Chapter 6

Internationalizing Your Application

Internationalization is the process of generalizing an application so that it can easily be customized—or *localized*—to run in more than one language environment. You can provide internationalized software that will produce output in a user's native language, format data (such as currency values and dates) according to local standards, and tailor software to a specific culture.

This chapter describes how to create such an application. It contains the following major sections:

- "Overview" presents an introduction to internationalization and defines some common terms.
- "Additional Reading on Internationalization" explains how to set the current locale and limitations of the locale system.
- "Character Sets, Codesets, and Encodings" describes various ways of encoding characters, the traditional ASCII being just one of these.
- "Cultural Items" discusses the ways in which different cultures affect the way a string can be viewed, for example in outputting or collating.
- "Locale-Specific Behavior" covers native language support (NLS) and the NLS database, regular expressions, and cultural data.
- "Strings and Message Catalogs"describes how to create and use catalogs of messages to send diagnostic information to users in various locales.
- "Internationalization Support in X11R6" describes internationalization support provided by X11, Release 6 (including features from X11R5).
- "Internationalization Support in Motif" points to information describing how to internationalize a Motif application.
- "User Input" discusses the translation of keyboard events into programmatic character strings for a variety of keyboards.

- "GUI Concerns" discusses internationalizing applications that use graphical user interfaces (GUIs)
- "Popular Encodings" presents some common non-ASCII encodings.

For a list of ISO 3166 country names and abbreviations, see Appendix A, "ISO 3166 Country Names and Abbreviations." You can find detailed information about fonts in Chapter 5, "Working With Fonts." Also, you can find additional information about internationalizing an application in the *Indigo Magic Desktop Integration Guide*.

Overview

Internationalized software can be made to produce output in a user's native language, to format data (such as dates and currency values) according to the user's local customs, and to otherwise make the software easier to use for users from a culture other than that of the original software developer. As computers become more widely used in non-American cultures, it becomes increasingly important that developers stop relying on the conventions of American programming and the English language in their programs. This chapter provides information on how to make your applications more widely accessible.

This section presents the following topics:

- "Some Definitions" covers locales, internationalization, localization, nationalized software, and multilingual software.
- "Areas of Concern in Internationalizing Software" points out a few concerns to watch for when internationalizing your software.
- "Standards" covers standard-compliant features.
- "Internationalizing Your Application: The Basic Steps" lists the procedures to use when internationalizing an icon.
- "Additional Reading on Internationalization" provides references you can consult for additional information about internationalization.

Some Definitions

This section defines some of the terms used in this chapter.

Locale

Locale refers to a set of local customs that determine many aspects of software input and output formatting, including natural language, culture, character sets and encodings, and formatting and sorting rules. The locale of a program is the set of such parameters that are currently selected. For information on the method for selecting locales, see "Additional Reading on Internationalization" below.

Internationalization (i18n)

Internationalization is the process of making a program capable of running in multiple locales without recompiling. To put it another way, an internationalized program is one that can be easily localized without changing the program itself. (See "Localization (l10n)," below, for an explanation of the term "localization.")

Note: The word "internationalization" consists of an *i* followed by 18 letters followed by an *n*. It is thus often abbreviated "i18n" in informal writing. On similar principles, "localization" is often abbreviated "l10n."

A program written for a specific locale may be difficult to run in a different environment. Rewriting such a program to operate in each desired environment would be tedious and costly.

Your goal as a developer should thus be to write *locale-independent* programs, programs that make no assumptions about languages, local customs, or coded character sets. Such internationalized applications can run in a user's native environment following native conventions with native messages, without recompiling or relinking. A single copy of an internationalized program can be used by a world of different users.

Localization (l10n)

Localization is the act of providing an internationalized application with the environment and data it needs to operate in a particular locale. For example, adding German system messages to IRIX is a part of localizing IRIX for the German locale.

Nationalized Software

Nationalized programs run in only one language and are governed by one set of customs; in other words, in a nationalized program the locale is built into the application. Even if the application doesn't use ASCII or English, as long as it is a single-language program it is nationalized, not internationalized. Most older UNIX programs can be thought of as being nationalized for the United States.

Consider two applications, *hello* and *bonjour*. The application *hello* always produces the output

```
Hello, world.
```

and *bonjour* always produces

```
Bonjour, tout le monde.
```

Neither *hello* nor *bonjour* are internationalized; they are both nationalized.

There are no special requirements for writing or porting nationalized applications, whether they are text or graphics programs. Terminal-based programs work on suitable terminals, including internationalized terminal emulators. "Suitable" means that the terminal supports any necessary fonts and understands the encoding of the application output. Graphics programs simply do as they have always done. Applications using existing interfaces to operate in non-English or non-ASCII environments should continue to compile and run under an internationalized operating system.

Multilingual Software

A *multilingual* program is one that uses several different locales at the same time. Examples are described in "Multilingual Support" on page 203.

Areas of Concern in Internationalizing Software

Few developers will have to pay attention to more than a few items described in this section. Most will need to catalog their strings. Some will need to use library routines for character sorting or locale-dependent date, time, or number formatting. A few whose applications use the eighth bit of 8-bit characters inappropriately will need to stop doing so. The few applications that do arithmetic to manipulate characters will need to be cleaned up. Some GUI designers will have to spend just a little more time thinking. But for the large majority of developers, there isn't much to do.

The information presented in the following sections addresses internationalization issues pertinent to a developer; some sections, however, may not be relevant to your applications.

Standards

IRIX internationalization includes these standards-compliant features, among others:

- ANSI C and POSIX (ISO 9945-1): Locale
- *X/OPEN Portability Guide, Issue 4* (XPG/4): XPG/4 message catalogs, interpretation of locale strings
- UNIX™ System V Release 4: Multi-National Language Support (MNLS) message catalogs
- X11R5 and X11R6: Input methods, text rendering, resource files

Internationalizing Your Application: The Basic Steps

To internationalize your icon, follow these steps:

1. Call **setlocale()** as soon as possible to put the process into the desired locale. See "Setting the Current Locale" on page 198 for instructions.
2. Make your application 8-bit clean. (An application is 8-bit clean if it does not use the high bit of any data byte to convey special information.) See "Eight-Bit Cleanliness" on page 205 for instructions.
3. If you're writing a multilingual application, you must do one of two things:
   - fork, and then call **setlocale()** differently in each process
   - call **setlocale()** repeatedly as necessary to change from language to language

   See "Multilingual Support" on page 203 for more information.
4. Use wide character (WC) or multibyte (MB) characters and strings to allow for more than one byte per character (this is needed for Asian languages, which often require two or even four bytes per character). See "Character Representation" on page 206 for more information.

5. Do not rely on ASCII and English sorting rules. Locale-specific collation should be performed with **strcoll()** and **strxfm()**. (These are table-driven functions; the tables are supplied as part of locale support.) See “Collating Strings” on page 211 for more information.
6. Use the **localeconv()** function to find out about general details of numeric formatting. Use **strfmon()** to format currency amounts in particular. See “Specifying Numbers and Money” on page 213 for more information.
7. Use **strftime()** to format dates and times (**strftime()** gives a host of options for displaying locale-specific dates and times.) See “Formatting Dates and Times” on page 214 for more information.
8. Avoid arithmetic on character values. Use the macros in *ctype.h* to get information about a given character. (These macros are table-driven and locale-sensitive.) If you prefer, you can use the functions that correspond to these macros instead. “Character Classification and ctype” on page 215 provides more detailed information on these macros and functions.
9. If you do your own regular expression parsing and matching, use the XPG/4 extensions to traditional regular expression syntax for internationalized software. See “Regular Expressions” on page 216 for more information.
10. Where possible, use the XPG/4, rather than the MNLS interface in order to maximize portability. See “Strings and Message Catalogs” on page 229 for more information.
11. Provide a catalog for your locale. See “SVR4 MNLS Message Catalogs” on page 233 for more information.
12. The File Typing Rule (FTR) strings that are used to customize the Indigo Magic desktop can be Internationalized. See “Internationalizing File Typing Rule Strings With MNLS” on page 237 for more information.
13. Use message catalogs for **printf()** format strings that take linguistic parameters, and allow localizers to localize the format strings as well as text strings. See “Variably Ordered Referencing of printf() Arguments” on page 238 for more information.
14. If you’re using Xlib, initialize Xlib’s internationalization state after calling **setlocale()**. See “Initialization for Xlib Programming” on page 242 for more information.
15. Specify a default fontset suitable for the default locale. Make sure that the application accepts localized fontset specifications via resources (or message catalogs) or command-line options. See “Fontsets” on page 243 for more information.

