TriTeal Enterprise Desktop (TED™) 4.0

Internationalization Programmer's Guide

Printed in USA

Part No. TED007004000

Printing History
July 1995 ... First Edition
Printed in the U.S.A

Sales Information and Technical Support

TriTeal Corporation
2011 Palomar Airport Road, Suite 200
Carlsbad, CA 92009
USA

Tel: (619) 930-2077
Fax: (619) 930-2081
Support E-Mail: support@triteal.com
Sales E-Mail: info@triteal.com
World Wide Web: http://www.triteal.com/

Please
Recycle

Contents

Figures

Tables

Preface

The *TriTeal Enterprise Desktop: Internationalization Programmer's Guide* provides information for internationalizing the desktop, enabling applications to support various languages and cultural conventions in a consistent user interface.

Specifically, this guide:

- Provides guidelines and hints for developers on how to write applications for worldwide distribution.
- Provides an overall view of internationalization topics that span different layers within the desktop.
- Provides pointers to reference and more detailed documentation. In some cases, standard documentation is referenced.

This guide is not intended to duplicate the existing reference or conceptual documentation but rather to provide guidelines and conventions on specific internationalization topics. This document focuses on internationalization topics and not on any specific component or layer in an open software environment.

Who Should Use This Book

This book provides various levels of information for the application programmer and developer and related fields.

How This Book Is Organized

Explanations of the contents of this book follow:

Chapter 1, "Introduction to Internationalization," provides an overview of internationalization and localizing within the desktop, including locales, fonts, drawing, inputting, interclient communication, and extracting user visual text. Information on the significance of internationalization standards is also provided.

Chapter 2, "Internationalization and the TriTeal Enterprise Desktop," covers the set of topics that developers commonly need to consider when internationalizing their applications, including locale management, localized resources, font management, localized text tasks, interclient communication for localized text, and internationalized functions.

Chapter 3, "Internationalization and Distributed Networks," discusses topics related to handling encoded characters in distributed networks. Basic principles and examples for interclient interoperability are provided to guide developers in internationalized distributed environments.

Chapter 4, "Motif Dependencies," topics include internationalized applications, locale management, localized text, international User Interface Language (UIL), and localized applications.

Chapter 5, "Xt and Xlib Dependencies," topics include locale management, localized text tasks, font set metrics, interclient communications conventions for localized text, and charset and font set encoding and registry information.

Appendix A, "Message Guidelines," is a set of guidelines for writing messages.

Related Publications

See the following documentation for additional information on topics presented in this book:

- ISO C: ISO/IEC 9899: 1990, *Programming Languages --- C* (technically identical to ANS X3.159-1989, Programming Language C).
- ISO/IEC 9945-1: 1990, (IEEE Standard 1003.1) *Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].*
- ISO/IEC DIS 9945-2: 1992, (IEEE Standard 1003.2-Draft) *Information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities*.
- OSF/Motif 1.2: *OSF Motif 1.2 Programmer's Reference*, *Revision 1.2*, Open Software Foundation, Prentice Hall, 1992, ISBN: 0-13-643115-1.
- Scheifler, W. R., *X Window System, The Complete Reference to Xlib, Xprotocol, ICCCM, XLFD* - X Version 11, Release 5, Digital Press, 1992, ISBN: 1-55558-088-2.
- X/Open: *X/Open CAE Specification System Interface Definition*, Issue 4, X/Open Company Ltd., 1992, ISBN: 1-872630-46-4.
- X/Open: *X/Open CAE Specification Commands and Utilities*, Issue 4, X/Open Company Ltd., 1992, ISBN: 1-872630-48-0.
- X/Open: *X/Open CAE Specification System Interface and Headers*, Issue 4, X/Open Company Ltd., 1992, ISBN: 1-872630-47-2.
- X/Open: *X/Open Internationalization Guide*, X/Open Company Ltd., 1992, ISBN: 1-872630-20-0.
- ISO/IEC 10646-1: 1993 (E): *Information Technology - Universal Multi-Octet Coded Character Set (UCS). Part 1: Architecture and Basic Multilingual Plane*.

What Typographic Changes and Symbols Mean

Table P-1 describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. system% You have mail.
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

Introduction to Internationalization 1

Internationalization is the designing of computer systems and applications for users around the world. Such users have different languages and may have different requirements for the functionality and user interface of the systems they operate. In spite of these differences, users want to be able to implement enterprise-wide applications that run at their sites worldwide. These applications must be able to *interoperate* across country boundaries, run on a variety of hardware configurations from multiple vendors, and be localized to meet local users' needs. This open, distributed computing environment is the reasoning behind common open software environments. The internationalization technology identified within this specification provides these benefits to a global market.

Overview of Internationalization

Multiple environments may exist within a common open system for support of different national languages. Each of these national environments is called a *locale,* which considers the language, its characters, fonts, and the customs used to input and format data. The Common Desktop Environment is fully internationalized such that any application can run using any locale installed in the system.

A locale defines the behavior of a program at run time according to the language and cultural conventions of a user's geographical area. Throughout the system, locales affect the following:

- Encoding and processing of text data.
- Identifying the language and encoding of resource files and their text values.
- Rendering and layout of text strings.
- Interchanging text that is used for interclient text communication.
- Selecting the input method (which code set will be generated) and the processing of text data.
- Encoding and decoding for interclient text communication.
- Bitmap/icon files.
- Actions and file types.
- User Interface Definition (UID) files.

An internationalized application contains no code that is dependent on the user's locale, the characters needed to represent that locale, or any formats (such as date and currency) that the user expects to see and interact with. The desktop accomplishes this by separating language- and culture-dependent information from the application and saving it outside the application.

Figure 1-1 shows the kinds of information that should be external to an application to simplify internationalization.

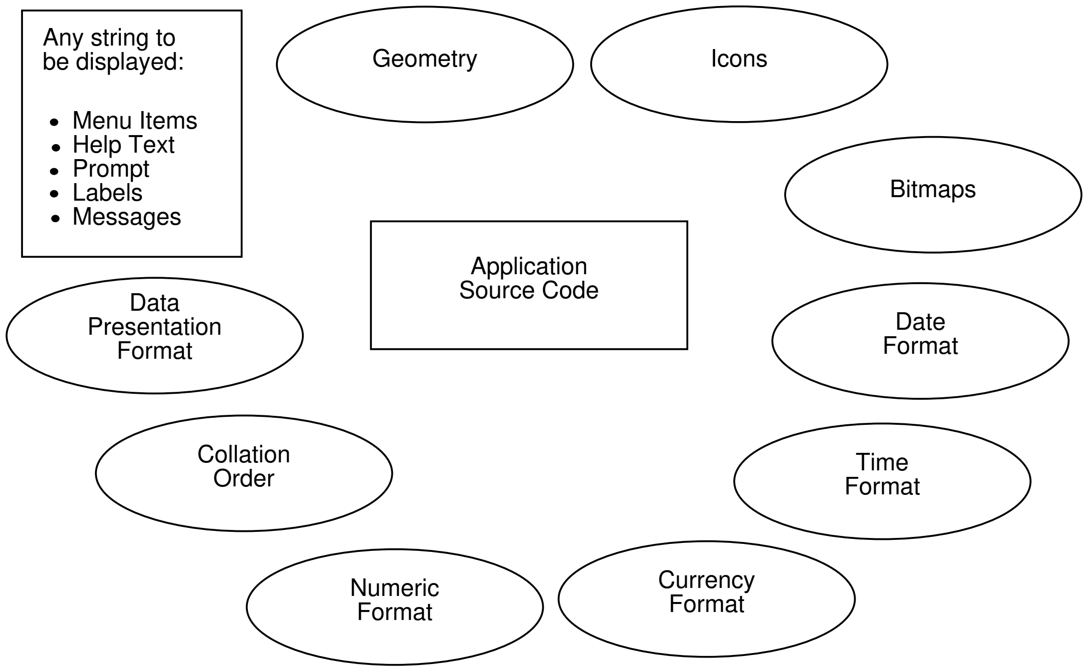


Figure 1-1 Information external to the application

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

An internationalized application is also adaptable to the requirements of different native languages, local customs, and character-string encodings. The process of adapting the operation to a particular native language, local custom, or string encoding is called *localization*. A goal of internationalization is to permit localization without program source modifications or recompilation.

For a quick overview of internationalization, refer to *X/Open CAE Specification System Interface Definition*, Issue 4, X/Open Company Ltd., 1992, ISBN: 1-872630-46-4.

Current State of Internationalization

Previously, the industry supplied many variants of internationalization from proprietary functions to the new set of standard functions published by X/Open. Also, there have been different levels of enabling, such as simple ASCII support, Latin/European support, Asian multibyte support, and Arabic/Hebrew bidirectional support.

The interfaces defined within the X/Open specification are capable of supporting a large set of languages and territories, including:

Script	**Description**
Latin Language	Americas, Eastern/Western European
Greek	Greece
Turkish	Turkey
East Asia	Japanese, Korean, and Chinese
Indic	Thai
Bidirectional	Arabic and Hebrew

Furthermore, the goal of the Common Desktop Environment is that localization of these technologies (translation of messages and documentation and other adaptation for local needs) be done in a consistent way, so that a supported user anywhere in the world will find the same *common localized environment* from vendor to vendor. End users and administrators can expect a consistent set of localization features that provide a complete application environment for support of global software.

Internationalization Standards

Through the work of many companies, the functionality of the internationalization application program interface has been standardized over time to include additional requirements and languages, particularly those of East Asia. This work has been centered primarily in the Portable Operating

System Interface for Computer Environments (POSIX) and X/Open specifications. The original X/Open specification was published in the second edition of the *X/Open Portability Guide* (XPG2) and was based on the Native Language Support product released by Hewlett-Packard. The latest published X/Open internationalization standard is referred to as XPG4.

It is important that each layer within the desktop use the proper set of standards interfaces defined for internationalization to ensure end users get a consistent, localized interface. The definition of a locale and the common open set of locale-dependent functions are based on the following specifications:

- *X Window System, The Complete Reference to Xlib, Xprotocol, ICCCM, XLFD - X Version, Release 5*, Digital Press, 1992, ISBN 1-55558-088-2.
- *ANSI/IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE.
- *OSF™ Motif 1.2 Programmer's Reference, Revision 1.2*, Open Software Foundation, Prentice Hall, 1992, ISBN 0-13-643115-1.
- *X/Open CAE Specification Commands and Utilities*, Issue 4, X/Open Company Ltd., 1992, ISBN 1-872630-48-0.

Within this environment, software developers can expect to develop *worldwide applications* that are portable, can interoperate across distributed systems (even from different vendors), and can meet the diverse language and cultural requirements of multinational users supported by the desktop standard locales.

Common Internationalization System

Figure 1-2 on page 7 shows a view of how internationalization is pervasive across a specific single-host system. The goal is that the applications (*clients*) are built to be shipped worldwide for the set of locales supported in the underlying system. Using standard interfaces improves access to global markets and minimizes the amount of localization work needed by application developers. Country representatives can be ensured of consistent localization within systems adhering to the principles of the desktop.

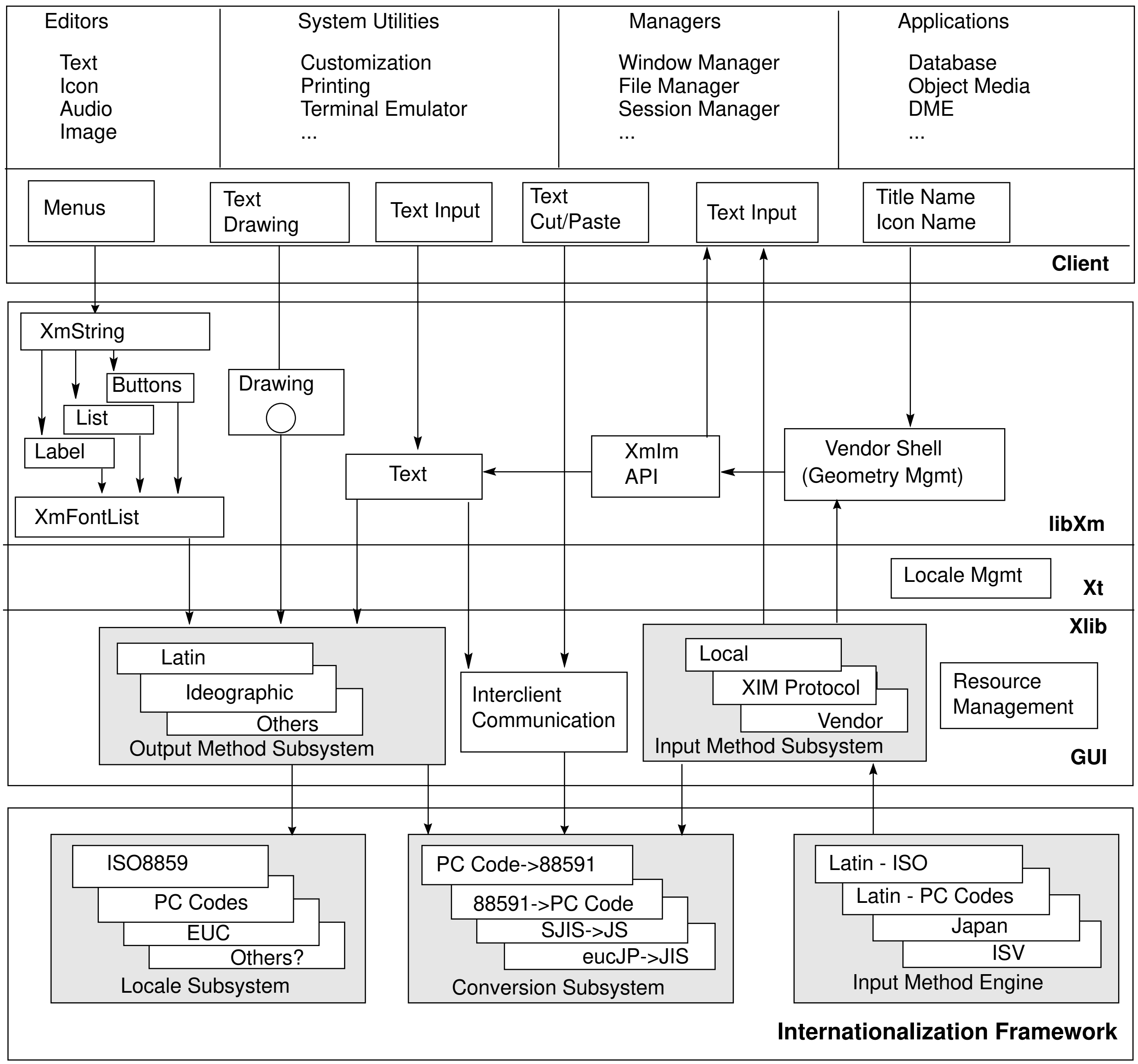


Figure 1-2 Common internationalized system

Locales

Most single-display clients operate in a single locale that is determined at run time from the setting of the environment variable, which is usually `$LANG` or the `xnlLanguage` resource. Locale environment variables, such as `LC_ALL`, `LC_CTYPE`, and `LANG`, can be used to control the environment. See "Xt Locale Management" on page 104 for more information.

The `LC_CTYPE` category of the locale is used by the environment to identify the locale-specific features used at run time. The fonts and input method loaded by the toolkit are determined by the `LC_CTYPE` category.

Programs that are enabled for internationalization are expected to call the `XtSetLanguageProc()` function (which calls `setlocale()` by default) to set the locale desired by the user. None of the libraries call the `setlocale()` function to set the locale, so it is the responsibility of the application to call `XtSetLanguageProc()` with either a specific locale or some value loaded at run time. If applications are internationalized and do not use `XtSetLanguageProc()`, obtain the locale name from one of the following prioritized sources to pass it to the `setlocale()` function:

- A command-line option
- A resource
- The empty string ("")

The empty string makes the `setlocale()` function use the `$LC_*` and `$LANG` environment variables to determine locale settings. Specifically, setlocale (`LC_ALL`, "") specifies that the locale should be checked and taken from environment variables in the order shown in Table 1-1 for the various locale categories.

Table 1-1 Locale Categories

Category	1st Env. Var.	2nd Env. Var.	3rd Env. Var.
`LC_CTYPE:`	`LC_ALL`	`LC_TYPE`	`LANG`
`LC_COLLATE:`	`LC_ALL`	`LC_COLLATE`	`LANG`
`LC_TIME:`	`LC_ALL`	`LC_TIME`	`LANG`

Table 1-1 Locale Categories

Category	1st Env. Var.	2nd Env. Var.	3rd Env. Var.
LC_NUMERIC:	LC_ALL	LC_NUMERIC	LANG
LC_MONETARY:	LC_ALL	LC_MONETARY	LANG
LC_MESSAGES:	LC_ALL	LC_MESSAGES	LANG

The toolkit already defines a standard command-line option (`-lang`) and a resource (`xnlLanguage`). Also, the resource value can be set in the server `RESOURCE_MANAGER`, which may affect all clients that connect to that server.

Fonts, Font Sets, and Font Lists

All X clients use fonts for drawing text. The basic object used in drawing text is `XFontStruct`, which identifies the font that contains the images to be drawn.

The desktop already supports fonts by way of the `XFontStruct` data structure defined by Xlib; yet, the encoding of the characters within the font must be known to an internationalized application. To communicate this information, the program expects that all fonts at the server are identified by an X Logical Font Description (XLFD) name. The XLFD name enables users to describe both the base characteristics and the charset (encoding of font glyphs). The term *charset* is used to denote the encoding of glyphs within the font, while the term *code set* means the encoding of characters within the locale. The charset for a given font is determined by the CharSetRegistry and CharSetEncoding fields of the XLFD name. Text and symbols are drawn as defined by the codes in the fonts.

A *font set* (for example, an `XFontSet` data structure defined by Xlib) is a collection of one or more fonts that enables all characters defined for a given locale to be drawn. Internationalized applications may be required to draw text encoded in the code sets of the locale where the value of an encoded character is not identical to the glyph index. Additionally, multiple fonts may be required to render all characters of the locale using one or more fonts whose encodings may be different than the code set of the locale. Since both code sets and charsets may vary from locale to locale, the concept of a font set is introduced through `XFontSet`.

While fonts are identified by their XLFD name, font sets are identified by a list of XLFD names. The list can consist of one or more XLFD names with the exception that only the base characteristics are significant; the encoding of the desired fonts is determined from the locale. Any charsets specified in the XLFD base name list are ignored and users need only concentrate on specifying the base characteristics, such as point size, style, and weight. A font set is said to be *locale-sensitive* and is used to draw text that is encoded in the code set of the locale. Internationalized applications should use font sets instead of font structs to render text data.

A *font list* is a libXm Toolkit object that is a collection of one or more font list entries. Font sets can be specified within a font list. Each font list entry designates either a font or a font set and is tagged with a name. If there is no tag in a font list entry, a default tag (`XmFONTLIST_DEFAULT_TAG`) is used. The font list can be used with the `XmString` functions found in the libXm Toolkit library. A font list enables drawing of compound strings that consist of one or more segments, each identified by a tag. This allows the drawing of strings with different base characteristics (for example, drawing a bold and italic string within one operation). Some non-`XmString`-based widgets, such as `XmText` of the libXm library, use only one font list entry in the font list. Motif font lists use the suffix : (colon) to identify a font set within a font list.

The user is generally asked to specify either a font list (which may contain either a font or font set) or a font set. In an internationalized environment, the user must be able to specify fonts that are independent of the code set because the specification can be used under various locales with different code sets than the character set (charset) of the font. Therefore, it is recommended that all font lists be specified with a font set.

Font Specification

The *font specification* can be either an X Logical Function Description (XLFD) name or an alias for the XLFD name. For example, the following are valid font specifications for a 14-point font:

```
-dt-application-medium-r-normal-serif-*-*-*-*-p-*-iso8859-1
```

OR

```
-*-r-*-14-*iso8859-1
```

Font Set Specification

The *font set specification* is a list of names (XLFD names or their aliases) and is sometimes called a *base name list*. All names are separated by commas, with any blank spaces before or after the comma being ignored. Pattern-matching (wildcard) characters can be specified to help shorten XLFD names.

Remember that a font set specification is determined by the locale that is running. For example, the ja_JP Japanese locale defines three fonts (character sets) necessary to display all of its characters; the following identifies the set of Gothic fonts needed.

- Example of full XLFD name list:

  ```
  -dt-mincho-medium-r-normal--14-*-*-m-*-jisx0201.1976-0,
  -dt-mincho-medium-r-normal--28-*-*-*-m-*-jisx0208.1983-0:
  ```

- Example of single XLFD pattern name:

  ```
  -dt-*-medium-*-*-*-m-*:
  ```

The preceding two cases can be used with a Japanese locale as long as fonts exist that match the base name list.

Font List Specification

A *font list specification* can consist of one or more entries, each of which can be either a font specification or a font set specification.

Each entry can be tagged with a name that is used when drawing a compound string. The tags are application-defined and are usually names representing the expected style of font; for example, `bold`, `italic`, `bigbold`. A null tag is used to denote the default entry and is associated with the `XmFONTLIST_DEFAULT_TAG` identifier used in `XmString` functions.

A font tag is identified when it is prefixed with an = (equal sign); for example, `=bigbold` (this matches the first font defined at the server). If an = is specified but there is no name following it, the specification is considered the *default font list entry*.

A font set tag is identified when it is prefixed with a : (colon); for example, `:bigbold` (this matches the first server set of fonts that satisfy the locale). If a : is specified but no name is given, the specification is considered the default font list entry. Within a font list entry specification, a base name list is separated by ; (semicolons) rather than by , (commas).

Example Font List Specification

For the Latin 1 locales, enter:

```
-*-r-*-14-*: ,\# default font list entry
-*-b-*-18-*:bigbold# Large Bold fonts
```

Base Font Name List Specification

The *base font name list* is a list of base font names associated with a font set as defined by the locale. The base font names are in a comma-separated list and are assumed to be characters from the portable character set; otherwise, the result is undefined. Blank space immediately on either side of a separating comma is ignored.

Use of XLFD font names permits international applications to obtain the fonts needed for a variety of locales from a single locale-independent base font name. The single base font name specifies a family of fonts whose members are encoded in the various charsets needed by the locales of interest.

An XLFD base font name can explicitly name the font's charset needed for the locale. This enables the user to specify an exact font for use with a charset required by a locale, fully controlling the font selection.

If a base font name is not an XLFD name, an attempt is made to obtain an XLFD name from the font properties for the font.

The following algorithm is used to select the fonts that are used to display text with font sets.

For each charset required by the locale, the base font name list is searched for the first of the following cases that names a set of fonts that exist at the server.

- The first XLFD-conforming base font name that specifies the required charset or a superset of the required charset in its CharSetRegistry and CharSetEncoding fields.
- The first set of one or more XLFD-conforming base font names that specify one or more charsets that can be remapped to support the required charset. The Xlib implementation can recognize various mappings from a required charset to one or more other charsets and use the fonts for those charsets. For example, JIS Roman is ASCII with the ~ (tilde) and \ (backslash)

characters replaced by the yen and overbar characters; Xlib can load an ISO8859-1 font to support this character set if a JIS Roman font is not available.

- The first XLFD-conforming font name, or the first non-XLFD font name for which an XLFD font name can be obtained, combined with the required charset (replacing the CharSetRegistry and CharSetEncoding fields in the XLFD font name). In the first instance, the implementation can use a charset that is a superset of the required charset.
- The first font name that can be mapped in some locale-dependent manner to one or more fonts that support imaging text in the charset.

