the insertion point.

In overstrike mode, the preedit string will replace the same number of characters, if available before the end of the text widget value, at the insertion point during the preedit process.

Input Context

An *input context* is the mechanism used to provide the state information needed to manage the information flow between the application and the input method. It is a combination of an input method, a locale specifying the encoding of character strings to be returned, an application window, and internal state information. The relationship between the input method and its input contexts is roughly comparable to that between the display and its windows. The following figure shows the relationships involved. The input method is determined by the **XmNinputMethod** resource of the nearest ancestor **VendorShell**, or by the locale specified by the application user.

Figure 3 Input Method and Input Contexts



Input Method Functions

The widgets in the Motif widget set are equipped to choose the appropriate input method based on the specified locale. This process is transparent to the user, as well as to the programmer of most applications. Most programmers will not need any more than to know what to expect under different values of the **XmNpreeditType** resource. Occasionally, however, a programmer will require direct access to the input method. For example, in order to write a widget that accepts typed input, the input method must be explicitly nominated. Motif makes available a high−level interface to the basic Xlib input method functions.

> **Note:** The following XmIm functions are made available only for the convenience of the authors of new widgets. Except for the **XmDrawingArea** widget, these routines should *not* be used with existing widgets.

- **XmImRegister**
- **XmImUnregister**
- **XmImGetXIM**
- **XmImCloseXIM**

A widget must be registered with an input method and context before an application can use **XmImMbLookupString** to retrieve multibyte text from the input method. The **XmImRegister** function is the usual method for selecting and allocating an input method for a widget. It starts from the given widget, and searches up the widget hierarchy until it finds a shell that has an **XmNinputMethod** resource. This resource is only defined for the **VendorShell** widget. If there is an open input method for this display, the current widget will be attached to it. If not, a new input method, specified by the **XmNinputMethod** resource, will be opened and attached to the **XmDisplay**. If the **XmNinputMethod** is NULL, or unspecified, the input method corresponding to the current locale will be chosen and opened. Motif will only support a single input method per application.

XmImRegister and **XmImSetValues** (or **XmImSetFocusValues**) do not return the **XIM** and **XIC** data structures to the calling application, but maintain the connections between the widget and its input method and input context in an internal registry until **XmImUnregister** is called. This will then permit the application to use **XmImMbLookupString** and **XmImSetFocus** functions, which take the widget as an argument.

The two **XmImRegister** and **XmImUnregister** functions, as well as and **XmImSetValues** (or **XmImSetFocusValues**), are the only input manager functions needed to establish input methods and contexts for all but the most unusual applications. These are the only calls needed to implement the Motif Text and TextField widgets.

Like the **XmImRegister** function, the **XmImGetXIM** function searches from the given widget, up the widget hierarchy, until it finds a shell that has an **XmNinputMethod** resource. If there is an open input method for the display, it will simply be returned. Otherwise, a new input method is opened, as specified by the **XmNinputMethod** resource found, and attached to the **XmDisplay**. If the **XmNinputMethod** is NULL, or unspecified, the input method corresponding to the current locale will be chosen and opened. Motif will only support a single input method per application. The **XmImRegister**, **XmImUnregister**, and **XmImSetValues** (or **XmImSetFocusValues**) functions will suffice for most applications. Use **XmImGetXIM** and **XmImCloseXIM** when the widget needs a different input policy than its parent, or needs access to the input method data structure.

Use **XmImCloseXIM** to release the input method associated with the input widget's **XmDisplay**. The function also frees all the input contexts associated with the input method, and their associated memory, and unregisters all widgets associated with the freed input contexts. To close only the input context for one widget, use **XmImUnregister** or **XmImFreeXIC**.

The following functions handle the creation and deletion of an input context.

- **XmImGetXIC**
- **XmImSetXIC**
- **XmImFreeXIC**

Use the **XmImGetXIC** function to create and register a new input context for a widget. A new input context is not required if the current input policy is **XmPER_SHELL** and an open input policy already exists. In this case, **XmImGetXIC** registers the input widget with the existing input context, and returns the shared **XIC**. The **XmImSetValues** function is equivalent to calling the **XmImGetXIC** function with a NULL argument list and an input policy of **XmINHERIT_POLICY**. Unlike **XmImRegister**, **XmImGetXIC** returns the created **XIC** data structure. Use **XmImGetXIC** to override the default input policy, or to specify new values for input context parameters.

The **XmImSetXIC** function allows the application to provide other input contexts to use with the widget and to access the current registered input context data structure. The input **XIC** is registered as the current **XIC**, and any **XIC** previously registered with the input widget is removed. The function returns the newly registered **XIC**. If the input **XIC** argument is NULL, **XmImSetXIC** simply returns the currently registered **XIC**.