16. Use X11R5 and X11R6 text rendering routines that understand multibyte and wide character strings, not the X11R4 text rendering routines **XDrawText()**, **XDrawString()**, and **XDrawImageString()**. See "Text Rendering Routines" on page 245 for more information.
17. Use X11R5 and X11R6 MB and WC versions of width and extents interrogation routines. See "New Text Extents Functions" on page 245 for more information.
18. If you are writing a toolkit text object, or if you can't use a toolkit to manage event processing for you, then you have to deal with input methods. Follow the instructions in "User Input" on page 248.
19. Use resources to label any object that employs some sort of text label. Your application's app-defaults file should specify every reasonable string resource. See "X Resources for Strings" on page 265 for more information.
20. Use dynamic layout objects that calculate layout depending on the natural (localized) size of the objects involved. Some IRIS IM widgets providing these services are XmForm, XmPanedWindow, and XmRowColumn. See "Dynamic Layout" on page 266 for more information. If you can't use dynamic layout objects, refer to "Layout" on page 266 for instructions.
21. Make sure that all icons and other pictographic representations used by your application are localizable. See "Icons" on page 267 for more information.

Additional Reading on Internationalization

For more information on internationalization, refer to:

- O'Reilly Volume 1, *Xlib Programming Manual*
- *X Window System*, by Robert Scheifler and Jim Gettys
- *X/Open Portability Guide*
- *OSF/Motif Style Guide*

Locales

An internationalized system is capable of presenting and receiving data understandably in a number of different formats, cultures, languages, and character sets. An application running in an internationalized system must indicate how it wants the system to behave. IRIX uses the concept of a locale to convey that information.

A process can have only one locale at a time. Most internationalization interfaces rely on the locale of the current process being set properly; the locale governs the behavior of certain library routines.

This section covers the following topics:

- "Setting the Current Locale" explains categories, locales, strings, location of locale-specific data, and locale naming conventions.
- "Limitations of the Locale System" describes multilingual support, misuses of locales, and encoding.

You can find additional information in "Locale-Specific Behavior" on page 216, which describes native language support, regular expressions, and cultural data.

Setting the Current Locale

Applications begin in the *C* locale. (*C* is the name used to indicate the system default locale; it usually corresponds to American English.) Applications should therefore call **setlocale()** as soon as possible to put the process into the desired locale. The syntax for **setlocale()** is:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

The call almost always looks either like this:

```
if (setlocale(LC_ALL, "") == NULL)
    exit_with_error();
```

or like this:

```
if (setlocale(LC_ALL, "") == NULL)
    setlocale(LC_ALL, "C");
```

Details of the two parameters are given in the next two sections.

Category

Applications need not perform every aspect of their work in the same locale. Although this approach is not recommended, an application could (for example) perform most of its activities in the English locale but use French sorting rules. You can use locale categories to do this kind of locale-mixing. (Mixing locale categories is not the same as multilingual support—see "Multilingual Support.")

The *category* argument is a symbolic constant that tells **setlocale()** which items in a locale to change. Table 6-1 lists the available category choices.

Table 6-1 Locale Categories

Category	Affects
LC_ALL	All categories below
LC_COLLATE	Regular expressions, **strcoll()**, and **strxfrm()**
LC_CTYPE	Regular expressions and ctype routines (such as **islower()**)
LC_MESSAGES[a]	**gettxt()**, **pfmt()**, and **nl_langinfo()**
LC_MONETARY	**localeconv()** and **strfomon()**
LC_NUMERIC	Decimal-point character for formatted I/O and nonmonetary formatting information returned by **localeconv()**
LC_TIME	**ascftime()**, **cftime()**, **getdate()**, and **strftime()**

a. LC_MESSAGES is supported by SVR4 but isn't required by XPG/4.

Categories correspond to databases that contain relevant information for each defined locale. The locations of these databases are given in the "Location of Locale-Specific Data" on page 201.

Locale

The **setlocale()** function attempts to set the locale of the specified category to the specified locale. You should almost always pass the empty string as the *locale* parameter to conform to user preferences.

On success, **setlocale()** returns the new value of the category. If **setlocale()** couldn't set the category to the value requested, it returns NULL and does not change locale.

The Empty String

An empty string passed as the *locale* parameter is special. It specifies that the locale should be chosen based on environment variables. This is the way a user specifies a preferred locale, and that preference should almost always be honored. The variables are checked hierarchically, depending on category, as shown in Table 6-2; for instance, if the category is LC_COLLATE, an empty-string locale parameter indicates that the locale should be chosen based on the value of the environment variable LC_COLLATE—or, if that value is undefined, the value of the environment variable LANG, which should contain the name of the locale that the user wishes to work in.

Table 6-2 Category Environment Variables

Category	First Environment Variable	Second Environment Variable
LC_COLLATE	LC_COLLATE	LANG
LC_CTYPE	LC_CTYPE	LANG
LC_MESSAGES	LC_MESSAGES	LANG
LC_MONETARY	LC_MONETARY	LANG
LC_NUMERIC	LC_NUMERIC	LANG
LC_TIME	LC_TIME	LANG

Specifying the category LC_ALL attempts to set each category individually to the value of the appropriate environment variable.

If no non-null environment variable is available, **setlocale()** returns the name of the current locale.

Nonempty Strings in Calls to setlocale()

Here are the possibilities for specifying the *locale* parameter:

NULL	Specifying a null pointer argument—not the same as the empty string—causes **setlocale()** to return the name of the current locale.
"C"	Specifying a locale value of the single-character string "C" requests whatever locale the system uses as a default. (Note that this is a string and not just a character.)
Other strings	Request a particular locale by specifying its name. This overrides any user preferences and should only be done with good reason.

Location of Locale-Specific Data

Except for XPG/4 message catalogs, locale-specific data (that is, the "compiled" files containing the collation information, monetary information, and so on) are located in */usr/lib/locale/*<locale>/<category>, where <locale> and <category> are the names of the locale and category, respectively. For example, the database for the LC_COLLATE category of the French locale *fr* would be in */usr/lib/locale/fr/LC_COLLATE*.

There will probably be multiple locales symbolically linked to each other, usually in cases where a specific locale name points to the more general case. For example, */usr/lib/locale/En_US.ascii* might point to */usr/lib/locale/C*.

Locale Naming Conventions

A locale string is of the form

language`[`*_territory*`[.`*encoding*`]][@`*modifier*`]...`

where

- *language* is the two-letter ISO 639 abbreviation for the language name.
- *territory* is the two-uppercase-letter ISO 3166 abbreviation for the territory name. (For a list of these abbreviations, see the table in Appendix A, "ISO 3166 Country Names and Abbreviations.")