For example, assume a locale requires the following charsets:

- ISO8859-1
- JISX0208.1983
- JISX0201.1976
- GB2312-1980.0

You can supply a base font name list that explicitly specifies the charsets, ensuring that specific fonts are used if they exist, as shown in the following example:

```
"-dt-mincho-Medium-R-Normal-*-*-*-*-*-M-*-JISX0208.1983-0,\
-dt-mincho-Medium-R-Normal-*-*-*-*-*-M-\
*-JISX0201.jisx0201\.1976-1,\
-dt-song-Medium-R-Normal-*-*-*-*-*-M-*-GB2312-1980.0,\
-*-default-Bold-R-Normal-*-*-*-*-M-*-ISO8859-1"
```

You can supply a base font name list that omits the charsets, which selects fonts for each required code set, as shown in the following example:

```
"-dt-Fixed-Medium-R-Normal-*-*-*-*-*-M-*,\
-dt-Fixed-Medium-R-Normal-*-*-*-*-*-M-*,\
-dt-Fixed-Medium-R-Normal-*-*-*-*-*-M-*,\
-*-Courier-Bold-R-Normal-*-*-*-*-M-*"
```

Alternatively, the user can supply a single base font name that selects from all available fonts that meet certain minimum XLFD property requirements, as shown in the following example:

```
"-*-*-*-R-Normal--*-*-*-*-*-M-*"
```

Text Drawing

The desktop provides various functions for rendering localized text, including simple text, compound strings, and some widgets. These include functions within the Xlib and Motif libraries.

Input Methods

The Common Desktop Environment provides the ability to enter localized input for an internationalized application that is using the Xm Toolkit. Specifically, the `XmText[Field]` widgets are enabled to interface with input methods provided by each locale. In addition, the `dtterm` client is enabled to use input methods.

By default, each internationalization client that uses the libXm Toolkit uses the input method associated with a locale specified by the user. The `XmNinputMethod` resource is provided as a modifier on the locale name to allow a user to specify any alternative input method.

The user interface of the input method consists of several elements. The need for these areas is dependent on the input method being used. They are usually needed by input methods that require complex input processing and dialogs. See Figure 1-3 for an illustration of these areas.

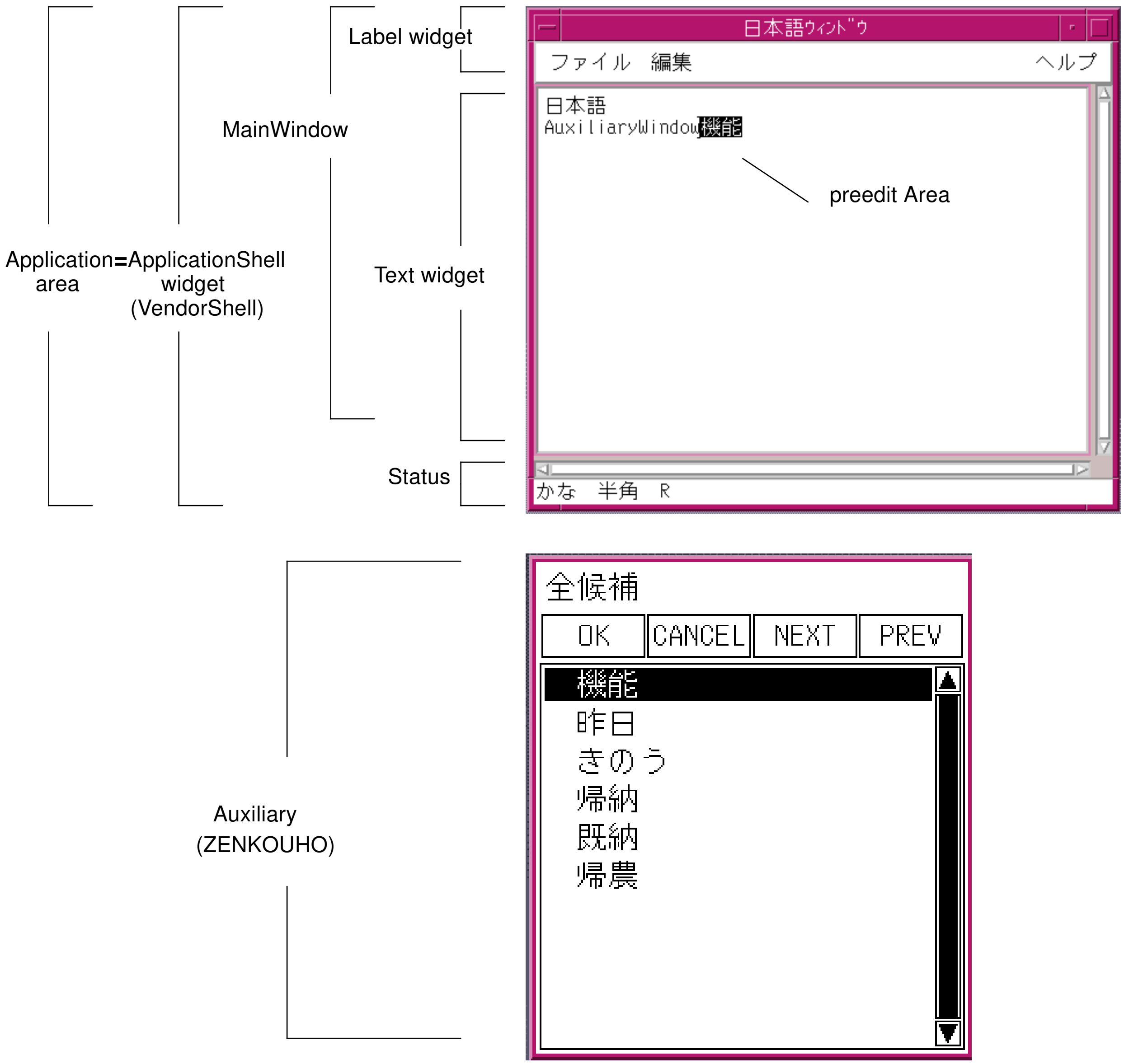


Figure 1-3 Example of VendorShell widget with auxiliary (Japanese)

preedit Area

A preedit area is used to display the string being preedited. The input method supports four modes of preediting: OffTheSpot, OverTheSpot (default), Root, and None.

Note – A string that has been committed *cannot* be reconverted. The status of the string is moved from the preedit area to the location where the user is entering characters.

OffTheSpot

In OffTheSpot style preediting using an input method, the location of preediting is fixed at just below the MainWindow area and on the side of the status area as shown in Figure 1-4. A Japanese input method is used for the example.

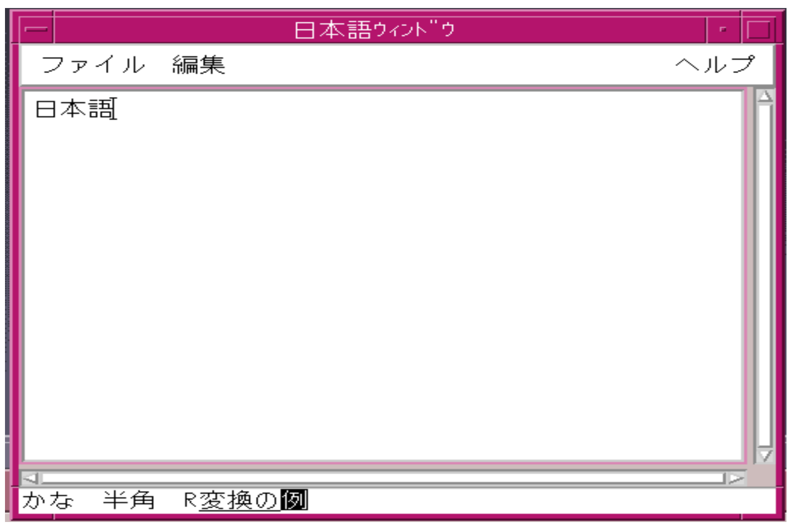


Figure 1-4 Example of OffTheSpot preediting with the VendorShell widget (Japanese)

In the system environment, when preediting using an input method, the preedit string being preedited may be highlighted in some form depending on the input method.

To use OffTheSpot style, set the XmNpreeditType resource of the VendorShell widget either with the XtSetValues() function or with a resource file. The XmNpreeditType resource can also be set as the resource of a TopLevelShell, ApplicationShell, or DialogShell widget, all of which are subclasses of the VendorShell widget class.

OverTheSpot (Default)

In OverTheSpot style, the location of the preedit area is set to where the user is entering characters (for example, the insert cursor position of the Text widget that has the current focus). The characters in a preedit area are displayed at the cursor position as an overlay window, and they can be highlighted depending on the input method.

Although a preedit area may consist of multiple lines in OverTheSpot mode. The preedit area is always within the MainWindow area and cannot cross its edges in any direction. Keep in mind that although the preedit string under construction may be displayed as though it were part of the Text widget's text, it is not passed to the client and displayed in the underlying edit screen until preedit ends. See Figure 1-5 on page 16 for an illustration.

To use OverTheSpot mode explicitly, set the XmNpreeditType resource of the VendorShell widget either with the XtSetValues() function or with a resource file. The XmNpreeditType resource can be set as the resource of a TopLevelShell, ApplicationShell, or DialogShell widget because these are subclasses of the VendorShell widget class.

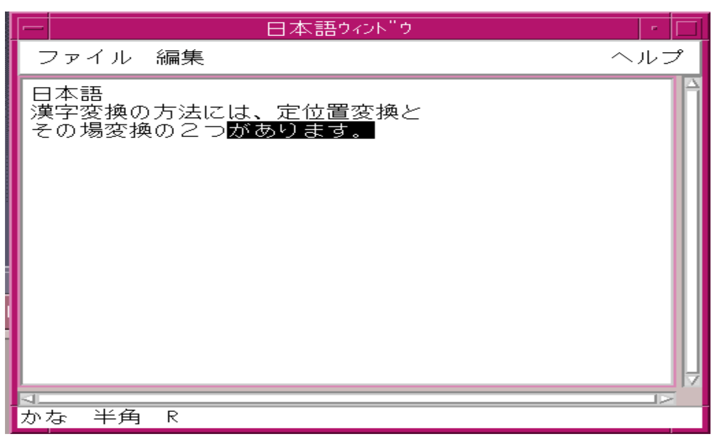


Figure 1-5 Example of OverTheSpot preediting with VendorShell widget (Japanese)

Root

In Root mode, the preedit and status areas are located separate from the client's window. The Root mode behavior is similar to OffTheSpot. See Figure 1-6 for an illustration.

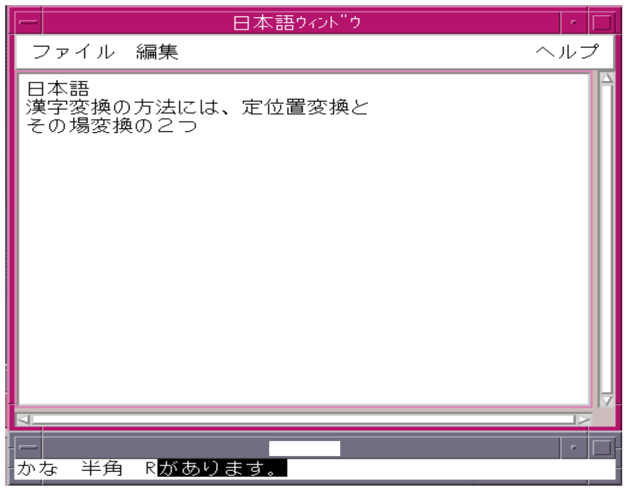


Figure 1-6 Example of Root preediting with the VendorShell widget (Japanese)

Status Area

A status area reports the input or keyboard status of the input method to the users. For OverTheSpot and OffTheSpot styles, the status area is located at the lower left corner of the VendorShell window.

- If Root style, the status area is placed outside the client window.
- If the preedit style is OffTheSpot mode, the preedit area is displayed to the right of the status area.

The `VendorShell` widget provides geometry management so that a status area is rearranged at the bottom corner of the VendorShell window if the VendorShell window is resized.

Auxiliary Area

An auxiliary area helps the user with preediting. Depending on the particular input method, an auxiliary area can be created. The Japanese input method in Figure 1-3 on page 14 creates the following types of auxiliary areas:

- ZENKOUHO
- JIS NUMBER
- Switching conversion method
  - SAKIYOMI-REN-BUNSETSU
  - IKKATSU-REN-BUNSETSU
  - TAN-BUNSETSU
  - FUKUGOU-GO

The aux area is managed by the input method and is used for any dialogs or menus required by the method.

MainWindow Area

A MainWindow area is the widget used as the working area of the input method. In the system environment, the sole child of the `VendorShell` widget is the `MainWindow` widget. It can be any container widget, such as a `RowColumn` widget. The user creates the container widget as the child of the `VendorShell` widget.

Focus Area

A focus area is any descendant widget under the `MainWindow` widget subtree that currently has focus. The Motif application programmer using existing widgets does not need to worry about the focus area. The important information to remember is that only one widget can have input method processing at a time. The input method processing moves to the window (widget) that currently has the focus.

Interclient Communications Conventions (ICCC)

The Interclient Communications Conventions (ICCC) defines the mechanism used to pass text between clients. Because the system is capable of supporting multiple code sets, it may be possible that two applications that are communicating with each other are using different code sets. ICCC defines how these two clients agree on how the data is passed between them. If two clients have incompatible character sets (for example, Latin1 and Japanese (JIS)), some data may be lost when characters are transported.

However, if two clients have different code sets but compatible character sets, ICCC enables these clients to pass information with no data lost. If code sets of the two clients are not identical, CompoundText encoding is used as the interchange with the `COMPOUND_TEXT` atom used. If data being communicated involves only portable characters (7-bit, ASCII, and others) or the ISO8859-1 code set, the data is communicated as is with no conversion by way of the `XA_STRING` atom.

Titles and icon names need to be communicated to the Window Manager using the `COMPOUND_TEXT` atom if nonportable characters are used; otherwise, the `XA_STRING` atom can be used. Any other encoding is limited to the ability to convert to the locale of the Window Manager. The Window Manager runs in a single locale and supports only titles and icon names that are convertible to the code set of the locale under which it is running.

The libXm library and all desktop clients should follow these conventions.

Internationalization and the TriTeal Enterprise Desktop 2

Multiple environments may exist within a common open system for support of different national languages. Each of these national environments is called a *locale,* which considers the language, its characters, fonts, and the customs used to input and format data. The TriTeal Enterprise Desktop is fully internationalized such that any application can run using any locale installed in the system.

Locale Management

For the desktop, most single-display clients operate in a single locale that is determined at run time from the setting of the environment variable, which is usually `$LANG`. The Xm library (libXm) can only support a single locale that is used at the time each widget is instantiated. Changing the locale after the Xm library has been initialized may cause unpredictable behavior.

All internationalized programs should set the locale desired by the user as defined in the locale environment variables. For programs using the desktop toolkit, the programs call the `XtSetLanguageProc()` function prior to calling any toolkit initialization function; for example, `XtAppInitialize()`. This function does all of the initialization necessary prior to the toolkit initialization. For nondesktop programs, the programs call the `setlocale()` function to set the locale desired by the user at the beginning of the program.

Locale environment variables (for example, `LC_ALL`, `LC_CTYPE`, and `LANG`) are used to control the environment. Users should be aware that the `LC_CTYPE` category of the locale is used by the X and Xm libraries to identify the locale-specific features used at run time. Yet, the `LC_MESSAGES` category is used by the message catalog services to load locale-specific text. Refer to “Extracting Localized Text” on page 40 for more information. Specifically, the fonts and input method loaded by the toolkit are determined by the setting of the `LC_CTYPE` category.

String encoding (for example, ISO8859-1 or Extended UNIX Code (EUC), in an application’s source code, resource files, and User Interface Language (UIL) files) should be the same as the code set of the locale where the application runs. If not, code conversion is required.

All components are shipped as a single, worldwide executable and are required to support the R5 sample implementation set of locales:
US, Western/Eastern Europe, Japan, Korea, China, and Taiwan.

Applications should be written so that they are code-set-independent and include support for any multibyte code set.

The following are the functions used for locale management:

- `XtSetLanguageProc()`
- `setlocale()`
- `XSupportsLocale()`
- `XSetLocaleModifiers()`

Font Management

When rendering text in an X Windows™ client, at least two aspects are sensitive to internationalization:

- Obtaining the localized text itself.
- Selecting the one or more fonts that contain all the glyphs needed to render the characters in the localized text.

"Extracting Localized Text" on page 40 describes how to choose the correct fonts to render localized text.

Matching Fonts to Character Sets

A font contains a set of glyphs used to render the characters of a locale. However, you may also want to do the following for a given locale:

- Determine the fonts needed.
- Specify the necessary fonts.
- Determine the charset of a font in a resource file.
- Choose multiple fonts per locale.

The last two fields of a font XFLD identify which glyphs are contained in a font and which value is used to obtain a specific glyph from the set. These last two fields identify the encoding of the glyphs contained in the font.

For example:

```
-adobe-courier-medium-r-normal--24-240-75-75-m-150-iso8859-1
```

The last two fields of this XLFD name are `iso8859` and `1`. These fields specify that the ISO8859-1 standard glyphs are contained in the font. Further, it specifies that the character code values of the ISO8859-1 standard are used to index the corresponding glyph for each character.

The font charset used by the application to render data depends on the locale you select. Because the font charset of the data changes is based on the choice of locale, the font specification must not be hardcoded by the application. Instead, it should be placed in a locale-specific `app-defaults` file, allowing localized versions of the `app-defaults` file to be created.

Further, the font should be specified as a fontset. A *fontset* is an Xlib concept in which an XLFD is used to specify the fonts. The font charset fields of the XLFD are specified by the Xlib code that creates the fontset and fills in these fields based on the locale that the user has specified.

For many languages (such as Japanese, Chinese, and Korean), multiple font charsets are combined to support single encoding. In these cases, multiple fonts must be opened to render the character data. Further, the data must be parsed into segments that correspond to each font, and in some cases, these segments must be transformed to convert the character values into glyphs indexes. The `XFontset`, which is a collection of all fonts necessary to render character data in a given locale, also deals with this set of problems. Further, a set of rendering and metric routines are provided that internally take care of breaking strings into character-set-consistent segments and transforming values into glyph indexes. These routines relieve the burden of the application developer, who needs only the user fontsets and the new X11R5 rendering and metric application program interfaces (APIs).

Font Objects

This section describes the following font objects:

- Font sets
- Fonts
- Font lists

Font Sets

Generally, all internationalized programs expecting to draw localized text using Xlib are required to use an `XmFontSet` for specifying the locale-dependent fonts. Specific fonts within a font set should be specified using XLFD naming conventions without the charset field specified. The resource name for an `XFontset` is `*fontSet`. Refer to “Localized Resources” on page 45 for a list of font resources.

Applications directly using Xlib to render text (as opposed to using `XmString` functions or widgets) may take advantage of the string-to-fontSet converter provided by Xt. For example, the following code fragment shows how to obtain a fontset when using Xt and when not using Xt:

```
/* pardon the double negative... means "If using Xt..." */
#ifndef NO_XT
typedef struct {
        XFontSet fontset;
            char *foo;
} ApplicationData,  *ApplicationDataPtr;
static XtResource my_resources[] = {
    { XtNfontSet, XtCFontSet, XtRFontSet, sizeof (XFontSet),
    XtOffset (ApplicationDataPtr, fontset), XtRString,
     "*-18-*"}}
XtGetApplicationResources(toplevel, &data, my_resources,
       XtNumber(my_resources), NULL, 0);
fontset = data.fontset;
#endif /* NO_XT */
#endif /* NO_XT */
...
#ifdef NO_XT
fontset = XCreateFontSet (dpy, "*-18-*", &missing_charsets,
    &num_missing_charsets. &default_string);
if (num_missing_charsets > 0) {
    (void) fprintf(stderr, "&s: missing charsets.\n",
                    program_name);
    XFreeStringList(missing_charsets);
}
#else
```

Fonts

Internationalized programs should avoid using fonts directly, that is, `XFontStruct`, unless they are being used for a specific charset and a specific character set. Use of `XFontStruct` may be limiting if the server you are connecting to does not support the specific charsets needed by a locale. The resource name for an `XFontStruct` is `*font`.

Font Lists

All programs using widgets or `XmString` to draw localized text are required to specify an `XFontList` name for specifying fonts. A *font list* is a list of one or more fontsets or fonts, or both. It is used to convey the list of fonts and fontsets a widget should use to render text. For more complicated applications, a font list may specify multiple font sets with each font set being tagged with a name;

for example, Bold, Large, Small, and so on. The tags are to be associated with a tag of an XmString segment. A tag may be used to identify a specific font or fontset within a font list.

Font Set and Font List Syntax

Table 2-1 shows the syntax for a font set and font list.

Table 2-1 Font Set and Font List Syntax

Resource Type	XLFD Separator	Terminator	FontEntry Separator
*fontSet:(Xlib)	comma	None	None
*fontList:(Motif)	semicolon	colon	comma

Here are some examples of font resource specifications:

```
app_foo*fontList: -adobe-courier-medium-r-normal--24-240-75-75-m-\
150-*:
```

The preceding fontList specifies a fontset, consisting of one or more 24-point Adobe Courier fonts, as appropriate for the user's locale.

```
app_foo*fontList: -adobe-courier-medium-r-normal--18-*; *-gothic-\
*-18-*:
```

This fontList specifies a fontset consisting of an 18-point Courier font (if available) for some characters in the users data, and an 18-point Gothic font for the others.

Motif-based applications sometimes need direct access to the font set contained in a font list. For example, an application that uses a DrawingArea widget may want to label one of the images drawn there. The following sample code shows how to extract a font set from a font list. In this example, the tag XmFONTLIST_DEFAULT_TAG looks for the font set because this is the tag that says "codeset of the locale." Applications should use the tag XmFONTLIST_DEFAULT_TAG for any string that could contain localized data.

```
XFontSet FontList2FontSet( XmFontList fontlist)
{
XmFontContext context;
XmFontListEntry next_entry;
XmFontType type_return = XmFONT_IS_FONT;
char* font_tag;
```

```
XFontSet fontset;
XFontSet first_fontset;
Boolean have_font_set = False;

if ( !XmFontListInitFontContext(&context, fontlist)) {
     XtWarning("fl2fs: can't create fontlist context...");
     exit 0;
}

while ((next_entry = XmFontListNextEntry(context)) != NULL) {

    fontset = (XFontSet) XmFontListEntryGetFont(next_entry,
                &type_return);
    if (type_return == XmFONT_IS_FONTSET ) {

        font_tag = XmFontListEntryGetTag(next_entry);

        if (!strcmp(XmFONTLIST_DEFAULT_TAG, font_tag) {
            return fontset;
       }
        /* Remember the 1st fontset, just in case... */
        if (!have_font_set) {
            first_fontset = fontset;
            have_font_set = True;
       }
    }
}
if (have_font_set)
    return first_fontset;
return (XFontSet)NULL;
}
```

Font Functions

The following Xlib font management API functions are available:

- `XCreateFontSet()`
- `XLocaleOfFontSet()`
- `XFontsOfFontSet()`
- `XBaseFontNameListOfFontSet()`
- `XFreeFontSet()`

The following Motif FontListAPI functions are available:

- `XmFontListEntryCreate()`

- `XmFontListEntryAppend()`
- `XmFontListEntryFree()`
- `XmFontListEntryGetTag()`
- `XmFontListEntryGetFont()`
- `XmFontListEntryLoad()`

Font Charsets

To improve basic interchange, fonts are organized according to the standard X-Consortium font charsets.

Default Font Set Per Language Group

Selecting base font names of a font set associated with a developer's language is usually easy because the developer is familiar with the language and the set of fonts needed.

Yet, when selecting the base font names of a font set for various locales, this task can be difficult because an XLFD font specification consists of 15 fields. For localized usage, the following fields are critical for selecting font sets:

- FAMILY_NAME %*F*
- WEIGHT_NAME %*W*
- SLANT %*S*
- ADD_STYLE %*A*
- SPACING %*SP*

This simplifies the number of fields, yet the possible values for each of these fields may vary per locale. The actual point size (POINT_SIZE) may vary across platforms.

Throughout this documentation, the following convention should be used when specifying localized fonts:

```
-dt-%F-%W-%S-normal-%A-*-*-*-%SP-*
```

The following describes the *minimum* set of recommended values for each field to be used within the desktop for the critical fields when specifying font sets in resource (`app-defaults`) files.

Latin ISO8859-1 Fonts

FOUNDRY	'dt'
FAMILY_NAME	'interface user' 'interface system' 'application'
WEIGHT_NAME	medium or bold
SLANT	r or i
ADD_STYLE	sans or serif
SPACING	p or m

Other ISO8859 Fonts

The same values defined for ISO8859-1 are recommended.