The **XmImFreeXIC** function unregisters all widgets currently registered with a particular input context. It then removes the **XIC** itself, and frees the memory allocated to it.

These functions may be used to modify an existing input context:

- **XmImSetValues**
- **XmImSetFocusValues**
- **XmImUnsetFocus**

Use **XmImSetValues** to create an input context, or to modify an existing one. If the current state of the input context does not allow modification, the **XIC** will be unregistered and deleted, and a new one will be created and registered with the input widget. Also, if there is not yet an **XIC** for the widget, one will be created using the given parameters.

The **XmImSetFocusValues** function is nearly identical to the **XmImSetValues** function, except that, after the input context values have been reset, the input focus window for the **XIC** is set to the window of the input widget.

Use the **XmImUnsetFocus** function to unset the focus window for any **XIC** registered to the input widget. If the focus window is not set, this function has no effect.

The following function is used to receive data from the input context:

- **XmImMbLookupString**

The **XmImMbLookupString** function is the heart of the input method. On input, it accepts a widget and a KeyEvent. Using the input context registered with the given widget, the function returns a buffer of multibyte text, a keysym computed for the event, and the length of the string in the output buffer.

Note that, if the key event did not complete some necessary pre−edit sequence, the length of the returned string may be zero. For example, in Figure 11−2, the key event that produced the "H" shown, since it does not specify a unique Hiragana character, would produce a zero−length return from **XmImMbLookupString**. The subsequent key event, shown in Figure 11−3, produces a "u." **XmImMbLookupString** can now map the English letters "Hu" to a unique Hiragana character, so that character is returned to the application, presumably to be drawn in some appropriate place.

Input and the Motif Text Widgets

The Motif Text and TextField widgets, when editable, provide a transparent connection to the locale−specific input method for text input. The application programmer specifies an appropriate font set in the text widget's **XmNfontList** resource and creates the widget as a descendant of **VendorShell**. **VendorShell** provides geometry management of the status and pre−edit areas. It also supplies a visual separator between the status area window and the application's top level window.

Setting the **VendorShell** resource **XmNpreeditType** dictates the location of the input method window. With an off−the−spot input method, the pre−edit and status area windows appear at the bottom of the application window.

Text Input Using a DrawingArea

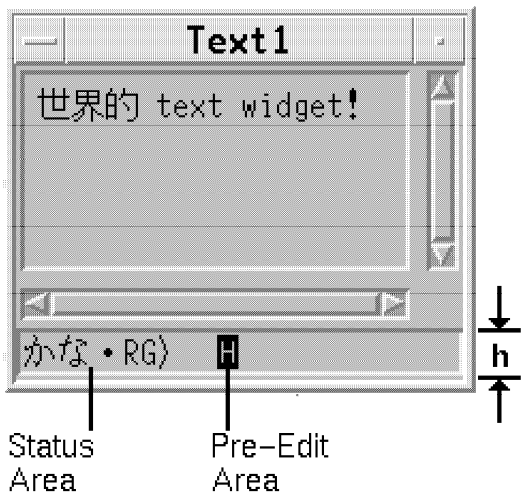
An application that needs special text processing may use a DrawingArea for text input and output. For internationalized text input with any widget other than the various Motif text widgets, the application must use the **XmIm** input method facilities. These allow the application to open an input method and input context and to obtain input from the input method. When using these facilities, an application may also need to handle input method geometry management, focus management, event filtering, and other issues.

Geometry Management of Pre−Edit and Status Areas

When an off−the−spot input method is used with the Text or TextField widget, the pre−edit and status areas are below the client's main window but inside the **VendorShell**. **VendorShell** accomplishes the necessary geometry management. If the application uses either **XtGetValues** or **XtSetValues** to get or set the height (**XmNheight**) of **VendorShell**, the height includes the height of the input method area.

The following figure shows a Text widget using an off−the−spot input method. The distance "h" is the additional height that the input manager needs to display the status and pre−edit areas. Note that, in off−the−spot, the pre−edit area is at the bottom of the interaction.

Figure 4 Text Widget Pre−Edit and Status Areas Using Off−the−Spot


Text1
世界的 text widget!
かな • RG)
h
Status
Area
Pre-Edit
Area

Compound Strings and Compound Text

Compound text is the standard format for exchanging textual data between X window system applications. This is necessary when the user moves text displayed in one code set to another window with text in a different code set. For example, the following figure shows two windows: one titled "UJIS" and the other titled "Shift JIS."