- *encoding* is the name of the character encoding (mapping between numbers and characters). For western languages, this is typically the codeset, such as 8859-1 or ASCII. For Asian languages, where an encoding may encode multiple codesets, the encodings themselves have names, such as UJIS or EUC (these encodings are described later in this section). "Character Sets, Codesets, and Encodings" on page 204 discusses codesets and encodings.
- *modifier*s are not actually part of the locale name definition; they give more specific information about the desired localized behavior of an application. For example, under X11R5 or X11R6, a user can select an input method with modifiers. (To use the *xwnmo* Input Method server provided by Silicon Graphics, for example, add **@im=_XWNMO** to the locale string.) No standards exist for this part of a locale string.

Language data is implementation specific; databases for the language *en* (English) might contain British cultural data in England and American cultural data in the United States. If other than the default settings are required, the territory field may be used. For example, the above cases could be more strictly defined by setting LANG to *en_EN* or *en_US*. Full rigor might lead to *en_EN.88591* for England (the locale encoding specification for ISO 8859-1 is "88591") and *en_US.ascii* for the USA.

ANSI C has defined a special locale value of *C*. The *C* locale is guaranteed to work on all compliant systems and provides the user with the system's default locale. This default is typically American English and ASCII, but need not be. POSIX has also defined a special locale value, *POSIX*, which is identical to the *C* locale.

The length of the locale string may not exceed NL_LANGMAX characters (NL_LANGMAX is defined in */usr/include/limits.h*). However, XPG/4 recommends that this string (not counting modifiers) not exceed 14 characters.

Limitations of the Locale System

This section explains multilingual support, misuse of locales, and the absence of filesystem information for encoding types.

Multilingual Support

There can be only one locale at a time associated with any given process in an internationalized system. Therefore, although multilingual applications—which give the appearance of using more than one locale at a time—can be created, internationalization does not provide inherent support for them. Here are two examples of multilingual programs:

- An application creates and maintains windows on four different displays, operated by four different users. The program has a single controlling process, which is associated with only one locale at any given time. However, the application can switch back and forth between locales as it switches between users, so the four users may each use a different locale.
- In a sophisticated editing system with a complex user interface, a user may wish to operate the interface in one language while entering or editing text in another. For instance, a user whose first language is German may wish to compose a Japanese document, using Japanese input and text manipulation, but with the user interface operating in German. (There is no standard interface for such behavior.)

In writing a multilingual application, the first task is identifying the locales for the program to run in and when they apply. (There is no standard method for performing this task.) Once the application has chosen the desired locales, it must do one of the following:

- fork, and then call **setlocale()** differently in each process
- call **setlocale()** repeatedly as necessary to change from language to language

Misuse of Locales

The LANG environment variable and the locale variables provide the freedom to configure a locale, but they do not protect the user from creating a nonsensical combination of settings. For example, you are allowed to set LANG to *fr* (French) and LC_COLLATE to *ja_JP.EUC* (Japanese). In such a case, string routines would assume text encoded in 8859-1—except for the sorting routines, which might assume French text and Japanese sorting rules. This would likely result in arbitrary-seeming behavior.

No Filesystem Information for Encoding Types

The IRIX filesystem does not contain information about what encoding should be associated with any given data. Thus, applications must assume that data presented to an application in some locale is properly encoded for that locale. In other words, a file is interpreted differently depending on locale; there is no way to ask the file what it thinks its encoding is.

For example, you may have created a file while in a Japanese locale using EUC. Later, you might try printing it while in a French locale. The results will likely resemble a random collection of Latin 1 characters.

This problem applies to almost all stored strings. Most strings are uninterpreted sequences of nonzero bytes. This includes, for example, filenames. You can, if you want to, name your files using Chinese characters in a Chinese locale, but the names will look odd to anyone who runs */bin/ls* on the same filesystem using a non-Chinese locale.

Character Sets, Codesets, and Encodings

One major difference between nationalized and internationalized software is the availability in internationalized software of a wide variety of methods for encoding characters. Developers of internationalized software no longer have the convenience of always being able to assume ASCII. Three terms that describe groupings of characters are the following:

character set An abstract collection of characters.

codeset A character set with exactly one associated numerical encoding for each character. The English alphabet is a character set; ASCII is a codeset.

encoding A set of characters and associated numbers; however, this term is more general than "codeset." A single encoding may include multiple codesets; *Extended UNIX Code (EUC*), for instance, is an encoding that provides for four codesets in one data stream.

This section describes these topics:

- "Eight-Bit Cleanliness" explains how to make 8-bit clean characters.
- "Character Representation" discuses multibyte and wide characters.
- "Multibyte Characters" covers using and handling multibyte characters, conversions to constant-size characters, and the number of bytes in a character and string.
- "Wide Characters" explains *wchar* strings, support routines, and conversion to multibyte characters.
- "Reading Input Data" covers nonuser-originated data.

For information on installing and using fonts with an application, refer to Chapter 5, "Working With Fonts."

Eight-Bit Cleanliness

A program is *8-bit clean* if it does not use the high bit of any data byte to convey special information. ASCII characters are specified by the low seven bits of a byte, so some programs use the high bit of a data byte as a flag; such programs are not 8-bit clean. Internationalized programs must be 8-bit clean, because they cannot expect data to be in the form of ASCII bytes; non-ASCII character sets usually use all eight bits of each byte to specify the character. But a program must go out of its way to manipulate bytes based on the value of the high bit, and since changing data without cause is seldom desirable, most programs are already 8-bit clean.

The old *csh* (before this problem was fixed in the IRIX 5.0 release) was a good example of a program that was not 8-bit clean; it used the high bit in input strings to distinguish aliases from unaliased commands. An effect of this misuse was that *csh* stripped the eighth bit from all characters. For example, the user command

```
echo I know an architect named Mañosa
```

Produced the response

```
I know an architect named Maqosa
```

Another example is the specification of Internet messages, which calls for 7-bit data. Thus, if *sendmail* fails to strip the 8th bit from a character prior to sending it, it violates a protocol; if it does strip the bit, it could garble a non-ASCII message (this protocol problem is being addressed).

One of the simplest things to do to remove the American bias from a program is to replace the ASCII assumption with the assumption that the Latin 1 codeset will be used. This approach is not true internationalization, but it can make the application usable in most of Western Europe. Latin 1 uses only one byte per character, unlike some other codesets, so 8-bit clean ASCII software should work without modification using the Latin 1 codeset.

Ensuring that code is 8-bit clean is the single most important aspect of internationalizing software.

Another caveat about 8-bit characters applies only to a particular set of circumstances: If you are not using a multibyte character type (see the next section), you should not declare characters as type *signed char*. (The default in IRIX C is for *char* to imply *unsigned char*.) If you try to cast a *signed char* to an *int* (as you must do to use the **ctype()** functions) and the character's high bit is set (as it may be in an 8-bit character set), the high bit is interpreted as a sign bit and extends into the full width of the int.

Character Representation

Western languages usually require only one byte for each character. Asian languages, however, often require two or even four bytes per character, and some Asian encodings allow a variable number of bytes per character.

The two kinds of encodings that allow more than one byte per character are

- multibyte (MB) characters are of variable size
- wide characters (WC or wchar characters) are a fixed number of bytes long)

The application developer must decide where to use WC and MB characters and strings:

- Multibyte strings are almost the default: string I/O uses MB, MB code works for ASCII and ISO 8859, and MB characters use less space than do wide characters. However, manipulating individual characters within a multibyte string is difficult.

  Note: Traditional strings are merely a special case of multibyte strings, where every character happens to be one byte long and there is only one codeset. All MB code, including conversion to and from wchars, works for traditional ASCII, or ISO 8859, strings.

- Applications that do heavy string manipulation typically use WC strings for such activity, because manipulating individual WC characters in a string is much simpler than doing the same thing with MB characters. So wide characters are used as necessary to provide programming ease or runtime speed; however, they take up more space than MB characters.

  Note: WC is system dependent—applications should not use it for I/O strings or communication.