JIS Japanese Font

FOUNDRY	'dt'
FAMILY_NAME	Gothic or Mincho
WEIGHT_NAME	medium or bold
SLANT	r
ADD_STYLE	*
SPACING	m

KSC Korean Font

FOUNDRY	'dt'
FAMILY_NAME	Totum or Pathang
WEIGHT_NAME	medium or bold
SLANT	r
ADD_STYLE	*
SPACING	m

Note – The FAMILY_NAME values may change depending on the official romanization of the two common font families in use. As background, Totum corresponds to fonts typically shipped as Gothic, Kodig, or Dotum; Pathang corresponds to fonts typically shipped as Myungo or Myeongjo.

CNS Traditional Chinese Font

FOUNDRY	'dt'
FAMILY_NAME	Sung and Kai
WEIGHT_NAME	medium or bold
SLANT	r
ADD_STYLE	*
SPACING	m

GB Simplified Chinese Font

FOUNDRY	'dt'
FAMILY_NAME	Song and Kai
WEIGHT_NAME	medium or bold
SLANT	r
ADD_STYLE	*
SPACING	m

Drawing Localized Text

There are several mechanisms provided to render a localized string, depending on the Motif or Xlib library being used. The following discusses the interfaces that are recommended for internationalized applications. Yet, it is recommended that *all* localized data be externalized from the program using the simple text.

Simple Text

The following Xlib multibyte (`char*`) drawing functions are available for internationalization:

- `XmbDrawImageString()`
- `XmbDrawString()`
- `XmbDrawText()`

The following Xlib wide character (`wchar_t*`) drawing functions are available for internationalization:

- `XwcDrawImageString()`
- `XwcDrawString()`
- `XwcDrawText()`

The following Xlib multibyte (`char*`) font metric functions are available for internationalization:

- `XExtentsOfFontSet()`
- `XmbTextEscapement()`
- `XmbTextExtents()`
- `XmbTextPerCharExtents`

The following Xlib wide character (`char_t*`) font metric functions are available for internationalization:

- `XExtentsOfFontSet()`
- `XwcTextEscapement()`
- `XwcTextExtents()`
- `XwcTextPerCharExtents`

XmString (Compound String)

For the Xm library, localized text should be inserted into XmString segments using `XmStringCreateLocalized()`. The tag associated with localized text is `XmFONTLIST_DEFAULT_TAG`, which is used to match an entry in a font list. Applications that mix several fonts within a compound string using `XmStringCreate()` should use `XmFONTLIST_DEFAULT_TAG` as the tag for any localized string.

More importantly, for interclient communications, the XmStringConvertToCT() function associates a segment tagged as XmFONTLIST_DEFAULT_TAG as being encoded in the code set of the locale. Otherwise, depending on the tag name used, the Xm library may not be able to properly identify the encoding on interclient communications for text data.

A localized string segment inside an XmString can be drawn with a font list having a font set with XmFONTLIST_DEFAULT_TAG. Use of a localized string is recommended for portability.

The following is an example of creating a font list for drawing a localized string:

```
XmFontList CreateFontList(Display* dpy, char* pattern)
{

XmFontListEntry font_entry;
XmFontList fontlist;
font_entry = XmFontListEntryLoad(dpy, pattern,
                                      XmFONT_IS_FONTSET,
                                      XmFONTLIST_DEFAULT_TAG);

fontlist = XmFontListAppendEntry(NULL, font_entry);
/* XmFontListEntryFree(font_entry); */

if ( fontlist == NULL ) {
      XtWarning("fl2fs: can't create fontlist...");
      exit (0);
}

return fontlist;
}

int main(argc,argv)
int argc;
char **argv;
{
  Display *dpy;            /* Display                 */
  XtAppContext app_context;/*  Application Context    */

  XmFontList fontlist;
  XmFontSet fontset;
  XFontStruct** fontstructs;
  char** fontnames;
  int i,n;
```

```
char *progrname;     /* program name without the full pathname */

if (progname=strrchr(argv[0], '/')){
  progname++;
}
else  {
  progname = argv[0];
}

/*    Initialize toolkit and open display.
 */
XtSetLanguageProc(NULL, NULL, NULL);
XtToolkitInitialize():
app_context = XtCreateApplicationContext();
dpy = XtOpenDisplay(app_context, NULL, progname, "XMdemos",
                      NULL, 0, &argc, argv);
if (!dpy) {
    XtWarning("fl2fs: can't open display, exiting...");
    exit(0);
}

fontlist = CreateFontList(dpy, argv[1]);
fontset = FontList2FontSet(fontlist);

/*
 * Print out BaseFontNames of Fontset
 */
n = XFontsOfFontSet(fontset, &fontstructs, &fontnames);

  printf("Fonts for %s is %d\n", argv[1], n);

  for (i = 0 ; i < n ; ++i ) printf("font[%d} - %s\n", i,\
                                          fontnames[i]);

  exit(1);

}
```

A localized string can be written in resource files because a compound string specified in resource files has a locale-encoded segment with `Xm_FONTLIST_DEFAULT_TAG`. For example, the fontList resource in the following example is automatically associated with `XmFONTLIST_DEFAULT_TAG`.

```
labelString:Japanese string
```

```
*fontList:-dt-interface system-medium-r-normal-L*-*-*-*-*-*-*:
```

The following set of `XmString` functions is recommend for internationalization:

- `XmStringCreateLocalized()`
- `XmStringDraw()`
- `XmStringDrawImage()`
- `XmStringDrawUnderline()`

The following set of `XmString` functions is *not* recommend for internationalization because it takes a direction that may not work with languages not covered:

- `XmStringCreateLtoR()`
- `XmStringSegementCreate()`

Inputting Localized Text

Input for localized text is typically done by using either the local input method or the network-based input method.

The *local input method* means that the input method is built in the Xlib. It is typically used for a language that can be composed using simple rules and that does not require language-specific features. The *network-based input method* means that the actual input method is provided as separate servers, and Xlib communicates with them through the XIM protocol to do the language-specific composition.

Basic Prompts and Dialogs

It is strongly recommended that applications use the `Text` widget to do all text input.

Input within a DrawingArea Widget

Many applications do their own drawing within a widget based on input. To provide consistency within the desktop environment, `XmIm` functions are recommended because the style and geometry management needed for an input method is managed by the `VendorShell` widget class. The application need only worry about handling key events, focus, and communicating the current input location within the drawing area. Using these functions requires some basic knowledge of the underlying Xlib input method architecture, but a developer need only be concerned with the `XmIm` pieces of information.

Application-Specific and Language-Specific Intermediate Feedbacks

Some applications may need to directly display intermediate feedback during preediting, such as when an application exceeds the functions supplied by Xlib. Examples of this include for PostScript™ rendering or using vertical writing.

The core Xlib provides the common set of interfaces that allow an application to display intermediate feedback during preediting. By registering the application's callbacks and setting the preediting style to `XNpreeditCallbacks`, an application can get the intermediate preediting data from the input method and can draw whatever it needs.

Applications intended to do sophisticated language processing may recognize extensions within a specific XIM implementation and its input method engines. Such applications are on the leading edge and will require familiarity with details of the XIM functions.

Text and TextField Widget

For basic prompts and dialogs, the `Text` or `TextField` widget is recommended. Besides resources, all of the `XmTextField` and `XmText` functions are available for getting and for setting localized text inside a `Text[Field]` widget.

Most `XmText` functions are based on the number of characters, *not* on the number of bytes. For example, all `XmTextPosition()` function positions are character positions, not byte positions. The `XmTextGetMaxLength()` function returns the number of bytes. When in doubt, remember that positions are always in character units.

The width of a `Text` or `TextField` widget is determined by the resource value of `XmNcolumns`. But, this value means the number of the widest characters in the font set, not the number of bytes or columns. For example, suppose that you have selected a variable-width font for the `Text` widget. The character *i* may have a width of 1 pixel, while the character *W* may have a width of 7 pixels. When a value of 10 is set for `XmNcolumns`, this is considered a request to make the `Text` widget wide enough to be able to display at least 10 characters. So the `Text` widget must use the width of the widest character to determine the pixel width of its core widget. With this example, it may be able to display 10 *W* characters in the widget, or 70 *i* characters. This structure for `XmNcolumns` may cause problems in locales whose code set is a multibyte and a multicolumn encoding. As such, this value should be set within a localized resource.

The following section identifies the set of functions available for applications that are used to manage input methods. For applications that use the `Text` and `TextField` widgets, refer to "Input Method (Keyboards)" on page 49.

Character Input within Customized Widgets Not Using Text(Field) Widgets

In some cases, an application may obtain character input from the user but does not use a `TextField` or `Text` widget to do so. For example, an application using a `DrawingArea` widget may allow the user to type in text directly into the DrawingArea. In this case, the application could use the Xlib XIM functions as described in later sections, or alternatively, the application may use the `XmIm` functions of Motif 1.2. The `XmIm` functions allow an application to connect to and interact with an input method with a minimum of code. Further, it allows the Motif `VendorShell` widget to take care of geometry management for the input method on the application's behalf.

Although the `XmIm` functions are shipped in all implementations of Motif 1.2, the functions are not documented in Motif 1.2. OSF has announced its intention to augment and document the `XmIm` functions for Motif 2.0. The functions described here are the Motif 1.2 `XmIm` functions.

Note – The Motif 1.2 `XmIm` functions do not support preedit callback style or status callback style input methods. The preedit callback can be used by the Xlib API. For more information, see "XIM Management" on page 38.

Following are the `XmIm` functions you can safely use in a Motif 1.2-based application. The formal description of the parameters and types can be found in the `Xm.h` header file.

Function Name	Description
`XmImRegister()`	Performs `XOpenIM()` and queries the input method for supported styles.

`XmImSetValues()`	Negotiates and selects the preedit and status styles.
`XmImSetFocusValues()`	Creates the XIC, if one does not exist. Notifies the input method that the widget has gained the focus. Sets the values passed to the XIC.
`XmImUnsetFocus()`	Notifies the input method that the widget has lost the focus.
`XmImMbLookupString()`	Xm equivalent of `XmbLookupString()`; converts one or more key events into a character. Return value is identical to `XmbLookupString()`.
`XmImUnregister()`	Disconnects the input method and the widget, allowing connection to a new input method. Does not necessarily close the input method (implementation-dependent).

The `XmImSetValues()` and `XmImSetFocusValues()` functions allow the application to pass information needed by the input method. It is important for the application to pass all values even though not all values are needed (for each supports preedit and status style). This is because the application can never be sure which style has been selected by the user or the `VendorShell` widget. Following are the arguments and data types of each value that should be passed in each call to the `XmImSet[Focus]Values()` function.

Argument Name	Data Type
`XmNbackground`	Pixel
`XmNforeground`	Pixel
`XmNbackgroundPixmap`	Pixmap
`XmNspotLocation`	XPoint
`XmNfontList`	Motif fontlist
`XmNlineSpace`	int (pixel height between consecutive baselines)

The `XmIm` functions are used in the following manner:

- Before initializing the toolkit, the application should call `XtSetLanguageProc(NULL, NULL, NULL)` to initialize the locale.

- After creating the widget where character input is desired, the application should call `XmImRegister(`*widget*`)` to open the input method and establish a connection.
- After establishing a connection to the input method, the application should pass the initial XIC values to the input method by calling `XmImSetValues()` and passing all of the values listed above. This function takes an `arg_list` and a `number_args` argument. The arg_list is loaded by calling `XtSetArg()`.
- Add an event handler, through the `XtAddEventHandler()` function, for the manager widget of the widget obtaining input from the input method. The event handler is for the FocusChangeMask mask. The handler should call `XmImSetFocusValues()` when gaining focus and should call `XmImUnsetFocus()` when losing focus. When setting focus for the input method, pass the full set of values listed above.
- Add a DestroyCallback for the widget obtaining input from the input method. In the destroy callback, call `XmImUnregister()` to notify the input method that you are breaking the connection between the widget and the input method.
- Use `XmImSetValues()` to notify the input method any time one or more of the input method values listed above change (for example, spotLocation).

XIM Management

Following are the XIM management functions:

Function Name	Description
`XOpenIM()`	Establishes a connection to an input method.
`XCloseIM()`	Removes a connection to an input method previously established with a call to `XOpenIM()`.
`XGetIMValues()`	Queries the input method for a list of properties. Currently, the only standard argument in Xlib is `XNQueryInputStyle`.
`XDisplayOfIM()`	Returns the display associated with an input method.
`XLocaleOfIM()`	Returns a string identifying the locale of the input method. There are no standard strings; the value returned by this call is implementation-defined.

`XCreateIC()`	Creates an input context. The input context contains both the data required (if any) by an input method and the information required to display that data.
`XDestroyIC()`	Destroys an input context, freeing any associated memory.
`XIMOfIC()`	Returns the input method currently associated with a given input context.
`XSetICValues()`	Passes zero or more values to an input context to control input of character data, or control display of preedit or status information. A table of all valid input context value arguments can be found in the X11R5 specification.
`XGetICValues()`	Queries an input context to get zero or more input context values. A table of all valid input context value arguments can be found in the X11R5 specification.

XIM Event Handling

Following are the XIM event handling functions:

Function Name	Description
`XmbLookupString()`	Converts keypress events into multibyte characters.
`XwcLookupString()`	Converts keypress events into wide characters.
`XmbResetIC()`	Resets an input context to its initial state. Any input pending on that context is deleted. Returns the current preedit value as a `char*` string. Depending on the implementation of the input method, the return value may be NULL.
`XwcResetIC()`	Resets an input context to its initial state. Any input pending on that context is deleted. Returns the current preedit value as a `wchar_t*` string.
`XFilterEvent()`	Allows the input method to process any incoming events to the clients before the application processes them.
`XSetICFocus()`	Notifies the input method that the focus window attached to the specified input context has received keyboard focus.

`XUnsetICFocus()`	Notifies the input method that the specified input context has lost the keyboard focus and that no more input is expected on the focus window attached to that context.

XIM Callback

X Input Methods (XIMs) provide three categories of callbacks. One is *preedit* callbacks, which allow applications to display the intermediate feedbacks during preediting. The second is *geometry* callbacks, which allow applications and XIM to negotiate the geometry to be used for XIM. The third is *status* callbacks, which allow applications to display the internal status of XIM.

Table 2-2 XIM Callbacks

XIM preedit Callbacks	XIM Status Callbacks	XIM preedit Caret Callbacks	XIM Geometry Callbacks
`(*preeditStart-Callback)()`	`(*StatusStartCall-back)()`	`(*preeditCaret-Callback)()`	`(*GeometryCall-back)()`
`(*preeditDoneCall-back)()`	`(*StatusDoneCall-back)()`		
`(*preeditDrawCall-back)()`	`(*StatusDrawCall-back)()`		

Extracting Localized Text

Although there are different methods to localize an application, the general rule is that any language-dependent information is outside the application and is stored in separate directories identified by a locale name.

This section describes how the user, the application developer, and the implementation combine to establish the language environment of the application. Two general approaches to localizing applications are also discussed. The following three methods can be used:

- Resource files
- Message catalogs
- Private files

Resource Files

This is the GUI toolkit mechanism for customizing all sorts of information about an application. The Intrinsic library (libXt) provides a sophisticated mechanism for merging the command-line options, application-defined resources, and user-defined resources. Resource files can be used for extracting localized text. The difference between resource files and message catalogs is that the resource database is compiled each time it is loaded. As such, care should be taken when deciding which strings to place in resource files and which to place in message catalogs.

Also note that the Xm library functions do not depend on the `LC_MESSAGE` category when specifying the location from which localized resources are loaded. Refer to the `XtSetLanguageProc()` man page for more information.

Message Catalogs

This is the traditional operating system mechanism for accessing external databases containing localized text. These functions load a precompiled catalog file that is ready to be accessed. They also provide defaults within the actual program for cases when no catalogs may be found.

The messaging support is based on both the XPG4 and System V Release 4 (SVR4) interfaces for accessing message catalogs.

Private Files

Private databases can be used by applications to provide generic, customized databases for more than just localization text. Usually, such databases do contain text. It is recommended that if the database is to be spread out over many files, some run-time indirect access of localized text be provided. Without this access, localization for the average user is a difficult effort. Generally, such private file formats are discouraged by groups doing localization. But problems are reduced if a tool is provided specifically for localization of text only.

Message Guidelines

Message guidelines foster consistent formatting of message and help information. They also promote creation and maintenance of messages that can be easily understood by inexperienced English-speaking end users, as well as by inexperienced translators. Use these guidelines to create message files that are consistent in language and clear in meaning. Distribution of these guidelines enable programmers and writers to coordinate their message-writing efforts. Default messages, external message files, and planned delivery of translatable messages are required for each executable to fully implement international language support.

Message Extraction Functions

One of the requirements of internationalizing programs (basic commands and utilities inclusive) is that the messages displayed on the output devices be in the language of the user. As these programs may be used in many countries (international locales), the messages must be translated into the various languages of these countries.

There are two sets of message extraction functions in the desktop environment: XPG4 functions and Xlib functions.

XPG4/Universal UNIX Messaging Functions

The XPG4 message facility consists of several components: message source files, catalog generation facilities, and programming interfaces. Following are the XPG4/Universal UNIX™ message functions:

- `catopen()`
- `catgets()`
- `catclose()`

XPG4 Messaging Examples

There are three parts to this example which demonstrates how to retrieve a message from a catalog. The first part shows the message source file and the second part shows the method used to generate the catalog file. The third part shows an example program using this catalog.

Message Source File

The message catalog can be specified as follows:

```
example.msg file:
$quote "
$ every message catalog should have a beginning set number.
```

```
$set 1 This is the set 1 of messages
1 "Hello world\n"
2 "Good Morning\n"
3 "example: 1000.220 Read permission is denied for the file
%s.\n"
$set 2
1 "Howdy\n"
```

Generation of Catalog File

This file is input to the `gencat` utility to generate the message catalog `example.cat` as follows:

```
gencat example example.msg
```

Accessing the Catalog in a Program

```
#include <locale.h>
#include <nl_types.h>
char *MF_EXAMPLE = "example.cat"

main()
{
        nl_catd catd;
        int error;

        (void)setlocale(LC_ALL, "");

        catd = catopen(MF_EXAMPLE, 0);
            /* Get the message number 1 from the first set.*/

        printf( catgets(catd,1,1,"Hello world\n") );
            /* Get the message number 1 from the second set.*/

        printf( catgets(catd, 2, 1,"Howdy\n") );
            /* Display an error message.*/

        printf( catgets(catd, 1, 4,"example: 100.220
            Permission is denied to read the file %s.\n") ,
            MF_EXAMPLE);
        catclose(catd);
}
```

Xlib Messaging Functions

The following Xlib messaging functions provide a similar input/output (I/O) operation to the resources.

- `XrmPutFileDatabase()`
- `XrmGetFileDatabase()`
- `XrmGetStringDatabase()`
- `XrmLocaleOfDatabase()`

They are described in *X Window System, The Complete Reference to Xlib, Xprotocol, ICCCM, XLFD - X Version 11, Release 5*.

Xlib Message and Resource Facilities

Part of internationalizing a system environment, toolkit-based application is not having any locale-specific data hardcoded within the application source. One common locale-specific item is messages (error and warning) returned by the application of the standard I/O.

In general, for any error or warning messages to be displayed to the user through a system environment toolkit widget or gadget, externalize the messages through message catalogs.

For dialog messages to be displayed through a toolkit component, externalize the messages through localized resource files. This is done in the same way as localizing resources, such as the `XmLabel` and `XmPushButton` classes' `XmNlabelString` resource or window titles.

For example, if a warning message is to be displayed through an `XmMessageBox` widget class, the `XmNmessageString` resource cannot be hardcoded within the application source code. Instead, the value of this resource must be retrieved from a message catalog. For an internationalized application expected to run in different locales, a distinct localized catalog must exist for each of the locales to be supported. In this way, the application need not be rebuilt.

Localized resource files can be put in the `/usr/dt/app-defaults/%L/appdefaults` subdirectories, or they can be pointed to by the `XENVIRONMENT` environment variable. The *%L* variable is replaced with the name of the locale used at run time.

Localized Resources

This section describes which widget and gadget resources are locale-sensitive. The information is organized by related functionality. For example, the first section describes those resources that are locale-sensitive for widgets used to display labels or to provide push-button functionality.

Labels and Buttons

Table 2-3 lists the localized resources that are used as labels. Many of them are of type `XmString`. The rest are of type `color` or `char*`. See the *Motif 1.2 Reference Manual* for detailed descriptions of these resources. In each case, the application should not hardcode these resources. If resource values need to be specified by the application, it should be done with the `app-defaults` file, ensuring that the resource can be localized.

Only the widget class resources are listed here; subclasses of these widgets are not listed. For example, the `XmDrawnButton` widget class does not introduce any new resources that are localized. However, it is a subclass of the `XmLabelWidget` widget class; therefore, its accelerator resource, `acceleratorText` resource, and so on, are also localized and should not be hardcoded by an application.

Table 2-3 Localized Resources

Widget Class	Resource Name
Core	*background:[1]
XmCommand	*command:
XmCommand	*promptString:
XmFileSelectionBox	*dirListLabelString:
XmFileSelectionBox	*fileListLabelString:
XmFileSelectionBox	*filterLabelString:
XmFileSelectionBox	*noMatchString:
XmLabel[Gadget]	*accelerator:
XmLabel[Gadget]	*acceleratorText:
XmLabel[Gadget]	*labelString:
XmLabel[Gadget]	*mnemonic:

Table 2-3 Localized Resources *(Continued)*

Widget Class	Resource Name
XmList	*stringDirection:
XmManager	*stringDirection:
XmMessageBox	*cancelLabelString:
XmMessageBox	*helpLabelString:
XmMessageBox	*messageString:
XmMessageBox	*okLabelString:
XmPrimitive	*foreground:[1]
XmRowColumn	*labelString:
XmRowColumn	*menuAccelerator:
XmRowColumn	*mnemonic:
XmRowColumn(SimpleMenu*)	*buttonAccelerators:
XmRowColumn	*mnemonic:
XmRowColumn	*mnemonic:
XmRowColumn	*mnemonic:
XmRowColumn	*mnemonic:
XmSelectionBox	*applyLabelString:
XmSelectionBox	*cancelLabelString:
XmSelectionBox	*helpLabelString:
XmSelectionBox	*listLabelString:
XmSelectionBox	*okLabelString:
XmSelectionBox	*selectionLabelString:
XmSelectionBox	*textAccelerators:

1. The foreground and background colors are not localized due to restrictions in the X protocol that require color names to be limited to the portable character set. Localized color names are left to applications to provide a localized database to map to a name encoded with the portable character set.

Note that the `XmRowColumn` widget has additional string resources that may be localized. These resources are listed in the `XmRowColumn` man page, under the heading "Simple Menu Creation Resource Set." As the title implies, these resources affect only `RowColumn` widgets created with the

XmCreateSimpleMenu() function. The resources affected are: *buttonAccelerators, *buttonAcceleratorText, *buttonMnemonics, *optionLabel, and *optionMnemonic. These resources are not included in Table 2-3 because they are rarely used and apply to RowColumn only when creating a simple menu.

List Resources

Several widgets allow applications to set or read lists of items in the widget. Table 2-4 shows which widgets allow this and the resources they use to set or read these lists. Because the list items may need to be localized, do not hardcode these lists. Rather, they should be set as resources in app-defaults files, allowing them to be localized. The type for each list is XmStringList.

Table 2-4 Resources Used for Reading Lists

Widget Class	Resource Name
XmList	*items:
XmList	*selectedItems:
XmSelectionBox	*listItems:

Title

Table 2-5 lists the resources used for setting titles and icon names. Normally, an application need only set the *title: and *iconName: resources. The encoding of each is automatically detected for clients doing proper locale management. All of these are of type char or XmString.