Figure 1 Reason for Compound Text

Both windows represent a Motif Text widget, one with some Japanese UJIS characters displayed, and the other with some Shift JIS characters. If the user wants to cut text from one window and paste it in the other window, compound text is used to pass data between the two. The Motif Text widget does this automatically.

If one of the widgets in the previous figure is a Label widget instead of a Text widget, a different situation exists. This is because the Label widget has its text data in compound string format, while the Text widget data is a simple character string. In order to pass text data between a Text or TextField widget and any other widget, the application needs to convert the compound string to compound text.

Motif has two functions, **XmCvtXmStringToCT** and **XmCvtCTToXmString**, for converting between compound strings and compound text.

XmCvtXmStringToCT converts a compound string to compound text. The converter uses the font list tag associated with a given compound string text component to select a compound text format for that component. A registry defines a mapping between font list tags and compound text encoding formats. The converter uses the following algorithm for each compound string text component:

1. If the associated tag is mapped to **XmFONTLIST_DEFAULT_TAG** in the registry, the converter passes the text of the compound string component to **XmbTextListToTextProperty** with an encoding style of **XCompoundTextStyle** and uses the resulting compound text for that component.
2. If the associated tag is mapped to an MIT registered charset in the registry, the converter creates the compound text for that component by using the charset (from the registry) and the text of the compound string component as defined in the X Consortium Standard *Compound Text Encoding*.
3. If the associated tag is mapped to a charset in the registry that is neither **XmFONTLIST_DEFAULT_TAG** nor an MIT registered charset, the converter creates the compound text for that component by using the charset (from the registry) and the text of the compound string component as an "extended segment" with a variable number of octets per character.
4. If the associated tag is not mapped in the registry, the result is implementation dependent.

An application can use **XmRegisterSegmentEncoding** to map a font list element tag to a compound text encoding format. For example, the application may be using a font list element tag of "BOLD" to identify a compound text component consisting of localized text to be displayed in a bold font. To ensure that the component is treated as localized text when converted to compound text, the tag "BOLD" should be mapped to

XmFONTLIST_DEFAULT_TAG as follows:

```
char *old_encoding = XmRegisterSegmentEncoding("BOLD",
                        XmFONTLIST_DEFAULT_TAG);
XtFree(old_encoding);
```

The following functions may be used both to convert text in a Motif compound string into the compound text format, and to create Motif compound strings from compound text.

- **XmCvtCTToXmString**
- **XmCvtXmStringToCT**

XmCvtCTToXmString converts compound text to a compound string. This function is implementation dependent. There is also the reverse function, called **XmCvtXmStringToCT**.

The following example uses the compound text format to change a window title to a string in the language of the locale. Of course, the language environment must be properly set, and the strings used must also translate properly into the target language using the locale's code set. This example uses the EUC coding of Japanese, and produces a **XmNtitle** reading "Information" and a **XmNdialogTitle** reading "Unsaved Changes." The following example consists of two pieces: a fragment from an X resource file and a piece of C code. First the X resource file fragment:

```
mwm*renderTable.fontName:       -*-fixed-medium-r-normal--*-150-*
mwm*renderTable.fontType:       XmFONT_IS_FONTSET
```

and now the C code:

```
Widget     toplevel;
Widget     dialog;
Arg        ArgList[10];
int        n;
Atom       atom;
XmString   compound_string1, compound_string2;
char      *compound_text;

    /* Set compound_string1 to "Information" (EUC coding) */
    compound_string1 = XmStringCreateLocalized("%$%s%U%)%a!<%7%g%s");
    /* Set compound_string2 to "Unsaved Changes" (EUC coding) */
    compound_string2 = XmStringCreateLocalized("JQ99$rJ]B8$7$F$$$[caret ]$;$s!#");

    atom = XmInternAtom(XtDisplay(toplevel), "COMPOUND_TEXT", False);

    compound_text = XmCvtXmStringToCT(compound_string1);

    n = 0;
    XtSetArg(ArgList[n], XmNtitle, compound_text); n++;
    XtSetArg(ArgList[n], XmNtitleEncoding, atom); n++;
    XtSetArg(ArgList[n], XmNiconName, compound_text); n++;
    XtSetArg(ArgList[n], XmNiconNameEncoding, atom); n++;
    XtSetValues(toplevel, ArgList, n);

    n = 0;
    XtSetArg(ArgList[n], XmNdialogTitle, compound_string2); n++;
    XtSetValues(dialog, ArgList, n);
```

See Chapter 16 for more information on transferring data between applications. The compound text format is described in the X Consortium Standard *Compound Text Encoding*.