Multibyte Characters

A multibyte character is a series of bytes. The character itself contains information on how many bytes long it is. Multibyte characters are referenced as strings (and are therefore of type *char* *); before parsing, a string is indistinguishable from a multibyte character. The zero byte is still used as a string (and MB character) terminator.

A string of MB characters can be considered a null-terminated array of bytes, exactly like a traditional string. A multibyte string may contain characters from multiple codesets. Usually, this is done by incorporating special bytes that indicate that the next character (and only the next character) will be in a different codeset. Very little application code should ever need to be aware of that, though; you should use the available library routines to find out information about multibyte strings rather than look at the underlying byte structure, because that structure varies from one encoding to another. For one example of an encoding that allows characters from multiple codesets, see "EUC" on page 271.

Use of Multibyte Strings

Multibyte strings are very easy to pass around. They efficiently use space (both data and disk space), since "extra" bytes are used only for characters that require them. MB strings can be read and written without regard to their contents, as long as the strings remain intact. Displaying MB strings on a terminal is done with the usual routines: **printf()**, **puts()**, and so on. Many programs (such as *cat*) need never concern themselves with the multibyte nature of MB strings, since they operate on bytes rather than on characters; so MB strings are often used for string I/O.

Manipulation of individual characters in an MB string can be difficult, since finding a particular character or position in a string is nontrivial (see "Handling Multibyte Characters," below). Therefore, it is common to convert to WC strings for that kind of work.

Handling Multibyte Characters

Usually, multibyte characters are handled just like *char* strings. Editing such strings, however, requires some care.

You cannot tell how many bytes are in a particular character until you look at the character. You cannot look at the *n*th character in a string without looking at all the previous *n* - 1 characters, because you cannot tell where a character starts without knowing where the previous character ends. Given a byte, you don't know its position within a character. Thus, we say the string has *state* or is *context-sensitive*; that is, the interpretation we assign to any given byte depends on where we are in a character.

This analysis of characters is locale-dependent, and therefore must be done by routines that understand locale.

Conversion to Constant-Size Characters

Multibyte characters and strings are convertible to wchars using **mbtowc()** for individual characters and **mbstowcs()** for strings (see the mbtowc(3) and mbstowcs() reference pages).

How Many Bytes in a Character?

To find out how many bytes make up a given single MB character, use **mblen()**, as shown in Example 6-1 (see also the mblen(3) reference page).

Example 6-1 Find Number of Bytes in an MB Character

```
#include <stdlib.h>
. . .
size_t n;
int len;
char *pStr;
. . .
len = mblen(pStr, n); /* examine no more than n bytes */
```

It is the application's responsibility to ensure that *pStr* points to the beginning of a character, not to the middle of a character.

The maximum number of bytes in a multibyte character is MB_LEN_MAX, which is defined in *limits.h*. The maximum number of bytes in a character under the current locale is given by the macro MB_CUR_MAX, defined in *stdlib.h*.

How Many Bytes in an MB String?

Since **strlen()** simply counts bytes before the first NULL, it tells you how many bytes are in an MB string.

How Many Characters in an MB String?

When **mbstowcs()** converts MB strings to WC strings, it returns the number of characters converted. This is the simplest way to count characters in an MB string.

Note: Many code segments that deal with individual characters within a string are better served by wide character strings. Since counting often involves conversion, such segments are often better served by working with a WC string, then converting back.

Getting the length without performing the conversion is straightforward, but not as simple. **mbtowc()** converts one character and returns the number of bytes used, but returns the same information without conversion if a NULL is passed as the address of the WC destination. Thus

```
len = mblen(pStr, n);
```

is equivalent to

```
len = mbtowc((wchar_t *) NULL, pStr, n);
```

In fact, **mblen()** calls **mbtowc()** to perform its count. Therefore, counting characters in an MB string without converting would look like the code in Example 6-2.

Example 6-2 Counting MB Characters Without Conversion

```
int cLen;
char *tStr = pStr;

numChars = 0;
cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
while (cLen > 0) {
    tStr += cLen;
    numChars++;
    cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
    if (cLen == -1)
        numChars = cLen; /* invalid MB character */
}
```

Wide Characters

A wide character (*WC* or *wchar*) is a data object of type *wchar_t*, which is guaranteed to be able to hold the system's largest numerical code for a character. *wchar_t* is defined in *stdlib.h*. Under IRIX 4.0.x, **sizeof(wchar_t)** was 1. In IRIX 5.1 and above, it is 4. All wchars on a system are the same size, independent of locale, encoding, or any other factors.

Uses for wchar Strings

The single advantage of WC strings is that all characters are the same size. Thus, a string can be treated as an array, and a program can simply index into the array in order to modify its contents. Most applications' *char* manipulation routines work with little modification other than a type change to *wchar_t*, with appropriate attention to byte count and **sizeof()**.

So, when applications have significant string editing to perform, they typically keep the strings in WC format while doing that editing. Those WC strings may or may not be converted to or from MB strings at other points in the application.

Wide characters are often large and are not as space efficient as multibyte strings. Applications that do not need to perform string editing probably shouldn't use *wchar*s. If an application intends to both maintain and edit large numbers of strings, then the developer needs to make size and complexity trade-off decisions.

Support Routines for Wide Characters

Analogs to the routines defined in *string.h* and *stdio.h* are supplied in *libw.a* and defined in *widec.h*. This includes routines such as **getwchar()**, **putwchar()**, **putws()**, **wscpy()**, **wslen()**, and **wsrchr()** (see the wcstring(3) reference page).

Conversion to MB Characters

Wide characters and strings are convertible to MB strings via **wctomb()** and **wcstombs()**, respectively.

Reading Input Data

Input can be divided into two categories: user events and other data. This section deals with nonuser-originated data, which is assumed to come from file descriptors or streams. User events are discussed in "User Input" on page 248.

It is generally fair to assume that unless otherwise specified, data read by an application is encoded suitably for the current locale. Text strings typically are in MB format.

Streams can be read in WC format by using routines defined in *widec.h*.

Cultural Items

This section discusses several aspects of a locale that may differ between locales. It includes these topics:

- "Collating Strings" describes string collation.
- "Specifying Numbers and Money" explains some monetary formats, and the **printf()** and **localeconv()** functions.
- "Formatting Dates and Times" covers using **strftime()** to format of dates and times.
- "Character Classification and ctype" discusses associations between character codes, and using macros and functions from */usr/lib/ctype.h*.
- "Regular Expressions" presents information for developers who do their own regular expression parsing and matching.

Also see "Cultural Data" for additional information.

Collating Strings

Different locales can have different rules governing collation of strings, even within identical encodings.

The Issue

In English, sorting rules are extremely simple: each character sorts to exactly one unique place. Under ASCII, the characters are even in numeric order. However, neither of those statements is necessarily true for other languages and other codesets. Furthermore:

- Sorting order for a language may be completely unrelated to the (numerical) order of the characters in a given encoding.
- Even with a correctly sorted list of the characters in a character set, you may not be able to sort words properly.
- Locales using identically encoded character sets may use very different sorting rules.

Programs using ASCII can do simple arithmetic on characters and directly calculate sorting relationships; such programs frequently rely on truisms such as the fact that

```
'a' < 'b'
```

in ASCII. But internationalized programs cannot rely on ASCII and English sorting rules. Consider some non-English collation rule types:

- *One-to-Two* mappings collate certain characters as if they were two. For example, the German *ß* collates as if it were "ss."
- *Many-to-One* mappings collate a string of characters as if they were one. For example, Spanish sorts "ch" as one character, following "c" and preceding "d." In Spanish, the following list is in correct alphabetical order: *calle*, *creo*, *chocolate*, *decir*.
- *Don't-Care Character* rules collate certain characters as if they were not present. For example, if "-" were a don't-care character, "co-op" and "coop" would sort identically.
- *First-Vowel* rules sort words based first on the first vowel of the word, then by consonants (which may precede or follow the vowel in question).
- *Primary/Secondary* sorts consider some characters as equals until there is a tie. For example, in French, a, á, à, and â all sort to the same primary location. If two strings (such as "tache" and "tâche") collate to the same primary order, then the secondary sort distinguishes them.
- Special case sorts exist for some Asian languages. For example, Japanese *kanji* has no strict sorting rules. *Kanji* strings can be sorted by the strokes that make up the characters, by the *kana* (phonetic) spellings of the characters, or by other agreed-upon rules.