Table 2-5 Resources Used for Setting Titles and Icon Names

Widget Class	Resource Name
TopLevelShell	*iconName:
TopLevelShell	*iconNameEncoding:[1]
WmShell	*title:
WmShell	*titleEncoding: [1]
XmBulletinBoard	*dialogTitle:
XmScale	*titleString:

1. This resource should not be set by the application. If the application calls XtSetLanguageProc, the default value (None) of this resource will automatically be set, ensuring that localized text can be used for the title.

Text Widget

Table 2-6 lists the `Text[Field]` resources that are locale-sensitive or about which the developer of an internationalized application should know.

Table 2-6 Locale-Sensitive Text[Field] Resources

Widget Class	Resource Name
XmSelectionBox	*textColumns:[1]
XmSelectionBox	*textString:
XmText	*columns:[1]
XmText	*modifyVerifyCallback:
XmText	*modifyVerifyCallbackWcs:
XmText	*value:
XmText	*valueWcs:
XmTextField	*columns:[1]
XmTextField	*modifyVerifyCallback:
XmTextField	*modifyVerifyCallbackWcs:
XmTextField	*value:
XmTextField	*valueWcs:

1. The *columns resource specifies the initial width of the `Text[Field]` widget in terms of the number of characters to be displayed. In the case of a variable width font or in a locale where the size of a character varies significantly, a column is the amount of space required to display the widest character in that locale's character repertoire. For example, a column width of 10 guarantees that at least 10 characters of the current locale can be displayed; it is possible (likely) that more than that number of characters can be displayed in the allocated space.

Input Method (Keyboards)

Table 2-7 lists localized resources for customizing the input method. These resources allow the user or the application to control which input method will be used for the specified locale and which preedit style (if applicable and available) will be used.

Table 2-7 Localized Resources for Input Method Customization

Widget Class	**Resource Name**
`VendorShell`	`*inputMethod:`
`VendorShell`	`*preeditType:`

Pixmap (Icon) Resources

Table 2-8 lists pixmap resources. In some cases, a different pixmap may be needed for a given locale.

Table 2-8 Pixmap Resources

Widget Class	**Resource Name**
`Core`	`*backgroundPixmap:`
`WMShell`	`*iconPixmap:`
`XmDragIcon`	`*pixmap:`
`XmDropSite`	`*animation[Mask|Pixmap]:`
`XmLabel[Gadget]`	`*labelInsensitivePixmap:`
`XmLabel[Gadget]`	`*labelPixmap:`
`XmMessageBox`	`*symbolPixmap:`
`XmPushButton[Gadget]`	`*armPixmap:`
`XmToggleButton[Gadget]`	`*selectInsensitivePixmap:`
`XmToggleButton[Gadget]`	`*selectPixmap:`

A *pixmap* is a screen image that is stored in memory so that it can be recalled and displayed when needed. The desktop 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 must be localized.

The desktop maintains caches of pixmaps and images. The `XmGetPixmapByDepth()` function 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, the `XmGetPixmapByDepth()` function searches for an X bitmap file whose name matches the requested image name. The `XmGetPixmapByDepth()` function calls the `XtResolvePathname()` function to search for the file. If the requested image name is an absolute path name, that path name is the search path for the `XtResolvePathname()` function. Otherwise, the `XmGetPixmapByDepth()` function constructs a search path in the following way:

- If the `XBMLANGPATH` environment variable is set, the value of that variable is the search path.
- If `XBMLANGPATH` is not set but `XAPPLRESDIR` is set, the `XmGetPixmapByDepth()` function uses a default search path with entries that include `$XAPPLRESDIR`, the user's home directory, and vendor-dependent system directories.
- If neither `XBMLANGPATH` nor `XAPPLRESDIR` is set, the `XmGetPixmapByDepth()` function uses a default search path with entries that include the user's home directory and vendor-dependent system directories.

These paths may include the *%B* substitution field. In each call to the `XtResolvePathname()` function, the `XmGetPixmapByDepth()` function substitutes the requested image name for *%B*. The paths may also include other substitution fields accepted by the `XtResolvePathname()` function. In particular, the `XtResolvePathname()` function 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.

Because there is no string-to-pixmap converter supplied by default, pixmaps are generally set by the application at creation time by first retrieving the pixmap with a call to `XmGetPixmap()`. `XmGetPixmap()` uses the current locale to determine where to locate the pixmap. (See the `XmGetPixmap()` man page for a description of how locale is used to locate the pixmap.)

Font Resources

Table 2-9 lists the localized font resources. All XmFontList resources are of type XmFontList. In almost all cases, a fontset should be used when specifying a fontlist element. The only exception is when displaying character data that does not appear in the character set of the user (for example, displaying math symbols or dingbats).

Table 2-9 Localized Font Resources

Widget Class	Resource Name
VendorShell	*buttonFontList:
VendorShell	*defaultFontList:
VendorShell	*labelFontList:
VendorShell	*textFontList:
XmBulletinBoard	*buttonFontList:
XmBulletinBoard	*defaultFontList:
XmBulletinBoard	*labelFontList:
XmBulletinBoard	*textFontList:
XmLabel[Gadget]	*fontList:
XmList	*fontList:
XmMenuShell	*buttonFontList:
XmMenuShell	*defaultFontList:
XmMenuShell	*labelFontList:
XmText	*fontList:
XmTextField	*fontList:

Operating System Internationalized Functions

Table 2-10 lists the base operating system internationalized functions in a common open software environment.

Applications should perform proper locale management with the assumption that a locale may have from 1 to 4 bytes per coded character.

Table 2-10 Base Operating System Internationalized Functions

Locale Management	**Single-byte**	**Multibyte**	**Wide Character**
Convert mb <-> wc		`mbtowc` `mbstowcs`	`wctomb` `wcstombs`
Classification	`isalpha` `is*`		`isalpha` `isw*` `wctype`
Case Mapping	`tolower` `toupper`		`towlower` `towupper`
Format Miscellaneous		`localeconv` `nl_langinfo`	
Format of Numeric		`strtol` `strtod`	`wcstol` `wcstod` `wcstoi`
Format Time/Monetary		`strftime` `strptime` `strfmon`	`wcsftime`
String Copy		`strcat` `strcpy` `strncat` `strncpy`	`wcscat` `wcsncat` `wcscpy` `wcsncpy`
String Collate		`strcoll`	`wcscoll` `wcsxfrm`
String Misc	`strlen`	`mblen`	`wcscmp` `wcsncmp`
String Search	`strchr` `strcspn` `strpbrk` `strrchr` `strspn` `strtok`		`wcschr` `wcscspn` `wcspbrk` `wcsrchr` `wcsspn` `wcstok` `wcswcs` `wcscspn`
I/O Display Width			`wcwidth`[1] `wcswidth`

Table 2-10 Base Operating System Internationalized Functions *(Continued)*

Locale Management	Single-byte	Multibyte	Wide Character
I/O Printf		printf vprintf sprintf vsprint fprintf vfprint	printf vprintf sprintf vsprint frpintf vfprint
I/O Scan		scanf sscanf fscanf	scanf sscanf fscanf
I/O Character	getc gets putc puts		fgetwc fgetws fputwc fputws ungetwc
Message		gettxt catopen catgets catclose	
Convert Codeset		iconv_open iconv iconv_close	

1. These functions are provided for applications using terminals. Graphical user interface (GUI) applications should not use these functions; instead, they should use font metric functions listed on page 31 to determine spacing.

Internationalization and Distributed Networks 3

This chapter discusses tasks related to internationalization and distributed networks.

Interchange Concepts

This section describes the way 8-bit user names and 8-bit data can be communicated on a network for communications utilities, such as ftp, mail, or interclient communication between the desktop clients.

There are three primary considerations for communicating data:

- Sender's code set and the receiver's code set.
- Whether the communications protocol allows 8-bit data or is limited to 7-bit coded data (for example, the Japanese JUNET passes Japanese Industrial Standard (JIS) coded data over 7-bit protocols).
- Type of interchange encoding available, per protocol rules. The actual conversion needed is dependent on the specific protocol used.

If the remote host uses the same code set as the local host, the following is true:

- If the protocol allows 8-bit data, no conversions are needed.
- If the protocol allows only 7-bit data, a method is needed to map the 8-bit code points to 7-bit ASCII values. This could be accomplished using the `iconv` framework and one of the following types of 7-bit encoded methods:
  - Map 8-bit data as specified in the POSIX.2 specification for uuencode and uudecode algorithms.
  - Optionally, the 8-bit data may be mapped to a 7-bit interchange encoding as defined by the protocol; for example, 7-bit ISO2022 in Xlib or base64 in Multipurpose Internet Message Extensions (MIME).

If the remote host's code set is different from that of the local host, the following two cases may apply. The conversion needed is dependent on the specific protocol used.

- If the protocol allows 8-bit data, the protocol will need to specify which side does the `iconv` conversion and to specify the encoding on the wire. In some protocols, an 8-bit interchange encoding is recommended that is capable of encoding all possible code sets and identifying character repertoire.
- If the protocol allows only 7-bit data, a 7-bit interchange encoding is needed, as is the identifying character repertoire.

iconv Interface

In a network environment, the code sets of the communicating systems and the protocols of communication determine the transformation of user-specified data so that it can be sent to the remote system in a meaningful way. The user data (not user names) may need to be transformed from the sender's code set to the receiver's code set, or 8-bit data may need to be transformed into a 7-bit form to conform to protocols. A uniform interface is needed to accomplish this.

In the following examples, using the `iconv` interface is illustrated by explaining how to use `iconv_open()`, `iconv()`, and `iconv_close()`. To do the conversion, `iconv_open()` must be followed by `iconv()`. The terms *7-bit interchange* and *8-bit interchange* are used to refer to any interchange encoding used for 7-bit and 8-bit data, respectively.

Sender and Receiver Use the Same Code Sets:

- If the protocol allows 8-bit data, use 8-bit data because the same code set is being used. No conversion is needed.

- If the protocol allows only 7-bit data, use `iconv`:
  - Sender

    `cd = iconv_open(locale_codeset, `*uuencoded*`);`
  - Receiver

    `cd = iconv_open("uucode", locale_codeset);`

Sender and Receiver Use Different Code Sets:

- If the protocol allows 8-bit data:
  - Sender

    `cd = iconv_open(locale_codeset,`*8-bitinterchange*`);`
  - Receiver

    `cd = iconv_open(`*8-bitinterchange*`, locale_codeset);`
- If the protocol allows only 7-bit data, do the following:
  - Sender

    `cd = iconv_open(locale_codeset, `*7-bitinterchange*`);`
  - Receiver

    `cd = iconv_open(`*7-bitinterchange*`, locale_codeset);`

The `locale_codeset` refers to the code set being used locally by the application. Note that while the `nl_langinfo(CODESET)` function may be used to obtain the code set associated with the current locale, it is implementation-dependent whether any conversion names match the return from the `nl_langinfo(CODESET)` function.

The Table 3-1 outlines how `iconv` can be used to perform conversions for various conditions. Specific protocols may dictate other conversions needed.

Table 3-1 Using iconv to Perform Conversions

	Communication with system using the same code set (for example, XYZ)		**Communication with system using different code sets or receiver's code set is unknown**	
Conversion to Use	**7-bit Protocol**	**8-bit Protocol**	**7-bit Protocol**	**8-bit Protocol**
code XYZ	Invalid	Best Choice	Invalid	Invalid if remote code set is unknown
7-bit Interchange ISO2022	OK	OK	Best Choice	OK
8-bit Interchange ISO2022 ISO 10646	Invalid[1]	OK	Invalid	Best Choice
7-bit Untagged quoted-printable uucode	OK	OK	Requires code set identification	Requires code set identification
8-bit Untagged base64	Invalid	OK	Requires code set identification	Requires code set identification

1. Invalid means the interchange encoding should not be used for the choice of code set and type of protocol.

Stateful and Stateless Conversions

Code sets can be classified into two categories: stateful encodings and stateless encodings.

Stateful Encodings

Stateful encoding uses sequences of control codes, such as shift-in/shift-out, to change character sets associated with specific code values.

For instance, under compound text, the control sequence "ESC$(B" can be used to indicate the start of Japanese 16-bit data in a data stream of characters, and "ESC(B" can be used to indicate the end of this double-byte character data and the start of 8-bit ASCII data. Under this stateful encoding, the bit value 0x43 could not be interpreted without knowing the shift state. The EBCDIC Asian code sets use shift-in/shift-out controls to swap between double- and single-byte encodings, respectively.

Converters that are written to do the conversion of stateful encodings to other code sets tend to be a little complex due to the extra processing needed.

Stateless Encodings

Stateless code sets are those that can be classified as one of two types:

- Single-byte code sets, such as the ISO8859 family.
- Multibyte code sets, such as PC codes for Japanese and Shift-JIS (SJIS).

The term *multibyte code sets* is also used to refer to any code set that needs one or more bytes to encode a character; multibyte code sets are considered stateless.

Note – Conversions are meaningful only if the code sets represent the same character set.

Simple Text Basic Interchange

When a program communicates data to another program residing on a remote host, a need may arise for conversion of data from the code set of the source machine to that of the receiver. For example, this happens when a PC system using PC codes needs to communicate with a workstation using an International Organization for Standardization/Extended UNIX Code (ISO/EUC) encoding. Another example occurs when a program obtains data in one code set but has to display this data in another code set. To support these conversions, a standard program interface is provided based on the XPG4 `iconv()` function definitions.

All components doing code set conversion should use the `iconv` functions as their interface to conversions. Systems are expected to provide a wide variety of conversions, as well as a mechanism to customize the default set of conversions.

iconv Conversion Functions

The common method of conversions from one code set to another is through a table-driven method. In some cases, these tables may be too large, hence an algorithmic method may be more desirable. To accommodate such diverse requirements, a framework is defined in XPG4 for code set conversions. In this framework, to convert from one code set to another, open a converter, perform the conversions, and close the converter. The `iconv` functions are `iconv_open()`, `iconv()`, and `iconv_close()`.

Code set converters are brought under the framework of the `iconv_open()`, `iconv()`, and `iconv_close()` set of functions. With these functions, it is possible to provide and to use several different types of converters. Applications can call these functions to convert characters in one code set into characters in another code set. With the advent of the `iconv` framework, converters can be provided in a uniform manner. The access and use of these converters is being standardized under X/Open XPG4.

X Interclient (ICCCM) Conversion Functions

Xlib provides the following functions for doing conversions.

X ICCCM Multibyte Functions	ICCCM Wide Character Functions
`XmbTextPropertyToTextList()`	`XwcTextPropertyToTextList()`
`XmbTextListToTextProperty()`	`XwcTextListToTextProperty()`

Note – The `libXm` library does provide the `XmStringConvertToCT()` and `XmStringConvertFromCT()` functions; however, these are not recommended because there are some hardcoded assumptions about certain XmString tags. For example, if the tag is `bold`, `XmStringConvertToCT()` is implementation-dependent. Across various platforms, the behavior of this function cannot be guaranteed in all international regions.

Refer to "Interclient Communications Conventions for Localized Text" on page 122 for more information.

Window Titles

The standard way for setting titles is to use resources. But for applications that set the titles of their windows directly, a localized title must be sent to the Window Manager. Use the `XCompoundTextStyle` encoding defined in `XICCEncodingStyle`, as well as the following guidelines:

- Compound text can be created either by `XmbTextListToTextProperty()` or `XwcTextListToTextProperty()`.
- Localized titles can be displayed using the `XmNtitle` and `XmNtitleEncoding` resources of the `WMShell` widget. Localized icon names can be displayed using the `XmNiconName` and `XmNiconNameEncoding` resources of the `TopLevelShell` widget.
- Localized titles of dialog boxes can also be displayed using the `XmNdialogTitle` resource of the `XmBulletinBoard` widget.
- Window Manager should have an appropriate fontlist for displaying localized strings.

Following is an example of displaying a localized title and icon name. Compound text is made from the compound string in this example.

```
include         <nl_types.h>
Widget          toplevel;
Arg             al[10];
int             ac;
XTextProperty   title;
char            *localized_string;
nl_catd         fd;

XtSetLanguageProc(NULL, NULL, NULL);
fd = catopen("my_prog", 0);
localized_string = catgets(fd, set_num, mes_num, "defaulttitle");
XmbTextListToTextProperty(XtDisplay(toplevel), &localized_string,
       1, XCompoundTextStyle, &title);
ac = 0;
XtSetArg(al[ac], XmNtitle, title.value); ac++;
XtSetArg(al[ac], XmNtitleEncoding, title.encoding); ac++;
XtSetValues(toplevel, al, ac);
```

If you are using a window rather than widgets, the `XmbSetWMProperties()` function automatically converts a localized string into the proper `XICCEncodingStyle`.

Mail Basic Interchange

In general, electronic mail (email) strategy has been one of turning email into a canonical, labeled format as opposed to optimizing a message given knowledge of the receiver's locale. This means that in the email world, you should always assume that the receiver *may* be in a different locale. In the desktop world, the default email transport is Simple Mail Transfer Protocol (SMTP), which only supports 7-bit transmission channels.

With this understanding, the email strategy for the desktop is as follows:

- The sending agents, by default (unless instructed otherwise by the user), converts a body part into a *standard* format for the sending transmission channel and labels the body part with the character encoding used.
- The receiving agent looks at the body part to see if it can support the character encoding; if it can, it converts it into the local character set.

In addition, because the MIME format is used for messages, any 8-bit to 7-bit transformations are done using the built-in MIME transport encodings (base64 or quoted-printable). See the Request for Comments (RFC) 1521 MIME standard specification.

Encodings and Code Sets

To understand code sets, it is necessary to first understand character sets. A *character set* is a collection of predefined characters based on the specific needs of one or more languages without regard to the encoding values used to represent the characters. The choice of which code set to use depends on the user's data processing requirements. A particular character set can be encoded using different encoding schemes. For example, the ASCII character set defines the set of characters found in the English language. The Japanese Industrial Standard (JIS) character set defines the set of characters used in the Japanese language. Both the English and Japanese character sets can be encoded using different code sets.

The ISO2022 standard defines a coded character set as a group of precise rules that defines a character set and the one-to-one relationship between each character and its bit pattern. A code set defines the bit patterns that the system uses to identify characters.

A code page is similar to a code set with the limitation that a code-page specification is based on a 16-column by 16-row matrix. The intersection of each column and row defines a coded character.

Code Set Strategy

The common open software environment code set support is based on International Organization for Standardization (ISO) and industry-standard code sets providing industry-standard code sets that satisfy the data processing needs of users.

Each locale in the system defines which code set it uses and how the characters within the code set are manipulated. Because multiple locales can be installed on the system, multiple code sets can be used by different users on the system. While the system can be configured with locales using different code sets, all system utilities assume that the system is running under a single code set.

Most commands have no knowledge of the underlying code set being used by the locale. The knowledge of code sets is hidden by the code-set-independent library subroutines (Internationalization libraries), which pass information to the code-set-dependent subroutines.

Because many programs rely on ASCII, all code sets include the 7-bit ASCII code set as a proper subset. Because the 7-bit ASCII code set is common to all supported code sets, its characters are sometimes referred to as the *portable* character set.

The 7-bit ASCII code set is based on the ISO646 definition and contains the control characters, punctuation characters, digits (0-9), and the English alphabet in uppercase and lowercase.

Code Set Structure

Each code set is divided into two principle areas:

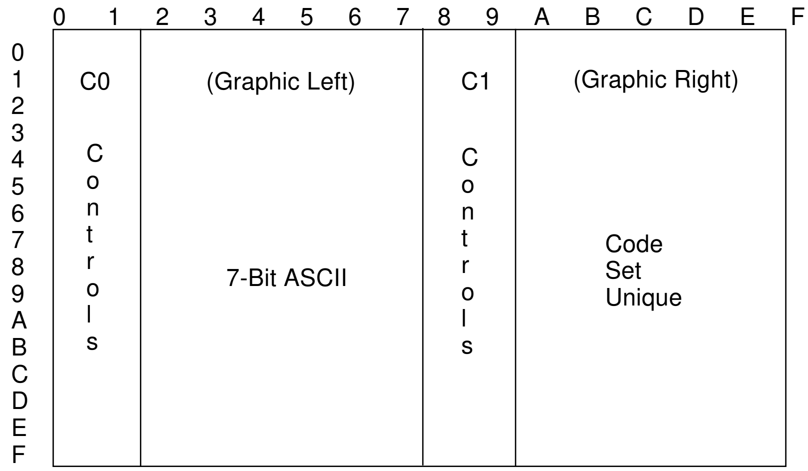
- Graphic Left (GL) Columns 0-7
- Graphic Right (GR) Columns 8-F

The first two columns of each code set are reserved by ISO standards for control characters. The terms C0 and C1 are used to denote the control characters for the Graphic Left and Graphic Right areas, respectively.

Note – The PC code sets use the C1 control area to encode graphic characters.

The remaining six columns are used to encode graphic characters (see Table 3-2 on page 65). Graphic characters are considered to be printable characters, while the control characters are used by devices and applications to indicate some special function.

Table 3-2 Code Set Overview



Control Characters

Based on the ISO definition, a control character initiates, modifies, or stops a control operation. A control character is not a graphic character, but can have graphic representation in some instances. The control characters in the ISO646-IRV character set are present in all supported code sets,and the encoded values of the C0 control characters are consistent throughout the code sets.

Graphic Characters

Each code set can be considered to be divided into one or more character sets, such that each character is given a unique coded value. The ISO standard reserves six columns for encoding characters and does not allow graphic characters to be encoded in the control character columns.

Single-Byte Code Sets

Code sets that use all 8 bits of a byte can support European, Middle Eastern, and other alphabetic languages. Such code sets are called single-byte code sets. This provides a limit of encoding 191 characters, not including control characters.

Multibyte Code Sets

The term *multibyte code sets* is used to refer to all possible code sets regardless of the number of bytes needed to encode any specific character. Because the operating system should be capable of supporting any number of bits to encode a character, a multibyte code set may contain characters that are encoded with 8, 16, 32, or more bits. Even single-byte code sets are considered to be multibyte code sets.

Extended UNIX Code (EUC) Code Set

The EUC code set uses control characters to identify characters in some of the character sets. The encoding rules are based on the ISO2022 definition for the encoding of 7-bit and 8-bit data. The EUC code set uses control characters to separate some of the character sets.

The term EUC denotes these general encoding rules. A code set based on EUC conforms to the EUC encoding rules but also identifies the specific character sets associated with the specific instances. For example, eucJP for Japanese refers to the encoding of the JIS characters according to the EUC encoding rules.

The first set (CS0) always contains an ISO646 character set. All of the other sets must have the most-significant bit (MSB) set to 1, and they can use any number of bytes to encode the characters. In addition, all characters within a set must have:

- Same number of bytes to encode all characters.
- Same column display width (number of columns on a fixed-width terminal).

Each character in the third set (CS2) is always preceded with the control character SS2 (single-shift 2, 0x8e). Code sets that conform to EUC do not use the SS2 control character other than to identify the third set.

Each character in the fourth set (CS3) is always preceded with the control character SS3 (single-shift 3, 0x8f). Code sets that conform to EUC do not use the SS3 control character other than to identify the fourth set.

ISO EUC Code Sets

The following code sets are based on definitions set by the International Organization for Standardization (ISO).

- ISO646-IRV
- ISO8859-1
- ISO8859-x
- eucJP
- eucTW
- eucKR

ISO646-IRV

The ISO646-IRV code set defines the code set used for information processing based on a 7-bit encoding. The character set associated with this code set is derived from the ASCII characters.

ISO8859-1

ISO8859-1 encoding is a single-byte encoding that is based on and is compatible with other ISO, American National Standards Institute (ANSI), and European Computer Manufacturer's Association (ECMA) code extension techniques. The ISO8859 encoding defines a family of code sets with each member containing its own unique character sets. The 7-bit ASCII code set is a proper subset of each of the code sets in the ISO8859 family.

The ISO8859-1 code set is called the ISO Latin-1 code set and consists of two character sets:

- ISO646-IRV Graphic Left, 7-bit ASCII character set.
- ISO8859-1 Graphic Right (Latin) character set.

These character sets combined include the characters necessary for Western European languages such as Danish, Dutch, English, Finnish, French, German, Icelandic, Italian, Norwegian, Portuguese, Spanish, and Swedish.

While the ASCII code set defines an order for the English alphabet, the Graphic Right (GR) characters are not ordered according to any specific language. The language-specific ordering is defined by the locale.

Other ISO8859 Code Sets

This section lists the other significant ISO8859 code sets. Each code set includes the ASCII character set plus its own unique characters.

ISO8859-2

Latin alphabet, No. 2, Eastern Europe

- Albanian
- Czechoslovakian
- English
- German
- Hungarian
- Polish
- Rumanian
- Serbo-Croatian
- Slovak
- Slovene

ISO8859-5

Latin/Cyrillic alphabet

- Bulgarian
- Byelorussian
- English
- (FYROM)Macedonian
- Russian
- Ukrainian

ISO8859-6

Latin/Arabic alphabet

- English
- Arabic

ISO8859-7

Latin/Greek alphabet

- English
- Greek

ISO8859-8

Latin/Hebrew alphabet

- English

- Hebrew

ISO8859-9

Latin/Turkish alphabet

- Danish
- Dutch
- English
- Finnish
- French
- German
- Irish
- Italian
- Norwegian
- Portuguese
- Spanish
- Swedish
- Turkish

eucJP

The EUC for Japanese consists of single-byte and multibyte characters (2 and 3 bytes). The encoding conforms to ISO2022 and is based on JIS and EUC definitions .

Table 3-3 Encoding for eucJP

CS	Encoding			Character Set
cs0	0xxxxxxx			ASCII
cs1	1xxxxxxx	1xxxxxxx		JIS X0208-1990
cs2	0x8E	1xxxxxxx		JIS X0201-1976
cs3	0x8F	1xxxxxxx	1xxxxxxx	JIS X0212-1990

JIS X0208-1990

A code of the Japanese graphic character set for information interchange (1990 version) that contains 147 special characters, 10 numeric digits, 83 Hiragana characters, 86 Katakana characters, 52 Latin characters, 48 Greek characters, 66 Cyrillic characters, 32 line-drawing elements, and 6355 Kanji characters.

JIS X0201

A code for information interchange that contains 63 Katakana characters.

JIS X0212-1990

A code of the supplementary Japanese graphic character set for information interchange (1990 version) that contains 21 additional special characters, 21 additional Greek characters, 26 additional Cyrillic characters, 27 additional Latin characters, 171 Latin characters with diacritical marks, and 5801 additional Kanji characters.

eucTW

The EUC for Traditional Chinese is an encoding consisting of characters that contain single-byte and multibyte (2 and 4 bytes) characters. The EUC encoding conforms to ISO2022 and is based on the Chinese National Standard (CNS) as defined by the Republic of China and the EUC definition, see Table 3-4.

Table 3-4 Encoding for eucTW

CS	Encoding			Character Set
cs0	0xxxxxxx			ASCII
cs1	1xxxxxxx	1xxxxxxx		CNS 11643.1992 - plane 1
cs2	0x8EA2	1xxxxxxx	1xxxxxxx	CNS 11643.1992 - plane 2
cs3	0x8EA3	1xxxxxxx	1xxxxxxx	CNS 11643.1992 - plane 3
	0x8EB0	1xxxxxxx	1xxxxxxx	CNS 11643.1992 - Plane 16

CNS 11643-1992 defines 16 planes for the Chinese Standard Interchange Code, each plane can support up to 8836 characters (94x94). Currently, only planes 1 through 7 have characters assigned. Table 3-5 shows the 16 planes of the CNS 11643-1992 standard.

Table 3-5 16 Planes of the CNS 11643-1992 Standard

Plane	Definition	# of Character	EUC Encoding
1	Most frequently used	6085	A1A1-FDCB
2	Secondary frequently	7650	8EA2 A1A1 - 8EA2 F2C4
3	Exec.Yuen EDP [1] center	6148	8EA3 A1A1 - 8EA3 E2C6
4	RIS[2], Vendor defined	7298	8EA4 A1A1 - 8EA4 EEDC
5	Rarely used by MOE[3]	8603	8EA5 A1A1 - 8EA5 FCD1
6	Variation char set 1 by MOE	6388	8EA6 A1A1 - 8EA6 E4FA
7	Variation char set 2 by MOE	6539	8EA7 A1A1 - 8EA7 E6D5
8	Undefined	0	8EA8 A1A1 - 8EA8 FEFE
9	Undefined	0	8EA9 A1A1 - 8EA9 FEFE
10	Undefined	0	8EAA A1A1 - 8EAA FEFE
11	Undefined	0	8EAB A1A1 - 8EAB FEFE
12	User Defined Character (UDC)	0	8EAC A1A1 - 8EAC FEFE
13	UDC	0	8EAD A1A1 - 9EAD FEFE
14	UDC	0	8EAE A1A1 - 8EAE FEFE
15	UDC	0	8EAF A1A1 - 8EAF FEFE
16	UDC	0	8EB0 A1A1 - 8EB0 FEFE

1. EDP: Center of Directorate, General of Budget, Accounting, and Statistics
2. RIS: Residence Information System
3. MOE: Ministry of Education

eucKR

The EUC for Korean is an encoding consisting of single-byte and multibyte characters (shown in Table 3-6). The encoding conforms to ISO2022 and is based on Korean Standard Code (KSC) set and EUC definitions.

Table 3-6 Encoding for eucKR.

CS	Encoding		Character Set
cs0	0xxxxxxx		ASCII
cs1	1xxxxxxx	1xxxxxxx	KS C 5601-1992
cs2			Not used
cs3			Not used

KSC 5601-1992 (code of the Korean character set for information interchange, 1992 version) contains 432 special characters, 30 Arabic and Roman numeral characters, 94 Hangul alphabet characters, 52 Roman characters, 48 Greek characters, 27 Latin characters, 169 Japanese characters, 66 Russian characters, 68 line-drawing elements, 2344 precomposed Hangul characters, and 4888 Hanja characters.

One Hangul character can be comprised of several consonants and vowels. Most Hangul words can be expressed in Hanja words. Hanja is a set of Traditional Chinese characters, which is currently used by Korean people. Each Hanja character has its own meaning and is thus more specific than Hangul most of the time.

Motif Dependencies 4

This chapter discusses tasks related to internationalizing with Motif.

Locale Management

The term *language environment* refers to the set of localized data that the application needs 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, User Interface Definition (UID) files, or message catalogs (on XPG3-compliant systems).

A single language environment is established when an application runs. The language environment in which an application operates is specified by the application user, often either by setting an environment variable (`LANG` or `LC_*` 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 by using the `setlocale()` function in a language procedure established by the `XtSetLanguageProc()` function. This causes Xt to cache a per-display language string that is used by the `XtResolvePathname()` function to find resource, bitmap, and User Interface Language (UIL) files.

An application that supplies a language procedure can either provide its own procedure or use an Xt default procedure. In either case, the application establishes the language procedure by calling the `XtSetLanguageProc()` function before initializing the toolkit and before loading the resource databases (such as by calling the `XtAppInitialize()` function). 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. This is done by using:

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

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

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

The application can use the default language procedure by making the call to the `XtSetLanguageProc()` function in the following manner:

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

By default, Xt does not install any language procedure. If the application does not call the `XtSetLanguageProc()` function, Xt uses as its per-display language string the value of the `xnlLanguage` resource if it is set. If the `xnlLanguage` resource is not set, Xt derives the language string from the `LANG` environment variable.

Note – The per-display language string that results from this process is implementation-dependent, and Xt provides no public means of examining the language string once it is established.

By supplying its own language procedure, an application can use any procedure it wants for setting the language string.

Font Management

The desktop uses font lists to display text. A *font* defines a set of glyphs that represent the characters in a given character set. A *font set* is a group of fonts that are needed to display text for a given locale or language. A *font list* is a list of fonts, font sets, or a combination of the two, that may be used. Motif has convenience functions to create a font list.

Font List Structure

The desktop requires a font list for text display. A font list is a list of font structures, font sets, or both, each of which has a tag to identify it. A font set ensures that all characters in the current language can be displayed. With font structures, the responsibility for ensuring that all characters can be displayed rests with the programmer (including converting from the code set of the locale to glyph indexes).

Each entry in a font list is in the form of a {*tag*, *element*} pair, where *element* can be either a single font or a font set. The application can create a font list entry from either a single font or a font set. For example, the following code segment creates a font list entry for a font set:

```
char font1[] =
       "-adobe-courier-medium-r-normal--10-100-75-75-M-60";
font_list_entry = XmFontListEntryLoad (displayID, font1,
       XmFONT_IS_FONTSET, "font_tag");
```

The `XmFontListEntryLoad()` function loads a font or creates and loads a font set. The following are the four arguments to the function:

displayID — Display on which the font list is to be used.

fontname — A string that represents either a font name or a base font name list, depending on the *nametype* argument.

nametype — A value that specifies whether the *fontname* argument refers to a font name or a base font name list.

tag — A string that represents the tag for this font list entry.

If the *nametype* argument is `XmFONT_IS_FONTSET`, the `XmFontListEntryLoad()` function creates a font set in the current locale from the value in the *fontname* argument. The character sets of the fonts specified in the font set are dependent on the locale. If *nametype* is `XmFONT_IS_FONT`, the `XmFontListEntryLoad()` function opens the font found in *fontname*. In either case, the font or font set is placed into a font list entry.

The following code example creates a new font list and appends the entry `font_list_entry` to it:

```
XmFontList font_list;
XmFontListEntry font_list_entry;
       .
       .
font_list = XmFontListAppendEntry (NULL, font_list_entry);
XmFontListEntryFree (font_list_entry);
```

Once a font list has been created, the `XmFontListAppendEntry()` function adds a new entry to it. The following example uses the `XmFontListEntryCreate()` function to create a new font list entry for an existing font list.

```
XFontSet font2;
char *font_tag;
XmFontListEntry font_list_entry2;
       .
       .
font_list_entry2 = XmFontListEntryCreate (font_tag,
        XmFONT_IS_FONTSET, (XtPointer)font2);
```

The `font2` parameter specifies an `XFontSet` returned by the `XCreateFontSet()` function. The arguments to the `XmFontListEntryCreate()` function are `font_tag`, `XmFONT_IS_FONTSET`, and `font2`, which are the tag, type, and font, respectively. The tag and the font set are the {*tag*, *element*} pair of the font list entry.

To add this entry to the font list, use the `XmFontListAppendEntry()` function again, only this time, its first parameter specifies the existing font list.

```
font_list = XmFontListAppendEntry(font_list, font_list_entry2);
XmFontListEntryFree(font_list_entry2);
```

Font Lists Examples

The syntax for specifying a font list in a resource file depends on whether the list contains fonts, font sets, or both.

Obtaining a Font

To obtain a font, specify a font and an optional font list element tag.

- If the tag is present, it should be preceded by an = (equal sign).
- If the tag is not present, do not use an = (equal sign).

Entries specifying more than one font are separated by a , (comma).

Obtaining a Font Set

To obtain a font set, specify a base font list and an optional font list element tag.

- If the tag is present, it should be preceded by a : (colon) instead of an = (equal sign).
- If the tag is not present, the colon *must* still be present as this is what distinguishes a font from a font set in the resource declaration.

Fonts specified in the base font list are separated by a ; (semicolon). Entries specifying more than one font set are separated by a , (comma).

Specifying a Font When the Font List Element Tag Is Absent

If the font list element tag is not present, the default `XmFONTLIST_DEFAULT_TAG` is used. Here are some examples.

- Specifying a font using the default font list element tag:

```
*fontList: fixed
*fontList: \
 -adobe-courier-medium-r-normal--10-100-75-75-M-60-iso8859-1
```

- Specifying a font list element tag:

```
*fontList: fixed=ROMAN, 8x13bold=BOLD
```

- Specifying two fonts, one with the default font list element tag and one with an explicit tag:

```
*fontList: fixed, 8x13bold=BOLD
```

Specifying a Font Set When the Font List Element Tag Is Absent

If the font list element tag is not present, the default `XmFONTLIST_DEFAULT_TAG` is used. Here are some examples of specifying a font set.

- Let Xlib select the fonts without specifying a font list element tag:

```
*fontList: -dt-application-medium-r-normal-*-m*-*-*-*-m-*
```

- Let Xlib select the fonts and specify a font list element tag as *MY_TAG:*

```
*fontList: -dt-application-medium-r-normal-*-m*-*-*-*-m*:MY_TAG
```

- Let Xlib select the fonts, specify a font list element tag for bold fonts, and use the default font list element tag for the others:

```
*fontList:-dt-application-medium-r-normal-*-m*-*-*-*-m-*:,\
  -dt-application-medium-r-normal-style2-m*-*-*-*-m-*:BOLD
```

Font List Syntax

The `XmFontList()` data type can contain one or more entries that are associated with one of the following elements:

`XFontStruct`	An X font that can be used to draw text encoded in the charset of the font, that is, *font-encoded text*.
`XFontSet`	A collection of `XFontStruct` fonts used to draw text encoded in a locale, that is, *localized text*.

The following syntax is used by the string-to-XmFontList converter:

```
XmFontList  := <fontentry> {’, ’fontentry}

fontentry   := <fontname><fontid>
             | <baselist><fontsetid>

baselist    := <fontname>{’;’<fontname>}

fontsetid   := ’:’<string> | <defaultfontset>

fontname    := <XLFD string>

fontid      := ’=’<string> | <defaultfont>

XLFD string:= refer to XLFD Specification
defaultfont:= NULL
defaultfontset:= ’:’NULL
string      := any character from ISO646IRV, except newline
```

A `fontentry` with a given `XmFontList` can specify either a font or a font set. In either case, the ID (`fontid` or `fontsetid`) can be referenced by a segment within a compound string (`XmString`).

Both `defaultfont` and `defaultfontset` can define the default `fontentry`, yet there can only be one default per `XmFontList`.

The `XmFONTLIST_DEFAULT_TAG` identifier always references the default `fontentry` when `XmString` is drawn. If the default `fontentry` is not specified, the first `fontentry` is used to draw.

The resource converter operates under a single locale so that all font sets created are associated with the same locale.

Note – Some implementations reserve the code set name of a locale as a special charset ID (`fontsetid` and `fontid`) within an `XmFontList` string. For this reason, application developers are cautioned *not* to use code set names if they want their applications to be portable across platforms.

Drawing Localized Text

A compound string is a means of encoding text so that it can be displayed in many different fonts without changing anything in the program. The desktop uses compound strings to display all text except that in the `Text` and `TextField` widgets. This section explains the structure of a compound string, the interaction between it and a font list (which determines how the compound string is displayed), and focuses on those aspects that are important to the internationalization process.

Compound String Components

A compound string is an internal encoding, consisting of tag-length-value segments. Semantically, a compound string has components that contain the text to be displayed, a tag (called a font list element tag) that is matched with an element of a font list, and an indicator denoting the direction in which it is to be displayed.

A compound string component can be one of the following four types:

- A font list element tag.
  - The font list element tag `XmFONTLIST_DEFAULT_TAG` indicates that the text is encoded in the code set of the current locale.
  - Other font list element tags are used later to match text with particular entries in a font list.
- A direction identifier.
- The text of the string. For internationalized applications, the text falls into two broad categories: either the text requires localized treatment or it does not.
- A separator.

The following describes each of the compound string components:

Font list element tag	Indicates a string value that correlates the text component of a compound string to a font or a font set in a font list.
Direction	Indicates the relationship between the order in which characters are entered on the keyboard and the order in which the characters are displayed on the screen. For

	example, the display order is left-to-right in English, French, German, and Italian, and right-to-left in Hebrew and Arabic.
Text	Indicates the text to be displayed.
Separator	Indicates a special form of a compound string component that has no value. It is used to separate other segments.

The desktop uses the specified font list element tag identified in the text component to display the compound string. A specified font list element tag is used until a new font list element tag is encountered. The desktop provides a special font list element tag, `XmFONTLIST_DEFAULT_TAG`, that matches a font that is correct for the current code set. It identifies the default entry in a font list. See "Compound Strings and Font Lists" on page 83 for more information.

The direction segment of a compound string specifies the direction in which the text is displayed. Direction can be left-to-right or right-to-left.

Compound Strings and Resources

Compound strings are used to display all text except that in the `Text` and `TextField` widgets. The compound string is set into the appropriate widget resource so that it can be displayed. For example, the label for the `PushButton` widget is inherited from the `Label` widget, and the resource is `XmNlabelString`, which is of type `XmString`. This means that the resource expects a value that is a compound string. A compound string can be created with a program or defined in a resource file.

Setting a Compound String Programmatically

An application can set this resource programmatically by creating the compound string using the `XmStringCreateLocalized()` compound string convenience function.

This function creates a compound string in the encoding of the current locale and automatically sets the font list entry tag to `XmFONTLIST_DEFAULT_TAG`.

The following code segment shows one way to set the XmNlabelString resource for a push button using a program.

```
#include <nl_types.h>
Widget  button;
Args    args[10];
int     n;
XmString button_label;
nl_msg my_catd;
(void)XtSetLanguageProc(NULL,NULL,NULL);
        .
        .
button_label = XmStringCreateLocalized (catgets(my_catd, 1, 1,
                "default label"),
                XmFONTLIST_DEFAULT_TAG);

/* Create an argument list for the button */
n = 0;
XtSetArg (args[n], XmNlabelString, button_label); n++;

/* Create and manage the button */
button = XmCreatePushButton (toplevel, "button", args, n);
XtManageChild (button);
XmStringFree (button_label);
```

Setting a Compound String in a Defaults File

In an internationalized program, the label string for the button label should be obtained from an external source. For example, the button label can come from a resource file instead of the program. For this example, assume that the push button is a child of a Form widget called form1.

```
*form1.button.labelString: Push Here
```

Here, the desktop's string-to-compound-string converter produces a compound string from the resource file text. This converter always uses XmFONTLIST_DEFAULT_TAG.

Compound Strings and Font Lists

When the desktop displays a compound string, it associates each segment with a font or font set by means of the font list element tag for that segment. The application must have loaded the desired font or font set, created a font list that contains that font or font set and its associated font list element tag, and created the compound string segment with the same tag.

The desktop follows a set search procedure when it binds a compound string to a font list entry in this way:

1. The desktop searches the font list for an exact match with the font list element tag specified in the compound string. If it finds a match, the compound string is bound to that font list entry.
2. If this does not provide a binding between the compound string and the font list, the desktop binds the compound string to the first element in the font list, regardless of its font list element tag.

For backward compatibility, if an exact match is not found, a value of `XmFONTLIST_DEFAULT_TAG` in either a compound string or a font list matches the tag that results from creating a compound string or font list entry with a tag of `XmSTRING_DEFAULT_CHARSET`.

Figure 4-1 on page 84 shows the relationships between a compound string, a font set, and a font list when the font list element tag is set to something other than `XmFONTLIST_DEFAULT_TAG`.

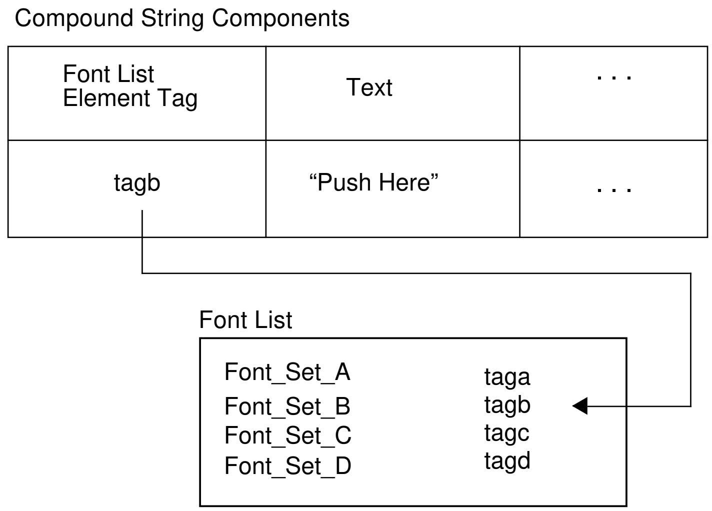


Figure 4-1 Relationships between compound strings, font sets, and font lists when the font list element tag is not XmFONTLIST_DEFAULT_TAG

The following example shows how to use a tag called `tagb`.

```
XFontSet    *font1;
XmFontListEntryfont_list_entry;
XmFontList font_list;
XmString label_text;
char**      missing;
int         missing_cnt;
char*       del_string;
char *tagb;    /* Font list element tag */
char *fontx;   /* Initialize to XLFD or font alias */
char *button_label;/* Contains button label text */
        .
        .
font1 = XCreateFontSet (XtDisplay(toplevel), fontx, & missing,
        & missing_cnt, & def_string);
font_list_entry = XmFontListEntryCreate (tagb, XmFONT_IS_FONTSET,
        (XtPointer)font1);
font_list = XmFontListAppendEntry (NULL, font_list_entry);
```

```
XmFontListEntryFree (font_list_entry);

label_text = XmStringCreate (button_label, tagb);
```

The `XCreateFontSet()` function loads the font set and the `XmFontListEntryCreate()` function creates a font list entry. The application must create an entry and append it to an existing font list or create a new font list. In either case, use the `XmFontListAppendEntry()` function. Because there is no font list in place, the preceding code example has a NULL value for the font list argument. The `XmFontListAppendEntry()` function creates a new font list called `font_list` with a single entry, `font_list_entry`. To add another entry to `font_list`, follow the same procedure but supply a nonnull font list argument.

Figure 4-2 shows the relationships between a compound string, a font set, and a font list when the font list element tag is set to `XmFONTLIST_DEFAULT_TAG`. In this case, the value field is locale text.

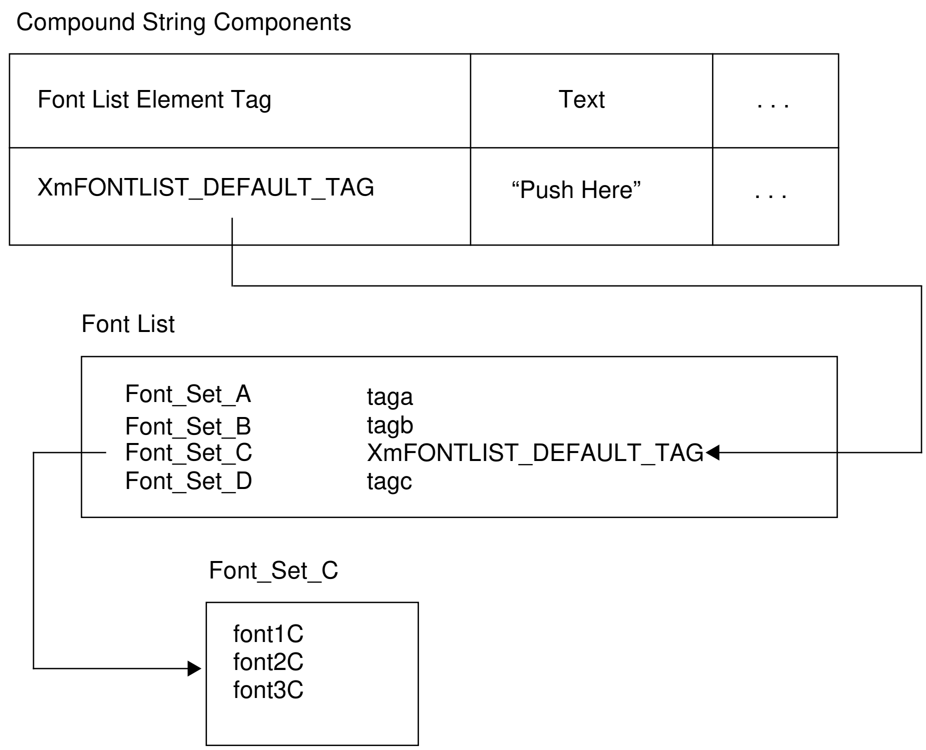


Figure 4-2 Relationships between compound strings, font sets, and font lists when a font list element tag is set to XmFONTLIST_DEFAULT_TAG

Here, the default tag points to Font_Set_C, which in turn identifies the fonts needed to display the characters in the language.