It should be clear that a programmer cannot hope to collate strings by simple arithmetic or by traditional methods.

The Solution

Locale-specific collation should be performed with **strcoll()** and **strxfrm()**. These are table-driven functions; the tables are supplied as part of locale support. The value of LC_COLLATE determines which ordering table to use. (See the strcoll(3) and strxfrm(3) reference pages.)

strcoll() has the same interface as **strcmp()** and can be directly substituted into code that uses **strcmp()**. However, **strcoll()** can consume more CPU time, so where it is used in a time-critical loop you may have to redesign.

Specifying Numbers and Money

Format of simple numbers differs from locale to locale. Characters used for decimal radix and group separators vary. Grouping rules may also vary. Even though we assume that decimal numbers are universal, there are some eighteen varying aspects of numeric formatting defined by a locale. Many of these are details of monetary formatting.

For example, Germany uses a comma to denote a decimal radix and a period to denote a group separator. English reverses these. India groups digits by two except for the last three digits before the decimal radix. Many locales have particular formats used for money, some of which are shown in Table 6-3.

Table 6-3 Some Monetary Formats

Country	Positive Format	Negative Format
India	Rs1,02,34,567.89	Rs(1,02,34,567.89)
Italy	L.10.234.567	-L.10.234.567
Japan	¥10,234,567	-¥10,234,567
Netherlands	F10.234.567,89	F-10.234.567,89
Norway	Kr10.234.567,89	Kr10.234.567,89-
Switzerland	SFr10,234,567.89	SFr10,234,567.89C

Using printf()

printf() function, detailed in the printf(3S) reference page, examines LC_NUMERIC and chooses the appropriate decimal radix. If none is available, it tries to use ASCII period. No further locale-specific formatting is done directly by **printf()**. However, see "Variably Ordered Referencing of printf() Arguments," for a way to handle locale-specific ordering of syntactic elements in messages.

Using localeconv()

The **localeconv()** function, detailed in the localeconv(3C) reference page, can be called to find out about numeric formatting data, including the decimal radix (inappropriately called *decimal_point*), the grouping separator (inappropriately called *thousands_sep*), the grouping rules, and a great deal of monetary formatting information.

The **localeconv()** function leaves actual use of formatting information other than the decimal radix to the application.

Using strfmon()

The **strfmon()** function, detailed in the strfmon(3S) reference page, is new with IRIX version 6.2. Like **sprintf()**, **strfmon()** takes an output area, a format string that contains conversion specifications, and one or more argument values to be converted. It creates an output string containing fixed data and converted values.

Only two conversion types are supported: *%i* to convert a double value to international currency representation, and *%n* to convert a double value to national currency representation. You can use **strfmon()** to format currency values as strings, and then use **printf()** or other functions to write the formatted strings.

Formatting Dates and Times

All of these dates can mean the same thing to different people:

```
92.1.4
4/1/92
1/4/92
```

All of these can mean the same time to different people:

```
2:30 PM

14:30

14h30
```

Dates and times can be easily formatted by using **strftime()**, which gives a host of options for displaying locale-specific dates and times. The **ascftime()** and **cftime()** functions give further options, but should be avoided because they do not conform to ANSI and XPG/4 specifications. The old **asctime()** and **ctime()** functions are now obsolete; use **strftime()** instead. For more information, see the strftime(3C) reference page.

Character Classification and ctype

The *ctype.h* header file is described in the ctype(3C) reference page and defines macros to determine various kinds of information about a given character: **isalpha()**, **isupper()**, **islower()**, **isdigit()**, **isxdigit()**, **isalnum()**, **isspace()**, **ispunct()**, **isprint()**, **isgraph()**, **iscntrl()**, and **isascii()**.

The Issue

When programmers knew that a character set was ASCII, some convenient assumptions could be made about characters and letters. It was common for programmers to do arithmetic with the ASCII code values in order to perform some simple operations. For example, raising a character to upper case could be done by subtracting the difference between the code for *a* and the code for *A*. Numeric characters could be identified by inspection: if they fell between *0* and *9*, they were numeric; otherwise, they weren't. You could tell if a character was (for instance) printable, a letter, or a symbol by comparing to known encoding values. Macros for such activity have long been available in *ctype.h*, but lots of programs did character arithmetic anyway. Since character encoding and linguistic semantics are completely independent, such arithmetic in an internationalized program leads to unpleasant results.

Furthermore, characters exist outside of ASCII that break some non-arithmetic assumptions. Consider the German character *ß* which is a lowercase alphabetic character (letter), yet has no uppercase. Consider also French (as written in France), where the uppercase of *é* is *E*, not *É*.

Clearly, the programmer of an internationalized application has no way of directly computing all the character associations that were available in English under ASCII.

The Solution

Strict avoidance of arithmetic on character values should remove any trouble in this area. The macros in *ctype.h* are table-driven and are therefore locale-sensitive. If you think of characters as abstract characters rather than as the numbers used to represent them, you can avoid pitfalls in this area.

Regular Expressions

XPG/4 specifies some extensions to traditional regular expression syntax for internationalized software. Few application developers do their own regular expression parsing and matching, however, so we do not include full details here. Briefly, the extensions provide the ability to specify matches based on:

- character class (such as *alpha*, *digit*, *punct*, or *space*)
- equivalence class (for instance, *a*, *á*, *à*, *â*, *A*, *Á*, *À*, and *Â* may be equivalent)
- collating symbols (allowing you to match the Spanish *ch* as one element because it is a single collating token)
- generalization of range specifications of the form $[c_1\text{-}c_2]$ to include the above

If you are processing expressions, see the description of internationalized regular expression grammar in "Using Regular Expressions."

Locale-Specific Behavior

You can internationalize an application so it can span a range of language and cultural environments. This section covers some locale-specific topics you should consider when internationalizing an application. Topics include

- "Overview of Locale-Specific Behavior"
- "Native Language Support and the NLS Database"
- "Using Regular Expressions"
- "Cultural Data"

Much of the information in this section is from the *X/Open Portability Guide*. For additional information on locale-specific behavior, refer to the *X/Open Portability Guide*, *Volume 3*, "XSI *Supplementary Definitions*."

Overview of Locale-Specific Behavior

This section covers

- "Local Customs"
- "Regular Expressions"
- "The ANSI X3.159-198X Standard for C"

Local Customs

To meet the requirements of local customs, the X/Open Native Language System (NLS) interface provides a set of library functions that allow cultural data appropriate to the user to be determined at run-time.

Regular Expressions

Regular expressions provide pattern-matching facilities for text. A variety of regular expression support libraries are supplied with IRIX. Most of them parse regular expressions in terms of machine collating sequences, the English language, and the ASCII coded character set.

When a program deals with internationalized input text, it is important to extend regular expression facilities to cover internationalized strings and coded character sets. It is difficult to write regular expressions that apply to more than one language, or to languages with accented/multi-character collating elements because of limitations in syntax.

Application programs can use the *wsregexp* function library, documented in the wsregexp(3W) reference page, to support internationalized regular expression behavior.