Text and TextField Widgets and Font Lists

The Text and TextField widgets display text information. To do so, they must be able to select the correct font in which to display the information. The Text and TextField widgets follow a set search pattern to find the correct font as follows:

1. The widget searches the font list for an entry that is a font set and has a font list element tag of XmFONTLIST_DEFAULT_TAG. If a match is found, it uses that font list entry. No further searching occurs.
2. The widget searches the font list for an entry that specifies a font set. It uses the first one found.
3. If no font set is found, the widget uses the first font in the font list.

Using a font set ensures that there are glyphs for every character in the locale.

Inputting Localized Text

In the system environment, the VendorShell widget class is enhanced to provide the interface to the input method. While the VendorShell class controls only one child widget in its geometry management, an extension has been added to the VendorShell class to enhance it for managing all components necessary in the interface to an input method. These components include the status area, preedit area, and the MainWindow area.

When the input method requires a status area or a preedit area or both, the VendorShell widget automatically instantiates the status and preedit areas and manages their geometry layout. Any status area or preedit area is managed by the VendorShell widget internally and is not accessible by the client. The widget instantiated as the child of the VendorShell widget is called the MainWindow area.

The input method to be used by the VendorShell widget is determined by the XmNinputMethod resource; for example, @im=alt. The default value of Null indicates to choose the default input method associated with the locale at the time that VendorShell is created. As such, the user can affect which input method is selected by either setting the locale, setting the XmNinputMethod resource, or setting both. The locale name is concatenated with the XmNinputMethod resource to determine the input method name. The locale name must not be specified in this resource. The modifier name for the XmNinputMethod resource needs to be in the form @im=*modifier*, where *modifier* is the string used to qualify which input method is selected.

The VendorShell widget can support multiple widgets that can share the input method. Yet only one widget can have the keyboard focus (for example, receive key press events and send them to an input method) at any given time. To support multiple widgets (such as Text widgets), the widgets need to be descendants of the VendorShell widget.

Note – The VendorShell widget class is a superclass of the TransientShell and TopLevelShell widget classes. As such, an instantiation of a TopLevelShell or a DialogShell is essentially an instantiation of a VendorShell widget class.

The VendorShell widget behaves as an input manager only if one of its descendants is an XmText[Field] instance. As soon as an XmText[Field] instance is created as a descendant of the VendorShell widget, VendorShell creates the necessary areas required by the particular input methods dictated by the current locale. Even if an XmText[Field] instance is not mapped but just created, VendorShell has the geometry management behavior as described previously.

A VendorShell widget does the following:

- Enables applications to process multibyte character input and output that is supported by the locales installed in the system.
- Manages an input method instance as defined in the XmIm reference functions.
- Supports preediting within a preedit area in either OffTheSpot, OverTheSpot, Root, or None mode. Localized text can be entered into any Text child widget in a multiple Text children widget tree by changing the focus.
- Provides geometry management for descendant child widgets.

Geometry Management

The `VendorShell` widget provides geometry management and focus management for the input method's user interface components, as necessary. If the locale warrants it (for example, if the locale is a Japanese Extended UNIX Code (EUC) locale), the `VendorShell` widget automatically allocates and manages the geometry of any required preedit area or status area or both.

Depending on the current preediting being done, an auxiliary area may be required. If so, the `VendorShell` widget also instantiates and manages the auxiliary area. Typically, the child of the `VendorShell` widget is a container widget (such as the `XmBulletinBoard` or `XmRowColumn` widgets) that can manage multiple `Text` and `TextField` widgets, which allow multibyte character input from the user. In this scenario, all `Text` widgets share the same input method.

Note – The status, preedit, and auxiliary areas are not accessible to the application programmer. For example, it is not intended for the application programmer to access the window ID of the status area. The user does not need to worry about the instantiation or management of these components as they are managed as required by the `VendorShell` widget class.

The application programmer has some control over the behavior of the input method user interface components through `XmNpreeditType` resources of the `VendorShell` widget class. See "Input Methods" on page 13 for a description of OffTheSpot and OverTheSpot modes.

Geometry management extends to all input method user interface components. When the application program window (a `TopLevelShell` widget) is resized, the input method user interface components are resized accordingly, and the preedited strings in them are rearranged as required. Of course, this assumes that the shell window has a resize policy of True.

When the `VendorShell` widget is created, if a specific input method requires a status area, preedit area, or both, the size of the VendorShell considers the areas required by these components. The extra areas required by the preedit and status areas are part of the `VendorShell` widget's area. They are also managed by the `VendorShell` widget, if resizing is necessary.

Because of the potential instantiation of these areas (status and preedit), depending on the input method currently being used, the size of the `VendorShell` widget area does not necessarily grow or shrink to accommodate exactly the size of its child. The size of the `VendorShell` widget area grows or shrinks to accommodate both its child's geometry *and* the geometry of these input method user interface areas. There may be a difference (for example, of 20 pixels) in height between the `VendorShell` widget and its child widget (the MainWindow area). The width geometry is *not* affected by the input method user interface components.

In summary, the requested size of the child is honored if possible; the actual size of the `VendorShell` may be larger than its child.

The requests to specify the geometry of the `VendorShell` widget and its child are honored as long as they do not conflict with each other or are within the constraint of the `VendorShell` widget's ability to resize. When they do conflict, the child's widget geometry request has higher precedence. For example, if the size of the child widget is specified as 100x100, the size of VendorShell is also specified as 100x100. The resulting VendorShell has a size of 100x120, while its child widget gets a size of 100x100. If the size of the child widget is not specified, the VendorShell shrinks its child widget if necessary to honor its own size specification. For example, if the size of VendorShell is specified as 100x100 and no size is specified for its child, the child widget has a size of 100x80. If the `VendorShell` widget is disabled from resizing, regardless of what the geometry request of its child is, the `VendorShell` widget honors only its own geometry specification.

Focus Management

Languages with large numbers of characters (such as Japanese and Chinese) require an input method that allows the user to compose characters in that language interactively. This is because, for these languages, there are many more characters than can be reasonably mapped to a terminal keyboard.

The interactive process of composing characters in such languages is called *preediting*. The preediting itself is handled by the input method. However, the user interface of the preediting is determined by the system environment. An interface needs to exist between the input method and the system environment. This is done through the `VendorShell` widget of the system environment.

Figure 4-3 illustrates a case with Japanese preediting. The string shown in reverse video is the string in preediting. This string can be moved across different windows by giving focus to the particular window. However, only one preediting session can occur at one time.

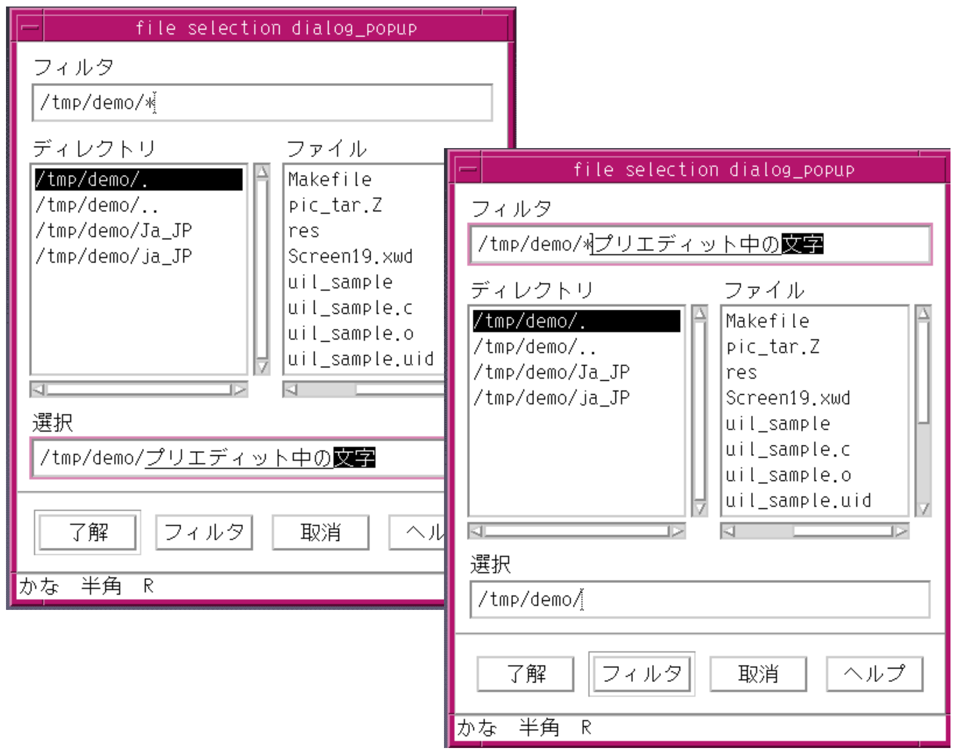


Figure 4-3 Japanese preediting example

For an example of focus management, suppose a `TopLevelShell` widget (a subclass of the `VendorShell` widget) has an `XmBulletinBoard` widget child (MainWindow area), which has five `XmText` widgets as children. Assume the locale requires the preedit area, and assume the OverTheSpot mode is specified. Because the `VendorShell` widget manages only one instance of an input method, you can run only one preedit area at a time inside the `TopLevelShell` widget. If the focus is moved from one `Text` widget to another, the current preedit string under construction is also moved on top of the `Text` widget that currently has focus. Processing of keys to the old `Text` widget is suspended temporarily. Subsequent interface of the input method, such as the delivery of the string at preedit completion, is made to the new, focused `Text` widget.

The string being preedited can be moved to the location of the focus; for example, by clicking the mouse.

A string that the end user is finished preediting and that is already confirmed *cannot* be reconverted. Once the string is composed, it is committed. Committing a string means that it is moved from the preedit area to the focus point of the client.

Internationalized User Interface Language

The capability to parse a multibyte character string as a string literal has been added to the User Interface Language (UIL). Creation of a UIL file is performed by using the characteristics of the target language and writing the User Interface Definition (UID) file.

Programming for Internationalized User Interface Language

The UIL compiler parses nonstandard charsets as locale text. This requires the UIL compiler to be run in the same locale as any locale text.

If the locale text of a widget requires a font set (more than one font), the font set must be specified within the resource file. The *font* parameter does not support font sets.

To use a specific language with UIL, a UIL file is written according to characteristics of the target language and compiled into a UID file. The UIL file that contains localized text needs to be compiled in the locale in which it is to run.

String Literals

The following shows examples of literal strings. The `cur_charset` value is always set to the default_charset value, which allows the string literal to contain locale text.

To set locale text in the string literal with the default_charset value, enter the following:

```
XmNlabelString = 'XXXXXX';
```

OR

```
XmNlabelString = #default_charset"XXXXXX";
```

Compile the UIL file with the LANG environment variable matching the encoding of the locale text. Otherwise, the string literal is not compiled properly.

Font Sets

The font set cannot be set through UIL source programming. Whenever the font set is required, you must set it in the resource file as the following example shows:

```
*fontList: -*-r-*-20-*:
```

Font Lists

UIL has three functions that are used to create font lists: FONT, FONTSET, and FONT_TABLE. The FONT and FONTSET functions create font list entries. The FONT_TABLE function creates a font list from these font list entries.

The FONT function creates a font list entry containing a font specification. The argument is a string representing an XLFD font name. The FONTSET function creates a font list entry containing a font set specification. The argument is a comma-separated list of XLFD font names representing a base name font list.

Both FONT and FONTSET have optional CHARACTER_SET declaration parameters that specify the font list element tag for the font list entry. In both cases, if no CHARACTER_SET declaration parameter is specified, UIL determines the font list element tag as follows:

- If the module contains no CHARACTER_SET declaration and if the uil command was called with the -s option or the Uil() function was started with use_setlocale_flag set, the font list element tag is XmFONTLIST_DEFAULT_TAG.

- Otherwise, the font list element tag is the code set component of the LANG environment variable, if it is set in the UIL compilation environment; or it is the value of XmFALLBACK_CHARSET if the LANG environment variable is not set or has no code set.

The FONT_TABLE function creates a font list from a comma-separated list of font list entries created by FONT or FONTSET. The resulting font list can be used as the value of a font list resource. If a single font list entry is supplied as the value for such a resource, UIL converts the entry to a font list.

Creating Resource Files

If necessary, set the input method-related resources in the resource file as shown in the following example:

```
*preeditType:  OverTheSpot, OffTheSpot, Root, or None
```

Setting the Environment

For a locale-sensitive application, set the UID file to the appropriate directory. Set the UIDPATH or XAPPLRESDIR environment variable to the appropriate value.

For example, to run the uil_sample program with an English environment (LANG environment variable is en_US), set uil_sample.uid with Latin characters at the $HOME/en_US directory, or set uil_sample.uid to a directory and set the UIDPATH environment variable to the full path name of the uil_sample.uid file.

To run the uil_sample program with a Japanese environment (LANG environment variable is ja_JP), create a uil_sample.uid file with Japanese (multibyte) characters at the $HOME/ja_JP directory, or place uil_sample.uid to a unique directory and set the UIDPATH environment variable to the full path name of the uil_sample.uid file. The following list specifies the possible variables:

%U	Specifies the UID file string.
%N	Specifies the class name of the application.
%L	Specifies the value of the xnlLanguage resource or LC_CTYPE category.
%l	Specifies the language component of the xnlLanguage resource or the LC_CTYPE category.

If the `XAPPLRESDIR` environment variable is set, the `MrmOpenHierarchy()` function searches the UID file in the following order:

1. UID file path name
2. `$UIDPATH`
3. %*U*
4. `$XAPPLRESDIR/`%*L*`/uid/`%*N*`/`%*U*
5. `$XAPPLRESDIR/`%*l*`/uid/`%*N*`/`%*U*
6. `$XAPPLRESDIR/uid/`%*N*`/`%*U*
7. `$XAPPLRESDIR/`%*L*`/uid/`%*U*
8. `$XAPPLRESDIR/`%*l*`/uid/`%*U*
9. `$XAPPLRESDIR/uid/`%*U*
10. `$HOME/uid/`%*U*
11. `$HOME/`%*U*
12. `/usr/lib/X11/`%*L*`/uid/`%*N*`/`%*U*
13. `/usr/lib/X11/`%*l*`/uid/`%*N*`/`%*U*
14. `/usr/lib/X11/uid/`%*N*`/`%*U*
15. `/usr/lib/X11/`%*L*`/uid/`%*U*
16. `/usr/lib/X11/`%*l*`/uid/`%*U*
17. `/usr/lib/X11/uid/`%*U*
18. `/usr/include/X11/uid/`%*U*

If the `XAPPLRESDIR` environment variable is not set, the `MrmOpenHierarchy()` function uses `$HOME` instead of the `XAPPLRESDIR` environment variable.

default_charset Character Set in UIL

With the `default_charset` string literal, any characters can be set as a valid string literal. For example, if the `LANG` environment variable is `el_GR`, the string literal with `default_charset` can contain any Greek character. If the `LANG` environment variable is `ja_JP`, the `default_charset` string literal can contain any Japanese character encoded in Japanese EUC.

If no character set is set to a string literal, the character set of the string literal is set as `cur_charset`. And, in the system environment, the `cur_charset` value is always set as `default_charset`.

Example: uil_sample

Figure 4-4 shows a UIL sample program on English and Japanese environments.

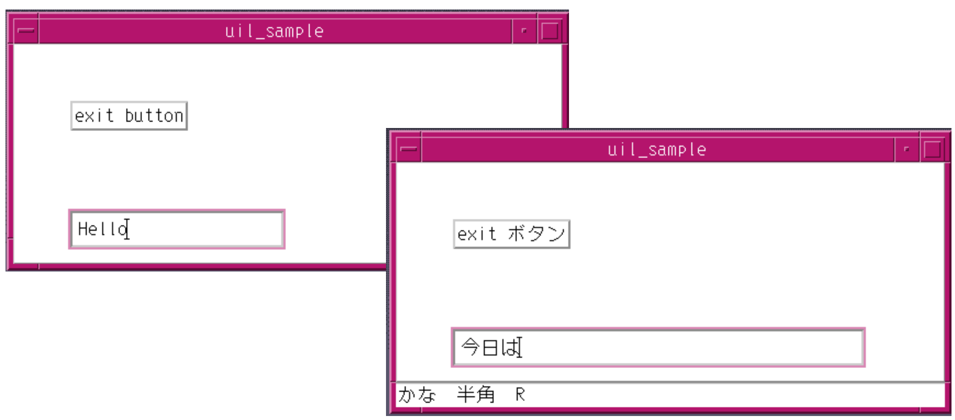


Figure 4-4 Sample UIL program on English and Japanese environments

In the following sample program, *LLL* indicates locale text, which can be Japanese, Korean, Traditional Chinese, Greek, French, or others.

```
uil_sample.uil
!
!       sample uil file - uil_sample.uil
!
!           C source file - uil_sample.c
!
!               Resource file - uil-sample.resource
```

```
!
module Test
        version = 'v1.0'
        names = case_sensitive
        objects = {
            XmPushButton = gadget;
        }
!************************************
!       declare callback procedure
!************************************
procedure
        exit_CB ;
!*****************************************************************
!       declare BulletinBoard as parent of PushButton and Text
!*****************************************************************
object
        bb : XmBulletinBoard {
            arguments{
                XmNwidth = 500;
                XmNheight = 200;
            };
            controls{
                XmPushButton   pb1;
                XmText     text1;
            };
        };
!****************************
!       declare PushButton
!****************************
object
        pb1 : XmPushButton {
            arguments{
                XmNlabelString = #Normal "LLLexit buttonLLL";
                XmNx = 50;
                XmNy = 50;
            };
            callbacks{
                XmNactivateCallback = procedure exit_CB;
            };
        };
!*********************
!       declare Text
!*********************
        text1 : XmText {
            arguments{
                XmNx = 50;
```

```
                XmNy = 150;
            };
        };
end module;
*
*           C source file - uil_sample.c
*
*/
#include <Mrm/MrmAppl.h>
#include <locale.h>
void exit_CB();
static      MrmHierarchy   hierarchy;
static      MrmType     *class;

/*******************************************/
/*      specify the UID hierarchy list*/
/******************************************/
static      char        *aray_file[]=
                    {"uil_sample.uid"
                    };
static      int     num_file = (sizeof aray_file / sizeof
aray_file[0]);
/*****************************************************/
/*      define the mapping between UIL procedure names*/
/*      and their addresses           */
/*****************************************************/
static      MRMRegisterArg reglist[]={
                    {"exit_CB",(caddr_t) exit_CB}
```

Compound Strings in UIL

Three mechanisms exist for specifying strings in UIL files:

- As string literals, which may be stored in UID files as either null-terminated strings or compound strings.
- As compound strings.
- As wide character strings.

Both string literals and compound strings consist of text, a character set, and a writing direction. For string literals and for compound strings with no explicit direction, UIL infers the writing direction from the character set. The UIL concatenation operator (&) concatenates both string literals and compound strings.

Regardless of whether UIL stores string literals in UID files as null-terminated strings or as compound strings, it stores information about each string's character set and writing direction along with the text. In general, UIL stores string literals or string expressions as compound strings in UID files under the following conditions:

- When a string expression consists of two or more literals with different character sets or writing directions.
- When the literal or expression is used as a value that has a compound string data type (such as the value of a resource whose data type is compound string).

UIL recognizes a number of keywords specifying character sets. UIL associates parsing rules, including parsing direction and whether characters have 8 or 16 bits, for each character set it recognizes. It is also possible to define a character set using the UIL `CHARACTER_SET` function.

The syntax of a string literal is one of the following:

- **'**[*character_string*]**'**
- [#*char_set*]
- **"**[*character_string*]**"**

For each syntax, the character set of the string is determined as follows:

- For a string declared as *'character_string***'**, the character set is the code set component of the `LANG` environment variable, if it is set in the UIL compilation environment; or it is the value of `XmFALLBACK_CHARSET` if the `LANG` environment variable is not set or has no code set. By default, the value of `XmFALLBACK_CHARSET` is ISO8859-1, but vendors may supply different values.
- For a string declared as #*char_set* **"***string***"**, the character set is *char_set*.
- For a string declared as **"***character_string***"**, the character set depends on whether the module has a `CHARACTER_SET` clause and whether the UIL compiler's `use_setlocale_flag` is set.
  - If the module has a `CHARACTER_SET` clause, the character set is the one specified in that clause.
  - If the module has no `CHARACTER_SET` clause but the `uil` command was started with the `-s` option, or if the `Uil()` function was started with `use_setlocale_flag` set, UIL calls the `setlocale()` function and parses the string in the current locale. The character set of the resulting string is `XmFONTLIST_DEFAULT_TAG`.

- If the module has no CHARACTER_SET clause and the uil command was started without the -s option, or if the Uil() function was started without use_setlocale_flag, the character set is the code set component of the LANG environment variable, if it is set in the UIL compilation environment, or the character set is the value of XmFALLBACK_CHARSET if LANG is not set or has no code set.

UIL always stores a string specified using the COMPOUND_STRING function as a compound string. This function takes as arguments a string expression and optional specifications of a character set, direction, and whether to append a separator to the string. If no character set or direction is specified, UIL derives it from the string expression, as described in the preceding section.

Note – Certain predefined escape sequences, beginning with a \ (backslash), may be displayed in string literals, with the following exceptions:

- A string in single quotation marks can span multiple lines, with each new line character escaped by a backslash. A string in double quotation marks cannot span multiple lines.
- Escape sequences are processed literally inside a string that is parsed in the current locale (a localized string).

Xt and Xlib Dependencies 5

This chapter discusses tasks related to internationalizing with Xt and Xlib.

Locale Management

The following defines support for the locale mechanism that controls all locale-dependent Xlib and TriTeal Enterprise Desktop functions.

X Locale Management

X locale supports one or more of the locales defined by the host environment. The Xlib conforms to the American National Standards Institute (ANSI) C library, and the locale announcement method is the `setlocale()` function. This function configures the locale operation of both the host C library and Xlib. The operation of Xlib is governed by the `LC_CTYPE` category; this is called the current locale.

The `XSupportsLocale()` function is used to determine whether the current locale is supported by X.

The client is responsible for selecting its locale and X modifiers. Clients should provide a means for the user to override the clients' locale selection at client invocation. Most single-display X clients operate in a single locale for both X and the host-processing environment. They configure the locale by calling three functions: `setlocale()`, `XSupportsLocale()`, and `XSetLocaleModifiers()`.

The semantics of certain categories of X internationalization capabilities can be configured by setting modifiers. Modifiers are named by implementation-dependent and locale-specific strings. The only standard use for this capability at present is selecting one of several styles of keyboard input methods.

The `XSetLocaleModifiers()` function is used to configure Xlib locale modifiers for the current locale.

The recommended procedure for clients initializing their locale and modifiers is to obtain locale and modifier announcers separately from one of the following prioritized sources:

1. A command-line option
2. A resource
3. The empty string (“ ”)

The first of these that is defined should be used.

Note – When a locale command-line option or locale resource is defined, the effect should be to set all categories to the specified locale, overriding any category-specific settings in the local host environment.

Locale and Modifier Dependencies

The internationalized Xlib functions operate in the current locale configured by the host environment and in the X locale modifiers set by the `XSetLocaleModifiers()` function, or in the locale and modifiers configured at the time some object supplied to the function was created. For each locale-dependent function, Table 5-1 lists locale and modifier dependencies.

Table 5-1 Locale and Modifier Dependencies

Locale from...	Affects the Function...	In the...
	Locale Query/Configuration	
setlocale	XSupportsLocale XSetLocaleModifiers	Locale queried Locale modified
	Resources	
setlocale	XrmGetFileDatabase XrmGetStringDatabase	Locale of XrmDatabase
XrmDatabase	XrmPutFileDatabase XrmLocaleOfDatabase	Locale of XrmDatabase
	Setting Standard Properties	
setlocale	XmbSetWMProperties	Encoding of supplied returned text (some WM_ property text in environment locale)
setlocale	XmbTextPropertyToTextList XwcTextPropertyToTextList XmbTextListToTextProperty XwcTextListToTextProperty	Encoding of supplied/returned text
	Text Input	
setlocale	XOpenIM	XIM input method
XIM	XCreateIC XLocaleOfIM, etc.	XIC input method configuration Queried locale
XIC	XmbLookupText XwcLookupText	Keyboard layout Encoding of returned text
	Text Drawing	
setlocale	XCreateFontSet	Charsets of fonts in XFontSet
XFontSet	XmbDrawText, XwcDrawText, etc. XExtentsOfFontSet, etc. XmbTextExtents, XwcTextExtents, etc.	Locale of supplied text Locale of supplied text Locale-dependent metrics

Table 5-1 Locale and Modifier Dependencies *(Continued)*

Locale from...	Affects the Function...	In the...
	Xlib Errors	
`setlocale`	`XGetErrorDatabaseText` `XGetErrorText`	Locale of error message

Clients can assume that a locale-encoded text string returned by an X function can be passed to a C library function, or the string result of a C library function can be passed to an X function, if the locale is the same at the two calls.