The ANSI X3.159-198X Standard for C

The American National Standards Committee X3J11 standard for the C programming language includes a number of library functions that are defined to operate internationally; that is, they modify their operation in a manner appropriate to the user's native language and cultural environment.

The X/Open definition includes the international functions in Table 6-4 as defined in *Draft ANSI X3.159, Programming Language C*. ANSI functions that are enhanced by the X/Open definition are marked with an asterisk.

Table 6-4 ANSI Compatible Functions

Function	
atof()	**scanf()** *
fprintf() *	**setlocale()**
fscanf() *	**sprintf()** *
isalnum()	**sscanf()** *
isalpha()	**strcoll()**
isgraph()	**sterror()**
islower()	**strftime()**
isprint()	**strtod()**
ispunct()	**strxfrm()**
isspace()	**tolower()**
isupper()	**toupper()**
printf() *	

Draft ANSI X3.159, Programming Language C also defines a number of multi-byte functions, and an additional function for manipulating monetary values. At this stage, the X/Open definition is only guaranteed to work correctly for single-byte 8-bit characters, and thus does not include the multi-byte functions.

In addition, X/Open defines internationalized regular expression compile and match functions, native language message-handling functions, and native language versions of the error-handling functions (see Table 6-5).

Table 6-5 X/Open Additional Functions

Function	
catclose()	**regexp()**
catgets()	**vfprintf()**
catopen()	**vprintf()**
nl_langinfo()	**vsprintf()**
perror()	

Native Language Support and the NLS Database

The X/Open NLS interface defines the functional capabilities of a generic database that holds various language-dependent entities. This section describes those entities:

- "Configuration Data"
- "Collating Sequence Tables"
- "Character Classification Tables"
- "Shift Tables"
- "Language Information"

Configuration Data

Configuration data identify the languages supported on a system in terms of the recognized settings of language, territory, and codeset. Each valid combination of these settings has its own set of collating sequence, character classification and shift tables, language information data, and message catalogs.

Collating Sequence Tables

Collating sequence tables define the collating sequence for each supported language. The binary values of characters in the associated coded character set are used as indices into the table, individual entries of which indicate the relative position of that character in the language collating sequence. The interface definition supports the following capabilities:

- one-to-one character mappings
- one-to-two character mappings, where certain characters require treatment as if they were two characters
- *n*-to-one character mappings, where certain character sequences require treatment as if they represented a single character in the collating sequence. The maximum value of *N* is defined separately for each supported language, where *N* is a number in the range [1,{NL_NMAX}].
- don't care characters, where certain characters are ignored by the collating sequence

These capabilities extend to providing support for the relative ordering of collating elements within an equivalent class (for example, where two characters are first compared for equality ignoring accents, and if equal, are then ordered by accent sequence).

Character Classification Tables

These contain the lookup tables for character classification. Each character code from the defined coded character set is used as an index into the relevant language lookup table. Each entry language lookup table contains a series of flags identifying the truth or falsehood of a particular language assertion, such as

- upper-case alphabetic character
- lower-case alphabetic character
- punctuation character
- control character
- space character

Shift Tables

Shift tables contain the corresponding upper- and lower-case combinations for each character defined in a coded character set. Thus, the upshifted or downshifted value of a character can be determined by accessing the relevant character entry in the shift table.

Language Information

Language information (or *langinfo*) contains message text specific to a particular localization. The library function **nl_langinfo()** provides a procedural interface to this data, allowing applications to discover cultural and language-specific information at run-time. Individual items of *langinfo* data are identified by constants in *Volume 2, XSI System Interfaces and Headers,* **<***langinfo.h***>**.

Information specific to a culture or language includes the following :

- Date and time formats
- Days of the week and months of the year
- Abbreviated names of days and months
- Radix character
- Separator for thousands
- Affirmative and negative responses to yes/no questions
- Currency symbol and its position within a currency value

Using Regular Expressions

Regular expression are used widely throughout the services and are powerful mechanisms for locating and manipulating patterns in text. In order to be compatible with a variety of historic UNIX systems, the IRIX Developer's Option includes the unique regular expression library sets listed in Table 6-6. Note that only the last, wsregexp, supports internationalization.

Table 6-6 Regular Expression Libraries in IRIX

Library Documentation	Type of Support Provided
regcmp(3G)	Function **regcmp()** compiles a pattern string; **regex()** applies the pattern to a target string. Syntax is said to be that of *ed* but "syntax and semantics have been changed slightly" in unspecified ways.
regcmp(1)	Command applies **regcmp()** against a file of pattern strings, generating C code for literal strings that can be included in a source program so as to preclude having to compile patterns at run-time.
REGEX(3)	Function **re_comp()** compiles a pattern string; **re_exec()** applies the last-compiled pattern against a target string. No means of storing compiled patterns. No documentation of supported syntax, but cross-references ed(1), with which it may or may not be compatible.
regexp(5)	Function **compile()** compiles a pattern string; **step()** or **advance()** applies a stored pattern against a target string. Unusual interface compiles these functions directly into your source module, using macro functions you must define. Pattern syntax clearly documented.
wsregexp(3W)	Function **wsrecompile()** compiles a pattern string; **wsrestep()** or **wsrematch()** applies a pattern against a target. Both pattern and target strings are wide characters. Expression syntax is that of regexp augmented with internationalization expressions.

Internationalized Regular Expressions

A few utilities distributed with IRIX, in particular *grep* (see the grep(1) reference page) support internationalized regular expressions, which provide additional syntax for matching character classes, sequences, or ranges. The internationalized regular expressions supported by the **wsregexp** library are as shown in Table 6-7.

Table 6-7 Character Expressions in Internationalized Regular Expressions

Expression	Description
c	The single character *c* where *c* is not a special character.
[[:*class*:]]	A character class expression. Any character of type **class**, as defined by category **LC_CTYPE** in the program's locale (for example, see **isalpha()**). For *class*, substitute one of the following: *alpha*, a letter *upper*, an upper-case letter *lower*, a lower-case letter *digit*, a decimal digit *xdigit*, a hexadecimal digit *alnum*, an alphanumeric (letter or digit) *space*, a character that produces white space in displayed text *punct*, a punctuation character *print*, a printing character *graph*, a character with a visible representation *cntrl*, a control character
[[=*c*=]]	An equivalence class. Any collation element defined as having the same relative order in the current collation sequence as *c*. As an example, if *A* and *a* belong to the same equivalence class, then both *[[=A=]b]* and *[[=a=]b]* are equivalent to *[Aab]*.

Table 6-7 (continued) Character Expressions in Internationalized Regular Expressions

Expression	Description
[[.*cc*.]]	A collating symbol. Multi-character collating elements must be represented as collating symbols to distinguish them from single-character collating elements. As an example, if the string *ch* is a valid collating element, then *[[.ch.]]* is treated as an element matching the same string of characters, while *ch* is treated as a simple list of *c* and *h*. If the string is not a valid collating element in the current collating sequence definition, the symbol is treated as an invalid expression.
[*c*-*c*]	Any collation element in the character expression range *c*-*c*, where *c* can identify a collating symbol or an equivalence class. If the hyphen character, -, appears immediately after an opening square bracket, or immediately prior to a closing square bracket, it has no special meaning.

Within square brackets, a period (.) that is not part of a *[[.c.]]* sequence, a colon (:) that is not part of a *[[:class:]]* sequence, and an equals sign (=) that is not part of a *[[=c=]]* sequence matches itself.

Table 6-8 shows examples of simple regular expressions.

Table 6-8 Examples of Internationalized Regular Expressions

Pattern	Definition
[[=a=]]bcd	any form of *a* followed by *bcd*
[[.ch.]-e]	any element that collates between *ch* and *e*
[[:lower:]]	any lower case letter

Cultural Data

The items of cultural data listed in Table 6-9 are defined in the C locale.

Table 6-9 Cultural Data Names, Categories, and Settings

Item	Category	Setting for the C Locale
D_T_FMT	LC_TIME	"%a %b %d %H:%M:%S %Y"
D_FMT	LC_TIME	"%m/%d/%y"
T_FMT	LC_TIME	"%H:%M:%S"
AM_STR	LC_TIME	"AM"
PM_STR	LC_TIME	"PM"
DAY_1	LC_TIME	"Sunday"
DAY_2	LC_TIME	"Monday"
DAY_3	LC_TIME	"Tuesday"
DAY_4	LC_TIME	"Wednesday"
DAY_5	LC_TIME	"Thursday"
DAY_6	LC_TIME	"Friday"
DAY_7	LC_TIME	"Saturday"
ABDAY_1	LC_TIME	"Sun"
ABDAY_2	LC_TIME	"Mon"
ABDAY_3	LC_TIME	"Tue"
ABDAY_4	LC_TIME	"Wed"
ABDAY_5	LC_TIME	"Thu"
ABDAY_6	LC_TIME	"Fri"
ABDAY_7	LC_TIME	"Sat"
MON_1	LC_TIME	"January"
MON_2	LC_TIME	"February"

Table 6-9 (continued) Cultural Data Names, Categories, and Settings

Item	Category	Setting for the C Locale
MON_3	LC_TIME	"March"
MON_4	LC_TIME	"April"
MON_5	LC_TIME	"May"
MON_6	LC_TIME	"June"
MON_7	LC_TIME	"July"
MON_8	LC_TIME	"August"
MON_9	LC_TIME	"September"
MON_10	LC_TIME	"October"
MON_11	LC_TIME	"November"
MON_12	LC_TIME	"December"
ABMON_1	LC_TIME	"Jan"
ABMON_2	LC_TIME	"Feb"
ABMON_3	LC_TIME	"Mar"
ABMON_4	LC_TIME	"Apr"
ABMON_5	LC_TIME	"May"
ABMON_6	LC_TIME	"Jun"
ABMON_7	LC_TIME	"Jul"
ABMON_8	LC_TIME	"Aug"
ABMON_9	LC_TIME	"Sep"
ABMON_10	LC_TIME	"Oct"
ABMON_11	LC_TIME	"Nov"
ABMON_12	LC_TIME	"Dec"
RADIXCHAR	LC_NUMERIC	"."
THOUSEP	LC_NUMERIC	""

Table 6-9 (continued) Cultural Data Names, Categories, and Settings

Item	Category	Setting for the C Locale
YESSTR	LC_ALL	"yes"
NOSTR	LC_ALL	"no"
CRNCYSTR	LC_MONENTARY	" "

NLS Interfaces

The NLS interfaces listed here are utilities and library functions.

NLS Utilities

The list below identifies the minimum set of utilities that provide 8-bit transparency on all X/Open compliant systems. The definitions of these commands, in terms of their syntax and parameters, are not changed by the operation of NLS.

ar	*date*	*kill*	*pg*	*tail*	*uulog*
awk	*diff*	*lex*	*pr*	*tar*	*uuname*
cancel	*echo*	*ln*	*ps*	*tee*	*uupick*
cat	*ed*	*lp*	*pwd*	*test*	*uustat*
cc	*egrep*	*lpstat*	*red*	*tr*	*uuto*
cd	*expr*	*ls*	*rm*	*true*	*uux*
chgrp	*false*	*mail*	*rmdir*	*tty*	*wait*
chmod	*fgrep*	*mailx*	*sed*	*umask*	*wc*
chown	*find*	*mkdir*	*sh*	*uname*	*who*
cmp	*gencat*	*mv*	*sleep*	*uniq*	
cp	*grep*	*pack*	*sort*	*unpack*	
cpio	*iconv*	*pcat*	*stty*	*uucp*	

The *cc*, *yacc*, and *lex* commands provide 8-bit transparency for characters contained in character strings, character constants, and comment strings. An 8-bit character string enables a programmer to define default messages in languages other than English. The support of 8-bit characters in identifier names is implementation defined.

The 8-bit operation of commands that communicate with other systems cannot be guaranteed in all circumstances. For example, intersystem mail may be restricted to 7-bit data by the underlying network, 8-bit data and filenames may not be portable to noninternationalized systems, and so forth. Under these circumstances, it is recommended that you use only characters defined in the ASCII 7-bit range of characters for data transfer between machines, and you use only characters defined in the Portable Filename Character Set for naming remote files.

NLS Library Functions

The list below shows library functions usable by internationalized application programs

atof()	**isgraph()**	**scanf()**	**toupper()**
catclose()	**islower()**	**setlocale()**	**vfprintf()**
catgets()	**isprint()**	**sprintf()**	**vprintf()**
catopen()	**ispunct()**	**sscanf()**	**vsprintf()**
fprint()	**isspace()**	**strcoll()**	
fscanf()	**isupper()**	**strerror()**	
gcvt()	**nl_langinfo()**	**strftime()**	
isalnum()	**perror()**	**strtod()**	
isalpha()	**printf()**	**strxfrm()**	
iscntrl()	**regexp()**	**tolower()**	

Also, all functions defined in the *X/Open Portability Guide, Volume 2, XSI System Interfaces and Headers*, and *X/Open Portability Guide, Volume 3, XSI Curses Interface*, provide 8-bit transparency on X/Open compliant systems.

XSI Curses Interface

The XSI curses interface is internationalized. For more information, see the *X/Open Portability Guide, Volume 3, XSI Curses Interface*.

Strings and Message Catalogs

Message catalogs are compiled databases of strings. While a major role of message catalogs is to provide communications text in locale-specific natural language, the strings can be used for any purpose. The idea is that an application uses only strings from a catalog, thus allowing localizers to supply catalogs suitable for a given locale.

Two different and incompatible interfaces to message catalogs exist in IRIX: *MNLS* and *XPG/4*. Developers working on SVR4 or other AT&T code, or related base-system utilities, probably use MNLS. Developers working on independent projects probably use XPG/4. Neither is a solid standard, but XPG/4 is closer to being a standard than MNLS. Thus applications developers who have to choose between the two interfaces are encouraged to use XPG/4 to maximize their portability. XPG/4 seems to be popular in Europe.

This section covers the following topics:

- "XPG/4 Message Catalogs"
- "SVR4 MNLS Message Catalogs"
- "Variably Ordered Referencing of printf() Arguments"

XPG/4 Message Catalogs

The XPG/4 message catalog interface requires that a catalog be opened before it is read, and requires that catalog references specify a catalog descriptor.

Since catalog references include a default to be used in case of failure, applications will work normally without a catalog when in the default locale. This means catalog generation is exclusively the task of localizers. But in order to inform the localizer as to what strings to translate and how they should comprise a catalog, the application developer should provide a catalog for the developer's locale.

Opening and Closing XPG/4 Catalogs

catopen() locates and opens a message catalog file:

```
#include <nl_types.h>
nl_catd catopen(char *name, int unused);
```

The argument *name* is used to locate the catalog. Usually, this is a simple, relative pathname that is combined with environment variables to indicate the path to the catalog (see "XPG/4 Catalog Location" for details). However, the catalog assumes names that begin with "/ " are absolute pathnames. Use of a hard-coded pathname like this is strongly discouraged; it doesn't allow the user to specify the catalog's locale through environment variables.

When an application is finished using a message catalog, it should close the catalog and free the descriptor using **catclose()**:

```
int catclose(nl_catd);
```

Using an XPG/4 Catalog

Catalogs contain sets of numbered messages. The application developer must know the contents of the catalog in order to specify the set and number of a message to be obtained.

catgets() is used to retrieve strings from a message catalog (see the catopen(3) and catgets(3) reference pages). Example 6-3 shows a program that reads the first message from the first message set in the appropriate catalog, and displays the result.