All text strings processed by internationalized Xlib functions are assumed to begin in the initial state of the encoding of the locale, if the encoding is state-dependent. All Xlib functions behave as if they do not change the current locale or X modifier setting. (This means that any function, provided within a library either by Xlib or by the application, that changes the locale or calls the `XSetLocaleModifiers()` function with a nonnull argument, must save and restore the current locale state on entry and exit.) Also, Xlib functions on implementations that conform to the ANSI C library do not alter the global state associated with the `mblen()`, `mbtowc()`, `wctomb()`, `and strtok()` ANSI C functions.

Xt Locale Management

Xt locale management includes the following two functions:

- `XtSetLanguageProc()`
- `XtDisplayInitialize()`

XtSetLanguageProc

Before the initialization of the Xt Toolkit, applications should normally call the `XtSetLanguageProc()` function with one of the following functions:

```
XtSetLanguageProc (NULL, NULL, NULL)
```

Note – The locale is not actually set until the toolkit is initialized (for example, by way of the `XtAppInitialize()` function). Therefore, the `setlocale()` function may be needed after the `XtSetLanguageProc()` function and the initializing of the toolkit (for example, if calling the `catopen()` function).

Resource databases are created in the current process locale. During display initialization prior to creating the per-screen resource database, the Intrinsics call to a specified application procedure to set the locale according to options found on the command line or in the per-display resource specifications.

The callout procedure provided by the application is of type `XtLanguageProc`, as in the following syntax:

```
typedef String(*XtLanguageProc)(displayID, languageID, clientdata);
Display *displayID;
String languageID;
XtPointer clientdata;
```

displayID Passes the display.

languageID Passes the initial language value obtained from the command line or server per-display resource specifications.

clientdata Passes the additional client data specified in the call to the `XtSetLanguageProc()` function.

The language procedure allows an application to set the locale to the value of the language resource determined by the `XtDisplayInitialize()` function. The function returns a new language string that is subsequently used by the `XtDisplayInitialize()` function to establish the path for loading resource files. This string is cached and is the locale of the display.

Initially, no language procedure is set by the intrinsics. To set the language procedure for use by the `XtDisplayInitialize()` function, use the `XtSetLanguageProc()` function:

```
XtLanguageProc XtSetLanguageProc(applicationcontext, procedure, clientdata)
XtAppContext applicationcontext;
XtLanguageProc procedure;
XtPointer clientdata;
```

applicationcontext Specifies the application context in which the language procedure is to be used or specifies a null value.

procedure Specifies the language procedure.

clientdata Specifies additional client data to be passed to the language procedure when it is called.

The `XtSetLanguageProc()` function sets the language procedure that is called from the `XtDisplayInitialize()` function for all subsequent displays initialized in the specified application context. If the *applicationcontext* parameter is null, the specified language procedure is registered in all application contexts created by the calling process, including any future application contexts that may be created. If the *procedure* parameter is null, a default language procedure is registered. The `XtSetLanguageProc()` function returns the previously registered language procedure. If a language procedure has not yet been registered, the return value is unspecified; but if this return value is used in a subsequent call to the `XtSetLanguageProc()` function, it causes the default language procedure to be registered.

The default language procedure does the following:

- Sets the locale according to the environment. On ANSI C-based systems, this is done by calling the setlocale (`LC_ALL`, "*language*") function. If an error is encountered, a warning message is issued with the `XtWarning()` function.
- Calls the `XSupportsLocale()` function to verify that the current locale is supported. If the locale is not supported, a warning message is issued with the `XtWarning()` function and the locale is set to "C".
- Calls the `XSetLocaleModifiers()` function specifying the empty string.
- Returns the value of the current locale. On ANSI C-based systems, this is the return value from a final call to the setlocale (`LC_CTYPE`, `NULL`) function.

A client can use this mechanism to establish a locale by calling the `XtSetLanguageProc()` function prior to the `XtDisplayInitialize()` function, as in the following example.

```
Widget top;
XtSetLanguageProc(NULL, NULL, NULL);
top = XtAppInitialize( ... );
...
```

XtDisplayInitialize

The `XtDisplayInitialize()` function first determines the language string to be used for the specified display and loads the application's resource database for this display-host-application combination from the following sources in order of precedence:

1. Application command line (argv.)

2. Per-host user environment resource file on the local host.

3. Resource property on the server or user-preference resource file on the local host.
4. Application-specific user resource file on the local host.
5. Application-specific class resource file on the local host.

The `XtDisplayInitialize()` function creates a unique resource database for each *display* parameter specified. When a database is created, a language string is determined for the *display* parameter in a manner equivalent to the following sequence of actions.

The `XtDisplayInitialize()` function initially creates two temporary databases. The first database is constructed by parsing the command line. The second database is constructed from the string returned by the `XResourceManagerString()` function or, if the `XResourceManagerString()` function returns a null value, the contents of a resource file in the user's home directory. The name for this user-preference resource file is `$HOME/.Xdefaults`.

The database constructed from the command line is then queried for the resource *name*.`xnlLanguage`, class *class*.`XnlLanguage`, where *name* and *class* are the specified application name and application class. If this database query is unsuccessful, the server resource database is queried; if this query is also unsuccessful, the language is determined from the environment. This is done by retrieving the value of the `LANG` environment variable. If no language string is found, the empty string is used.

The application-specific class resource file name is constructed from the class name of the application. It points to a localized resource file that is usually installed by the site manager when the application is installed. The file is found by calling the `XtResolvePathname()` function with the parameters (*displayID*, *applicationdefaults*, `NULL`, `NULL`, `NULL`, `NULL`, `0`, `NULL`). This file should be provided by the developer of the application because it may be required for the application to function properly. A simple application that needs a minimal set of resources in the absence of its class resource file can declare fallback resource specifications with the `XtAppSetFallbackResources()` function.

The application-specific user resource file name points to a user-specific resource file and is constructed from the class name of the application. This file is owned by the application and typically stores user customizations. Its name is found by calling the `XtResolvePathname()` function with the parameters (*displayID*, NULL, NULL, NULL, *path*, NULL, 0, NULL), where *path* is defined in an operating-system-specific manner. The *path* variable is defined to be the value of the XUSERFILESEARCHPATH environment variable if this is defined. Otherwise, the default is vendor-defined.

If the resulting resource file exists, it is merged into the resource database. This file can be provided with the application or created by the user.

The temporary database created from the server resource property or user resource file during language determination is then merged into the resource database. The server resource file is created entirely by the user and contains both display-independent and display-specific user preferences.

If one exists, a user's environment resource file is then loaded and merged into the resource database. This file name is user- and host-specific. The user's environment resource file name is constructed from the value of the user's XENVIRONMENT environment variable for the full path of the file. If this environment variable does not exist, the XtDisplayInitialize() function searches the user's home directory for the .Xdefaults-*host* file, where *host* is the name of the machine on which the application is running. If the resulting resource file exists, it is merged into the resource database. The environment resource file is expected to contain process-specific resource specifications that are to supplement those user-preference specifications in the server resource file.

Font Management

International text drawing is done using a set of one or more fonts, as needed for the locale of the text.

The two methods of internationalized drawing within the system environment allow clients to choose one of the static output widgets (for example, XmLabel) or to choose the DrawingArea widget to draw with any other primitive function.

Static output widgets require that text be converted to XmString.

The following information explains the mechanism for managing fonts using the Xlib routines and functions.

Creating and Freeing a Font Set

Xlib international text drawing is done using a set of one or more fonts, as needed for the locale of the text. Fonts are loaded according to a list of base font names supplied by the client and the charsets required by the locale. The `XFontSet` is an opaque type.

- The `XCreateFontSet()` function is used to create an international text drawing font set.
- The `XFontsOfFontSet()` function is used to obtain a list of `XFontStruct` structures and full font names given an XFontSet.
- To obtain the base font name list and the selected font name list given an `XFontSet`, use the `XBaseFontNameListOfFontSet()` function.
- To obtain the locale name given an `XFontSet`, use the `XLocaleOfFontSet()` function.
- The `XLocaleOfFontSet()` function returns the name of the locale bound to the specified `XFontSet` as a null-terminated string.
- The `XFreeFontSet()` function frees the specified font set. The associated base font name list, font name list, `XFontStruct` list, and `XFontSetExtents`, if any, are freed.

Obtaining Font Set Metrics

Metrics for the internationalized text drawing functions are defined in terms of a *primary draw direction*, which is the default direction in which the character origin advances for each succeeding character in the string. The Xlib interface is currently defined to support only a left-to-right primary draw direction. The *drawing origin* is the position passed to the drawing function when the text is drawn. The *baseline* is a line drawn through the drawing origin parallel to the primary draw direction. *Character ink* is the pixels painted in the foreground color and does not include interline or intercharacter spacing or image text background pixels.

The drawing functions are allowed to implement implicit text direction control, reversing the order in which characters are rendered along the primary draw direction in response to locale-specific lexical analysis of the string.

Regardless of the character rendering order, the origins of all characters are on the primary draw direction side of the drawing origin. The screen location of a particular character image may be determined with the `XmbTextPerCharExtents()` or `XwcTextPerCharExtents()` functions.

The drawing functions are allowed to implement context-dependent rendering, where the glyphs drawn for a string are not simply a combination of the glyphs that represent each individual character. A string of two characters drawn with the `XmbDrawString()` function may render differently than if the two characters were drawn with separate calls to the `XmbDrawString()` function. If the client adds or inserts a character in a previously drawn string, the client may need to redraw some adjacent characters to obtain proper rendering.

The drawing functions do not interpret newline characters, tabs, or other control characters. The behavior when nonprinting characters are drawn (other than spaces) is implementation-dependent. It is the client's responsibility to interpret control characters in a text stream.

To find out about context-dependent rendering, use the `XContextDependentDrawing()` function. The `XExtentsOfFontSet()` function obtains the maximum extents structure given an XFontSet. The `XmbTextEscapement()` and `XwcTextEscapement()` functions obtain the escapement in pixels of the specified text as a value. The `XmbTextExtents()` and `XwcTextExtents()` functions obtain the overall bounding box of the string's image and a logical bounding box (*overall_ink_return* and *overall_logical_return* arguments respectively). The `XmbTextPerCharExtents()` and `XwcTextPerCharExtents()` functions return the text dimensions of each character of the specified text, using the fonts loaded for the specified font set.

Drawing Localized Text

The functions defined in this section draw text at a specified location in a drawable. They are similar to the `XDrawText()`, `XDrawString()`, and `XDrawImageString()` functions except that they work with font sets instead of single fonts, and they interpret the text based on the locale of the font set instead of treating the bytes of the string as direct font indexes. If a BadFont error is generated, characters prior to the offending character may have been drawn.

The text is drawn using the fonts loaded for the specified font set; the font in the graphics context (GC) is ignored and may be modified by the functions. No validation that all fonts conform to some width rule is performed.

Use the `XmbDrawText()` or `XwcDrawText()` function to draw text using multiple font sets in a given drawable. To draw text using a single font set in a given drawable, use the `XmbDrawString()` or `XwcDrawString()` function. To draw image text using a single font set in a given drawable, use the `XmbDrawImageString()` or `XwcDrawImageString()` function.

Inputting Localized Text

The following discusses the Xlib and desktop mechanisms used for international text input. If you are using Motif `Text[Field]` widgets or you are using the XmIm APIs for text input, this section provides background information. However, it will not impact your application design or coding practice. If you are not interested in how character input is achieved from the keyboard with low-level Xlib calls, you can proceed to "Interclient Communications Conventions for Localized Text" on page 122.

Xlib Input Method Overview

This section provides definitions for terms and concepts used for internationalized text input and a brief overview of the intended use of the mechanisms provided by Xlib.

A large number of languages in the world use alphabets consisting of a small set of symbols (letters) to form words. To enter text into a computer in an alphabetic language, a user usually has a keyboard on which there are key symbols corresponding to the alphabet. Sometimes, a few characters of an alphabetic language are missing on the keyboard. Many computer users who speak a Latin-alphabet-based language only have an English-based keyboard. They need to press a combination of keystrokes to enter a character that does not exist directly on the keyboard. A number of algorithms have been developed for entering such characters, known as European input methods, the compose input method, or the dead-keys input method.

Japanese is an example of a language with a phonetic symbol set, where each symbol represents a specific sound. There are two phonetic symbol sets in Japanese: *Katakana* and *Hiragana*. In general, Katakana is used for words that are of foreign origin, and Hiragana for writing native Japanese words. Collectively, the two systems are called *Kana*. Hiragana consists of 83 characters; Katakana, 86 characters.

Korean also has a phonetic symbol set, called *Hangul*. Each of the 24 basic phonetic symbols (14 consonants and 10 vowels) represent a specific sound. A syllable is composed of two or three parts: the initial consonants, the vowels, and the optional last consonants. With Hangul, syllables can be treated as the basic units on which text processing is done. For example, a delete operation may work on a phonetic symbol or a syllable. Korean code sets include several thousands of these syllables. A user types the phonetic symbols that make up the syllables of the words to be entered. The display may change as each phonetic symbol is entered. For example, when the second phonetic symbol of a syllable is entered, the first phonetic symbol may change its shape and size. Likewise, when the third phonetic symbol is entered, the first two phonetic symbols may change their shape and size.

Not all languages rely solely on alphabetic or phonetic systems. Some languages, including Japanese and Korean, employ an *ideographic* writing system. In an ideographic system, rather than taking a small set of symbols and combining them in different ways to create words, each word consists of one unique symbol (or, occasionally, several symbols). The number of symbols may be very large: approximately 50,000 have been identified in *Hanzi*, the Chinese ideographic system.

There are two major aspects of ideographic systems for their computer usage. First, the standard computer character sets in Japan, China, and Korea include roughly 8,000 characters, while sets in Taiwan have between 15,000 and 30,000 characters, which make it necessary to use more than one byte to represent a character. Second, it is obviously impractical to have a keyboard that includes all of a given language's ideographic symbols. Therefore a mechanism is required for entering characters so that a keyboard with a reasonable number of keys can be used. Those input methods are usually based on phonetics, but there are also methods based on the graphical properties of characters.

In Japan, both Kana and Kanji are used. In Korea, Hangul and sometimes Hanja are used. Now, consider entering ideographs in Japan, Korea, China, and Taiwan.

In Japan, either Kana or English characters are entered and a region is selected (sometimes automatically) for conversion to Kanji. Several Kanji characters can have the same phonetic representation. If that is the case, with the string entered, a menu of characters is presented and the user must choose the appropriate option. If no choice is necessary or a preference has been established, the input method does the substitution directly. When Latin characters are converted to Kana or Kanji, it is called a *Romaji conversion*.

In Korea, it is usually acceptable to keep Korean text in Hangul form, but some people may choose to write Hanja-originated words in Hanja rather than in Hangul. To change Hangul to Hanja, a region is selected for conversion and the user follows the same basic method as described for Japanese.

Probably because there are well-accepted phonetic writing systems for Japanese and Korean, computer input methods in these countries for entering ideographs are fairly standard. Keyboard keys have both English characters and phonetic symbols engraved on them, and the user can switch between the two sets.

The situation is different for Chinese. While there is a phonetic system called *Pinyin* promoted by authorities, there is no consensus for entering Chinese text. Some vendors use a phonetic decomposition (Pinyin or another), others use ideographic decomposition of Chinese words, with various implementations and keyboard layouts. There are about 16 known methods, none of which is a clear standard.

Also, there are actually two ideographic sets used: *Traditional Chinese* (the original written Chinese) and *Simplified Chinese*. Several years ago, the People's Republic of China launched a campaign to simplify some ideographic characters and eliminate redundancies altogether. Under the plan, characters would be streamlined every five years. Characters have been revised several times now, resulting in the smaller, simpler set that makes up Simplified Chinese.

Input Method Architecture

As shown in the previous section, there are many different input methods used today, each varying with language, culture, and history. A common feature of many input methods is that the user can type multiple keystrokes to compose a single character (or set of characters). The process of composing characters from keystrokes is called *preediting*. It may require complex algorithms and large dictionaries involving substantial computer resources.

Input methods may require one or more areas in which to show the feedback of the actual keystrokes, to show ambiguities to the user, to list dictionaries, and so on. The following are the input method areas of concern.

Status area	Intended to be a logical extension of the light-emitting diodes (LEDs) that exist on the physical keyboard. It is a window that is intended to present the internal state of the input method that is critical to the user. The status area may consist of text data and bitmaps or some combination.
preedit area	Intended to display the intermediate text for those languages that are composing prior to the client handling the data.
Auxiliary area	Used for pop-up menus and customizing dialog boxes that may be required for an input method. There may be multiple auxiliary areas for any input method. Auxiliary areas are managed by the input method independent of the client. Auxiliary areas are assumed to be a separate dialog that is maintained by the input method.

There are various user interaction styles used for preediting. The following are the preediting styles supported by Xlib.

OnTheSpot	Data is displayed directly in the application window. Application data is moved to allow preedit data to be displayed at the point of insertion.
OverTheSpot	Data is displayed in a preedit window that is placed over the point of insertion.
OffTheSpot	preedit window is displayed inside the application window but not at the point of insertion. Often, this type of window is placed at the bottom of the application window.
Root window	preedit window is the child of RootWindow.

It would require a lot of computing resources if portable applications had to include input methods for all the languages in the world. To avoid this, a goal of the Xlib design is to allow an application to communicate with an input method placed in a separate process. Such a process is called an *input server*. The server to which the application should connect is dependent on the environment when the application is started up: what the user language is and the actual encoding to be used for it. The input method connection is said to be *locale-dependent*. It is also user-dependent; for a given language, the user can choose, to some extent, the user-interface style of input method (if there are several choices).

Using an input server implies communications overhead, but applications can be migrated without relinking. Input methods can be implemented either as a token communicating to an input server or as a local library.

The abstraction used by a client to communicate with an input method is an opaque data structure represented by the XIM data type. This data structure is returned by the `XOpenIM()` function, which opens an input method on a given display. Subsequent operations on this data structure encapsulate all communication between client and input method. There is no need for an X client to use any networking library or natural language package to use an input method.

A single input server can be used for one or more languages, supporting one or more encoding schemes. But the strings returned from an input method are always encoded in the (single) locale associated with the XIM object.

Input Contexts

Xlib provides the ability to manage a multithreaded state for text input. A client may be using multiple windows, each window with multiple text entry areas, with the user possibly switching among them at any time. The abstraction for representing the state of a particular input thread is called an *input context*. The Xlib representation of an input context is an XIC. See Figure 5-1 on page 116 for an illustration.

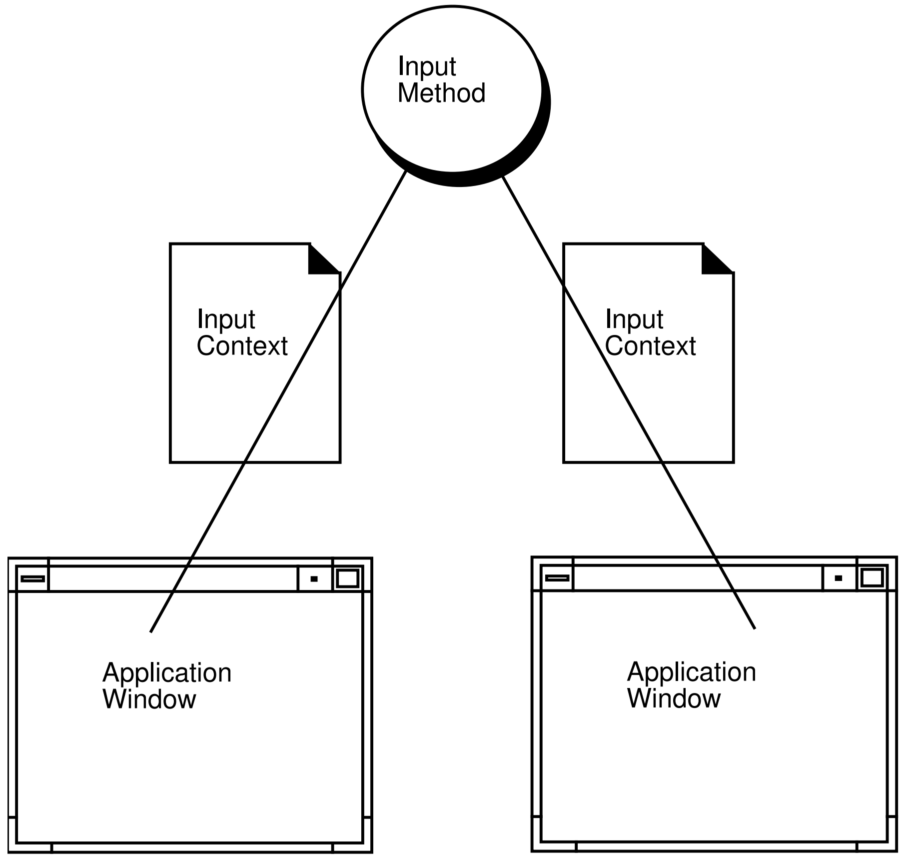


Figure 5-1 Input method and input contexts

An input context is the abstraction retaining the state, properties, and semantics of communication between a client and an input method. An input context is a combination of an input method, a locale specifying the encoding of the character strings to be returned, a client window, internal state information, and various layout or appearance characteristics. The input context concept somewhat matches for input the graphics context abstraction defined for graphics output.

One input context belongs to exactly one input method. Different input contexts can be associated with the same input method, possibly with the same client window. An XIC is created with the `XCreateIC()` function, providing an XIM argument, affiliating the input context to the input method for its lifetime. When an input method is closed with the `XCloseIM()` function, no affiliated input contexts should be used again (and should preferably be deleted before closing the input method).

Considering the example of a client window with multiple text entry areas, the application programmer can choose to implement the following:

- As many input contexts are created as text-entry areas. The client can get the input accumulated on each context each time it looks up that context.
- A single context is created for a top-level window in the application. If such a window contains several text-entry areas, each time the user moves to another text-entry area, the client has to indicate changes in the context.

Application designers can choose a range of single or multiple input contexts, according to the needs of their applications.

Keyboard Input

To obtain characters from an input method, a client must call the `XmbLookupString()` function or `XwcLookupString()` function with an input context created from that input method. Both a locale and display are bound to an input method when they are opened, and an input context inherits this locale and display. Any strings returned by the `XmbLookupString()` or `XwcLookupString()` function are encoded in that locale.

Xlib Focus Management

For each text-entry area in which the `XmbLookupString()` or `XwcLookupString()` function is used, there is an associated input context.

When the application focus moves to a text-entry area, the application must set the input context focus to the input context associated with that area. The input context focus is set by calling the `XSetICFocus()` function with the appropriate input context.

Also, when the application focus moves out of a text-entry area, the application should unset the focus for the associated input context by calling the `XUnsetICFocus()` function. As an optimization, if the `XSetICFocus()` function is called successively on two different input contexts, setting the focus on the second automatically unsets the focus on the first.

Note – To set and unset the input context focus correctly, it is necessary to track application-level focus changes. Such focus changes do not necessarily correspond to X server focus changes.

If a single input context is used to do input for multiple text-entry areas, it is also necessary to set the focus window of the input context whenever the focus window changes.

Xlib Geometry Management

In most input method architectures (OnTheSpot being the notable exception), the input method performs the display of its own data. To provide better visual locality, it is often desirable to have the input method areas embedded within a client. To do this, the client may need to allocate space for an input method. Xlib provides support that allows the client to provide the size and position of input method areas. The input method areas that are supported for geometry management are the status area and the preedit area.

The fundamental concept on which geometry management for input method windows is based is the proper division of responsibilities between the client (or toolkit) and the input method. The division of responsibilities is the following:

- The client is responsible for the geometry of the input method window.
- The input method is responsible for the contents of the input method window. It is also responsible for creating the input method window per the geometry constraints given to it by the client.

An input method can suggest a size to the client, but it cannot suggest a placement. The input method can only suggest a size: it does not determine the size, and it must accept the size it is given.

Before a client provides geometry management for an input method, it must determine if geometry management is needed. The input method indicates the need for geometry management by setting the `XIMpreeditArea()` or

XIMStatusArea() function in its XIMStyles value returned by the XGetIMValues() function. When a client decides to provide geometry management for an input method, it indicates that decision by setting the XNInputStyle value in the XIC.

After a client has established with the input method that it will do geometry management, the client must negotiate the geometry with the input method. The geometry is negotiated by the following steps:

- The client suggests an area to the input method by setting the XNAreaNeeded value for that area. If the client has no constraints for the input method, it either does not suggest an area or sets the width and height to 0 (zero). Otherwise, it sets one of the values.
- The client gets the XIC XNAreaNeeded value. The input method returns its suggested size in this value. The input method should pay attention to any constraints suggested by the client.
- The client sets the XIC XNArea value to inform the input method of the geometry of the input method's window. The client should try to honor the geometry requested by the input method. The input method must accept this geometry.

Clients performing geometry management must be aware that setting other IC values may affect the geometry desired by an input method. For example, the XNFontSet and XNLineSpacing values may change the geometry desired by the input method. It is the responsibility of the client to renegotiate the geometry of the input method window when it is needed.

In addition, a geometry management callback is provided by which an input method can initiate a geometry change.

Event Filtering

A filtering mechanism is provided to allow input methods to capture X events transparently to clients. It is expected that toolkits (or clients) using the XmbLookupString() or XwcLookupString() function call this filter at some point in the event processing mechanism to make sure that events needed by an input method can be filtered by that input method. If there is no filter, a client can receive and discard events that are necessary for the proper functioning of an input method. The following provides a few examples of such events:

- Expose events that are on a preedit window in local mode.
- Events can be used by an input method to communicate with an input server. Such input server protocol-related events have to be intercepted if the user does not want to disturb client code.
- Key events can be sent to a filter before they are bound to translations such as Xt provides.

Clients are expected to get the XIC `XNFilterEvents` value and add to the event mask for the client window with that event mask. This mask can be 0.

Callbacks

When an OnTheSpot input method is implemented, only the client can insert or delete preedit data in place and possibly scroll existing text. This means the echo of the keystrokes has to be achieved by the client itself, tightly coupled with the input method logic.

When a keystroke is entered, the client calls the `XmbLookupString()` or `XwcLookupString()` function. At this point, in the OnTheSpot case, the echo of the keystroke in the preedit has not yet been done. Before returning to the client logic that handles the input characters, the lookup function must call the echoing logic for inserting the new keystroke. If the keystrokes entered so far make up a character, the keystrokes entered need to be deleted, and the composed character is returned. The result is that, while being called by client code, input method logic has to call back to the client before it returns. The client code, that is, a callback routine, is called from the input method logic.

There are a number of cases where the input method logic has to call back the client. Each of those cases is associated with a well-defined callback action. It is possible for the client to specify, for each input context, which callback is to be called for each action.

There are also callbacks provided for feedback of status information and a callback to initiate a geometry request for an input method.

X Server Keyboard Protocol

This section discusses the server and keyboard groups.

A *keysym* is the encoding of a symbol on a keycap. The goal of the server's keysym mapping is to reflect the actual key caps on the physical keyboards. The user can redefine the keyboard by running the `xmodmap` command with the new mapping desired.

X Version 11 Release 4 (X11R4) allows for definition of a bilingual keyboard at the server. The following describes this capability.

A list of keysyms is associated with each key code. The following list discusses the set of symbols on the corresponding key:

- If the list (ignoring trailing NoSymbol entries) is a single keysym `K`, the list is treated as if it were the list `K NoSymbol K NoSymbol`.
- If the list (ignoring trailing NoSymbol entries) is a pair of keysyms `K1 K2`, the list is treated as if it were the list `K1 K2 K1 K2`.
- If the list (ignoring trailing NoSymbol entries) is three keysyms `K1 K2 K3`, the list is treated as if it were the list `K1 K2 K3 NoSymbol`.

When an explicit *void* element is desired in the list, the `VoidSymbol` value can be used.

The first four elements of the list are split into two groups of keysyms. Group 1 contains the first and second keysyms; Group 2 contains the third and fourth keysyms. Within each group, if the second element of the group is `NoSymbol`, the group is treated as if the second element were the same as the first element, except when the first element is an alphabetic keysym `K` for which both lowercase and uppercase forms are defined. In that case, the group is treated as if the first element is the lowercase form of `K` and the second element is the uppercase form of `K.`

The standard rules for obtaining a keysym from an event make use of the Group 1 and Group 2 keysyms only; no interpretation of other keysyms in the list is given here. The modifier state determines which group to use. Switching between groups is controlled by the keysym named MODE SWITCH by attaching that keysym to some key code and attaching that key code to any one of the modifiers Mod1 through Mod5. This modifier is called the *group modifier*. For any key code, Group 1 is used when the group modifier is off, and Group 2 is used when the group modifier is on.

Within a group, the keysym to use is also determined by the modifier state. The first keysym is used when the Shift and Lock modifiers are off. The second keysym is used when the Shift modifier is on, when the Lock modifier is on, and when the second keysym is uppercase alphabetic, or when the Lock modifier is on and is interpreted as ShiftLock. Otherwise, when the Lock modifier is on and is interpreted as CapsLock, the state of the Shift modifier is applied first to select a keysym; if that keysym is lowercase alphabetic, the corresponding uppercase keysym is used instead.

No spatial geometry of the symbols on the key is defined by their order in the keysym list, although a geometry might be defined on a vendor-specific basis. The server does not use the mapping between key codes and keysyms. Rather, it stores it merely for reading and writing by clients.

The KeyMask modifier named Lock is intended to be mapped to either a CapsLock or a ShiftLock key, but which one it is mapped to is left as an application-specific decision, user-specific decision, or both. However, it is suggested that users determine mapping according to the associated keysyms of the corresponding key code.

Interclient Communications Conventions for Localized Text

The following information explains how components use Interclient Communications Conventions (ICCC) to communicate text data and is offered as a guideline to understand how ICCC selections are performed. The `XmText widget`, `XmTextField` widget, and the `dtterm` command adhere to these guidelines.

The toolkit is enhanced for internationalized ICCC compliance. The selection mechanism of `XmText`, `XmTextField`, and `dtterm` is enhanced to ensure proper matching of data and data encoding in any selection transaction. This includes standard cut-and-paste operations.

For developers who use the toolkit to write their applications, the toolkit enables the application to be ICCC-compliant. However, for developers who may use another non-ICCC-compliant toolkit to develop applications that communicate with toolkit-based applications, the following may be helpful.

Owner of Selection

Any owner returns at least the following atom list when XA_TARGETS is requested on some localized text:

- Atom code set of current locale
- COMPOUND_TEXT
- XA_STRING

When XA_TEXT is requested, the owner returns its text *as is* with the encoding type of the property set to the code set of the current locale (no data conversion). An atom is created, representing the name of the code set of the locale.

When COMPOUND_TEXT is requested, the owner converts its localized text to compound text and passes it with the property type of COMPOUND_TEXT.

When XA_STRING is requested, the owner attempts to convert the localized text to XA_STRING. If the text string contains characters that cannot be converted to XA_STRING, the operation is unsuccessful.

Note – XA_STRING is defined to be ISO8859-1.

Requester of Selection

A requester first requests XA_TARGET when text data is to be communicated with the selection owner.

The requester then searches for one of the following atoms in priority order:

- Atom for the code set of the requester's locale
- COMPOUND_TEXT
- XA_STRING
- XA_TEXT

If the code set of the requester's locale matches one of the targets, the requester makes a request using the atom representing that code set. The XA_TEXT atom is used *only* if none of the other atoms is found. Because the owner returns a property with a type representing its encoding, the requester attempts to convert to the code set of its locale.

If the type COMPOUND_TEXT or XA_STRING is requested, the requester attempts to convert the text property to the code set of its current locale by using the XmbTextPropertyToTextList() or XwcTextPropertyToTextList() functions. These are used when the owner client and requester client are running under different code sets.

When converting from COMPOUND_TEXT or XA_STRING, not all text data is guaranteed to be converted; only those characters that are in common between the owner and the requester will be converted.

XmClipboard

XmClipboard is also enhanced to be ICCC-compliant in conjunction with the XmText and XmTextField widgets. When text is being put on the clipboard by way of the XmText and XmTextField widgets, the following ICCC protocol is implemented:

When text is being retrieved from the clipboard by way of the XmText and XmTextField widgets, the text from the clipboard is converted to encoding of the current locale from either COMPOUND_TEXT or XA_STRING. All text on the clipboard is assumed to be in either the compound text format or the string format.

Note – If text is put directly on the clipboard, the application needs to specify the format, or encoding type in the form of an atom, along with the text to put on the clipboard. Similarly, if text is retrieved directly from the clipboard, the retrieving application needs to check the format to see what encoding the data on the clipboard is encoded in and take the appropriate action.

Passing Window Title and Icon Name to Window Managers

The default of the XtNtitleEncoding and XtNiconNameEncoding resources for the VendorShell class is set to None. This is done only when using the Xm library. The Xt library still retains XA_STRING as the default for the resources.

This is done so that, as a default case, the XmNtitle and XmNiconName resources are converted to a standard ICCC interchange, such as compound text, based on the assumption the text (title and icon name) is localized text.

It is recommended that the user not set the `XtNtitleEncoding` and `XtNiconNameEncoding` resources. Instead, ensure that the `XtNtitle` and `XtNiconName` resources are strings encoded in the encoding of the currently active locale of the running client. If the `None` value is used, the toolkit converts the localized text to the standard ICCC style. (The encoding communicated is `COMPOUND_TEXT` or `XA_STRING`.) If the `XtNtitleEncoding` and `XtNiconNameEncoding` resources are set, the `XtNtitle` and `XtNiconName` resources are not converted in any way and are communicated to the Window Manager with the encoding specified.

Assuming the Window Manager being communicated with is ICCC-compliant, that Window Manager is able to use the encoding type of `COMPOUND_TEXT` or `XA_STRING`, or both.

When setting the `XmNdialogTitle` resource of the `XmBulletinBoard` widget class, remember that there is a restriction on the charset segment. For charsets that are *not* X Consortium-standard compound text encodings or `XmFONTLIST_DEFAULT_TAG`-associated, the text segment is treated as localized text. Localized text is converted to either compound text or ISO8859-1 before being communicated to the Window Manager.

The Window Manager is enhanced so that it always converts the client title and icon name passed from clients to the encoding of its current locale, and an `XmString` is created using the `XmFONTLIST_DEFAULT_TAG` identifier. Thus, the client title and icon name are always drawn with the default font list entry of the Window Manager font list.

Note – This allows clients running with different code sets but with similar character sets to communicate their titles to the Window Manager. For example, both a PC code client and an ISO8859-1 client can display their titles regardless of the code set of the Window Manager.

Messages

Part of internationalizing a system environment toolkit-based application is not to have any locale-specific data hardcoded within the application source. One common locale-specific item is messages (error and warning) returned by the application of the standard I/O (input/output).

In general, for any error or warning messages to be displayed to the user through a system environment toolkit widget or gadget, the messages need to be externalized through message catalogs.

For dialog messages to be displayed through a toolkit component, the messages need to be externalized through localized resource files. This is done in the same way as localizing resources, such as the `XmLabel` and `XmPushbutton` classes' `XmNlabelString` resource or window titles.

For example, if a warning message is to be displayed through an `XmMessageBox` widget class, the `XmNmessageString` resource cannot be hardcoded within the application source code. Instead, the value of this resource needs to be retrieved from a message catalog. For an internationalized application expected to run in different locales, a distinct localized catalog must exist for each of the locales to be supported. In this way, the application need not be rebuilt.

The localized resource files can be put in the `/usr/dt/app-defaults/`*%L* subdirectories or they can be pointed to by the `XENVIRONMENT` environment variable. The *%L* variable indicates the locale used at run time.

The preceding two choices are left as design decisions for the application developer.

Message Guidelines

A

Refer to the information in this appendix to write messages that are easily internationalized.

File-Naming Conventions

The conventions used in naming files with user messages are discussed here. Usually, the message source file has the suffix `.msg`; the generated message catalog has the suffix `.cat`. There may be other such files related to messages. The following criteria must be met for a file to have these suffixes:

- It is X/Open-compliant.
- It becomes a `*.cat` file through the use of the `gencat` command.

Cause and Recovery Information

Whenever possible, explain to users exactly what has happened and what they can do to remedy the situation.

The message `Bad arg` is not very helpful. However, the following message tells users exactly what to do to make the command work:

```
Do not specify more than 2 files on the command line
```

Similarly, the message `Line too long` does not giver users recovery information. However, the following message gives users more specific recovery information:

```
Line cannot exceed 20 characters
```

If detailed recovery information is necessary for a given error message, add it to the appropriate place in online information or help.

See "Sample Messages" on page 135 for samples of original and rewritten messages.

Comment Lines for Translators

A message source file should contain comments to help the translator in the process of translation. These comments will not be part of the message catalog generated. The comments are similar to C language comments to help document a program. A dollar sign ($) followed by a space will be interpreted by the translation tool and the `gencat` command as comments. The following is an example of a comment line in a message source file.

```
$ This is a comment
```

Use comment lines to tell translators and writers what variables, such as *%s*, *%c*, and *%d*, represent. For example, note whether the variable refers to such things as a user, file, directory, or flag.

Place the comment line directly beneath the message to which it refers, rather than at the bottom of the message catalog. Global comments for an entire set can be placed directly below the $set directive in the source file.

Specify in a comment line any messages within the message catalog that are obsolete.

Programming Format

For the programming format of messages, see the following list.

- Do not construct messages from clauses. Use flags or other means within the program to pass information so that a complete message can be issued at the proper time.
- Do not use hardcoded English text as a variable for a *%s* string in an existing message. This is also the construction of messages and is not translatable.
- Capitalize the first word of the sentence, and use a period at the end of the sentence or phrase.
- End the last line of the message with `\n` (backslash followed by a lowercase n, indicating a new line). This also applies to one-line messages.
- Begin the second and remaining lines of a message with `\t` (backslash followed by a lowercase `t`, indicating a tab).
- End all other lines with `\n\` (backslash followed by a lowercase n, followed by another backslash, indicating a new line).
- If, for some reason, the message should not end with a new line, use a comment to tell the writers.
- Precede each message with the name of the command that called the message, followed by a colon. The command name should precede the component number in error messages. The command name is shown in the following example as it should appear in a message:

  ```
  OPIE "foo: Opening the file."
  ```

Writing Style

The following guidelines on the writing style of messages include terminology, punctuation, mood, voice, tense, capitalization, and other usage questions.

- Use sentence format. One-line and one-sentence messages are preferable.
- Add articles (*a*, *an*, *the*) when necessary to eliminate ambiguity.
- Capitalize the first word of the sentence and use a period at the end.

- Use the present tense. Do not allow future tense in a message. For example, use the sentence:

  ```
  The foo command displays a calendar.
  ```

  Instead of:

  ```
  The foo command will display a calendar.
  ```

- Do not use the first person (*I* or *we*) anywhere in messages.
- Avoid using the second person.

  Do not use the word *you* except in help and interactive text.

- Use active voice. The first line is the original message. The second line is the preferred wording.

  ```
  MYNUM "Month and year must be entered as numbers."
  MYNUM "foo: 7777-222 Enter month and year as numbers.\n"
  ```

  7777-222 is the message ID.

- Use the imperative mood (command phrase) and active verbs: *specify, use*, *check*, *choose*, and *wait* are examples.
- State messages in a positive tone. The first line is the original message. The second line is the preferred wording.

  ```
  BADL "Don't use the f option more than once."
  BADL "foo: 7777-009 Use the -f flag only once.\n"
  ```

- Do not use nouns as verbs. Use words only in the grammatical categories shown in the dictionary. If a word is shown only as a noun, do not use it as a verb. For example, do not *solution* a problem (or, for that matter, *architect* a system).
- Do not use prefixes or suffixes. Translators may not understand words beginning with *re-*, *un-*, *in-*, or *non-*, and the translations of messages that use these prefixes or suffixes may not have the meaning you intended. Exceptions to this rule occur when the prefix is an integral part of a commonly used word. The words *previous* and *premature* are acceptable; the word *nonexistent,* is not.

- Do not use plurals. Do not use parentheses to show singular or plural, as in *error(s)*, which cannot be translated. If you must show singular and plural, write *error or errors*. A better way is to condition the code so that two different messages are issued depending on whether the singular or plural of a word is required.
- Do not use contractions. Use the single word *cannot* to denote something the system is unable to do.
- Do not use quotation marks. This includes both single and double quotation marks. For example, do not use quotation marks around variables such as *%s*, *%c*, and *%d* or around commands. Users may take the quotation marks literally.
- Do not hyphenate words at the end of lines.
- Do not use the standard highlighting guidelines in messages, and do not substitute initial or all caps for other highlighting practices.
- Do not use *and/or*. This construction does not exist in other languages. Usually it is better to say *or* to indicate that it is not necessary to do both.
- Use the 24-hour clock. Do not use *a.m.* or *p.m.* to specify time. For example, write *1:00 p.m.* as *1300*.
- Avoid acronyms. Only use acronyms that are better known to your audience than their spelled-out versions. To make a plural of an acronym, add a lowercase *s*, without an apostrophe. Verify that it is not a trademark before using it.
- Avoid the "no-no" words. Examples are *abort*, *argument*, and *execute*. See the project glossary.
- Retain meaningful terminology. Keep as much of the original message text as possible while ensuring that the message is meaningful and translatable.

Usage Statements

The usage statement is generated by commands when at least one flag that is not valid has been included in the command line. The usage statement must not be used if only the data associated with a flag is missing or incorrect. If this occurs, an error message unique to the problem is used.

- Show the command syntax in the usage statement. For example, a possible usage statement for the `del` command reads:

```
Usage: del {File ...|-}
```

- Clauses defining the purpose of a command are to be removed.
- Capitalize the first letter of such words (parameters) as *File, Directory, String, Number,* and so on only when used in a usage statement.
- Do not abbreviate parameters on the command line. It may be perfectly obvious to experienced users that *Num* means *Number*, but spell it out to ensure correct translation.
- Use only the following delimiters in usage statements:

Delimiter	**Description**
[]	Parameter is optional.
{ }	There is more than one parameter choice, but one of the parameters is required. (See the following text.)
|	Choose one parameter only. [a|b] indicates that you can choose *a* or *b* or neither *a* nor *b*. {a|b} indicates that you must choose either *a* or *b*.
..	Parameter can be repeated on the command line. (Note that there is a space before the ellipsis.)
-	Standard input.

- A usage statement parameter does not require square brackets or braces if it is required and is the only choice, as in the following:

```
banner String
```

- In usage statements, put a space between flags that must be separated on the command line. For example:

```
unget [-n] [-rSID] [-s] {File|-}
```

- If flags can be used together without a separating space, do not separate them with a space on the command line. For example:

```
wc [-cwl] {File ...|-}
```

- When the order of flags on the command line does not make a difference, put them in alphabetical order. If the case is mixed, put lowercase versions first:

```
get -aAijlmM
```

- Some usage statements can be long and involved. Use your best judgment to determine where you should end lines in the usage statement. The following example shows an old-style usage statement for the `get` command:

```
Usage: get [-e|-k] [-cCutoff] [-iList] [-rSID] [-wString] [xList]
[-b] [-gmnpst] [-l[p]] File ...
Retrieves a specified version of a Source Code Control System
(SCCS) file.
```

Standard Messages

Certain commands have standard errors defined in POSIX.2 documentation. Follow the guidelines set up in POSIX.2, if applicable.

- Tell the user to `Press the ------ key` to select a key on the keyboard, including the specific key to press (such as, `Press Ctrl-D`).
- Unless the system is overloaded, there is no need to tell the user to `Try again later`. That should be obvious from the message.
- When writing message text, use the word *parameter* to describe text on the command line; use the word *value* to indicate numeric data.
- Use the word *flag* rather than the words *command option.*
- Do not use commas to set off the one-thousandth place in values.
- Do not use 1,000. Use 1000.
- If a message must be set off with an asterisk, use two asterisks at the beginning of the message and two asterisks at the end of the message.

```
** Total **
```

- Use *log in* and *log off* as verbs.

```
Log in to the system; enter the data; then log off.
```

- Use *user name*, *group name*, and *login* as nouns.

```
The user name is sam.
The group name is staff.
The login directory is /u/sam.
```

- User number and group number refer to the number associated with the user's name and group.
- Do not use the term *superuser*. The *root user* may not have all privileges.

- Use the words *command string* to describe the command with its parameters.
- Many of the same messages occur frequently. Table A-1 lists the new standard message that replaces the old message.

Table A-1 New Standard Messages

Use the Following Standard Messages	Instead of These Messages
`Cannot find or open the file.`	`Can't open filename.`
`Cannot find or access the file.`	`Can't access`
`The syntax of a parameter is not valid.`	`syntax error`

Regular Expression Standard Messages

Table A-2 lists the standard regular expression error messages, including the message number associated with each regular expression error:

Table A-2 Regular Expression Standard Messages

Number	Use These Standard Messages	Instead of These Messages
11	`Specify a range end point that is less than 256.`	`Range end point too large.`
16	`The character or characters between \{ and \} must be numeric.`	`Bad number.`
25	`Specify a \digit between 1 and 9 that is not greater than the number of subpatterns.`	`\digit out of range.`
36	`A delimiter is not correct or is missing.`	`Illegal or missing delimiter.`
41	`There is no remembered search string.`	`No remembered search string.`
42	`There is a missing \( or \).`	`\(\) imbalance.`
43	`Do not use \( more than 9 times.`	`Too many \(.`
44	`Do not specify more than 2 numbers between \{ and \}.`	`More than two numbers given in \{ and \}.`

Table A-2 Regular Expression Standard Messages *(Continued)*

45	An opening \{ must have a closing \}.	} expected after \.
46	The first number cannot exceed the second number between \{ and \}.	First number exceeds second in \{ and \}.
48	Specify a valid end point to the range.	Invalid end point in range expression.
49	For each [there must be a].	[ ] imbalance.
50	The regular expression is too large for internal memory storage. Simplify the regular expression.	Regular expression overflow.

Sample Messages

These are examples of original messages and rewritten messages. The rewritten message follows each original message.

```
AFLGKEYLTRS "Too Many -a Keyletters (Ad9)"
AFLGKEYLTRS "foo: 7777-007 Use the -a flag less than 11 times.\n"

FLGTWICE "Flag %c Twice  (Ad4)"
FLGTWICE "foo: 7777-004 Use the %c header flag once.\n"

ESTAT "can't access %s.\n"
ESTAT "foo: 7777-031 Cannot find or access %s.\n"

EMODE "foo: invalid mode\n"
EMODE "foo: 7777-033 A mode flag or value is not correct.\n"

DNORG "-d has no argument  (ad1)"
DNORG "foo: 7777-001 Specify a parameter after the -d flag.\n"

FLOORRNG "floor out of range  (ad23)"
FLOORRNG "foo: 7777-021 Specify a floor value greater than 0\n\
\tand less than 10000.\n"

AFLGARG "bad -a argument  (ad8)"
AFLGARG "foo: 7777-006 Specify a user name, group name, or\n\
\tgroup number after the -a flag.\n"

BADLISTFMT "bad list format  (ad27)"
BADLISTFMT "foo: 7777-025 Use numeric version and release\
\tnumbers.\n"
```

Index

G

H

I

W

X