Example 6-3 Reading an XPG/4 Catalog

```
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>

#define SET1      1
#define WRLD_MSG  1

int main(){
    nl_catd msgd;
    char *message;
    setlocale(LC_ALL, "");

msgd = catopen("hw",0);
    message = catgets(msgd, SET1, WRLD_MSG,"Hello, world\n");
    printf(message);
    catclose(msgd);
}
```

The previous example uses **printf()** instead of **puts()** in order to make a point: the format string of **printf()** came from a catalog. Note the crucial difference between these two statements:

```
printf(catgets(msgd, set, num, defaultStr));
printf("%s", catgets(msgd, set, num, defaultStr));
```

In the first statement, the catalog provides the **printf()** formatting string, possibly containing conversion specifications and escape sequences. In the second statement, the string from the catalog is treated as data and not interpreted for conversion specifications. For further discussion of issues relating to this important distinction, see "Variably Ordered Referencing of printf() Arguments."

XPG/4 Catalog Location

XPG/4 message catalogs are located using the environment variable NLSPATH. The default NLSPATH is */nlslib/%L/%N*, where *%L* is filled in by the LANG environment variable and *%N* is filled in by the *name* argument to **catopen()**. NLSPATH can specify multiple pathnames in ordered precedence, much like the PATH variable. The following is a sample NLSPATH assignment:

```
NLSPATH=/usr/lib/locale/%L/%N:/usr/local/lib/locale/%L/%N:/usr/defaults/%N
```

Creating XPG/4 Message Catalogs

Message catalogs are of this general form (these forms are detailed in the gencat(1) reference page):

```
$set n comment
a message-a\n
b message-b\n
c message-c\n
$quote "
d " message-d "
$this is a comment
```

Each message is identified by a *message number* and a *set*. Sets are often used to separate messages into more easily usable groups, such as error messages, help messages, directives, and so on. Alternatively, you could use a different set for each source file, containing all of that source file's messages.

$set *n* specifies the beginning of set *n*, where *n* is a set identifier in the range from 1 to NL_SETMAX. All messages following the **$set** statement belong to set *n* until either a **$delset** or another **$set** is reached. You can skip set numbers (for example, you can have a set 3 without having a set 2), but the set numbers that you use must be listed in ascending numerical order (and every set must have a number). Any string following the set identifier on the same line is considered a comment.

$delset *n* deletes the set *n* from a message catalog.

$quote *c* specifies a quote character, *c*, which can be used to surround message text so that trailing spaces or null (empty) messages are visible in a message source line. By default, there is no quote character and messages are separated by newlines. To continue a message onto a second line, add a backslash to the end of the first line:

```
$set 1
1 Hello, world.
2 here is a long \
string.\n
3 Hello again.
n message-text-n
```

Message #2 in set #1 is "here is a long string.\n".

Compiling XPG/4 Message Catalogs

After creating the message catalog sources, you need to compile them into binary form using *gencat*, which has the following syntax:

```
gencat catfile msgfile [msgfile ...]
```

where *catfile* is the target message catalog and *msgfile* is the message source file (see the gencat(1) reference page). If an old *catfile* exists, *gencat* attempts to merge new entries with the old. *gencat* "resolves" set and message number conflicts with new information replacing the old.

The *catfile* then needs to be placed in a location where **catopen()** can find it; see the "XPG/4 Catalog Location" on page 231.

SVR4 MNLS Message Catalogs

There are many ways to use strings from MNLS message catalogs. You can get strings directly and then use them, or you can use output routines that search catalogs.

Putting MNLS Strings Into a Catalog

An MNLS catalog source file contains a list of strings separated by new lines. For an empty string, an empty line is used. Strings are referenced by line number in the original source file.

Applications access the catalog by line number, so it's very important not to change the line numbers of existing catalog entries. This means that, when you want to add a new string to an existing catalog source, you should always append it to the end of the file—if you put it in the middle of the file, then you change the line number for subsequent strings.

The following tools can help you compile MNLS message catalogs:

exstr(1)	Searches a C source file for literal strings and lists them, or replaces them with MNLS function calls.
mkmsgs(1)	Creates a message catalog for a particular locale, converting source text lines to the form used by exstr.
srchtxt(1)	Displays selected strings from a message catalog.

When a file of strings is ready to be compiled, simply run *mkmsgs* and put the results in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Using MNLS in Shell Scripts

One difference between MNLS and XPG/4 catalog functions is that the MNLS catalog can be used from commands, and hence it can be used to internationalize a shell script. The following table summarizes MNLS functions that have both a command line and a function library version:

gettxt(1)	Retrieve a string from the catalog.
lfmt(1)	Retrieve a format string, insert arguments, display to *stderr* and to system log or textport.
pfmt(1)	Retrieve a format string, insert arguments, display to *stderr*.

Specifying MNLS Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified every time a string is needed.

To specify the default message catalog to be used by subsequent calls to MNLS functions that reference catalogs, use **setcat()**:

```
#include <pfmt.h>
char *setcat(const char *catalog);
```

catalog is limited to 14 characters, and may contain no character equal to zero or to the ASCII codes for slash (/) or colon (:). (See the setcat(3) reference page.)

setcat() doesn't check to see if the catalog name is valid; it just stores the string for future reference. For an example of use, see the following topic. The catalog indicated by the string must be found in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Getting Strings From MNLS Message Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified in each reference call. Strings are read from a catalog via **gettxt()** (see the gettxt(3) reference page):

```
#include <unistd.h>
char *gettxt(const char *msgid, const char *defaultStr);
```

msgid is a string containing two fields separated by a colon:

```
msgfilename:msgnumber
```

The *msgfilename* is a catalog name as described previously in the "Specifying MNLS Catalogs" on page 234. For example, to get message 10 from the *MQ* catalog, you could use either:

```
char *str = gettxt("MQ:10", "Hello, world.\n");
```

or

```
setcat("MQ");
str = gettxt(":10", "Hello, world.\n");
```

Using pfmt()

pfmt() is one of the most important routines dealing with MNLS catalogs, because it is used to produce most system diagnostic messages. **pfmt()** formats like **printf()** and produces standard error message formats (see the pfmt(3) reference page for the function, or pfmt(1) for shell use). It can usually be used in place of **perror()**. For example,

```
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce, by default (such as when the Mozambique locale is unavailable),

```
ERROR: Permission denied.
```

The syntax of **pfmt()** is

```
#include <pfmt.h>
int pfmt(FILE *stream, long flags, char *format, ... );
```

The *flags* are used to indicate severity, type, or control details to **pfmt()**. The format string includes information specifying which message from which catalog to look for. Flag details are discussed in the following section. The format is discussed in the "Format Strings for pfmt()" on page 236.

Labels, Severity, and Flags

pfmt() flags are composed of several groups; specify no more than one from each group. Specify multiple flags by using OR. The groups are as follows:

output format control	MM_NOSTD, MM_STD
catalog access control	MM_NOGET, MM_GET
severity	MM_HALT, MM_ERROR, MM_WARNING, MM_INFO
action message specification	MM_ACTION

pfmt() prints messages in the form *label:severity:text*. *Severity* is specified in the *flags*. The *text* comes from a message catalog (or a default) as specified in the *format*, and the *label* is specified earlier by the application.

In the example above, if no label has been set, we get only the output:

```
ERROR: Permission denied.
```

Typically, an application sets the label once early in its life; subsequent error messages have the label prepended. For example

```
setlabel("UX:myprog");
...
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce (by default)

```
UX:myprog: ERROR: Permission denied.
```

For details, consult the **pfmt(**3**)** and **setlabel(**3**)** reference pages.

Format Strings for pfmt()

pfmt() format strings are of this form:

`[[`*catalog*`:]`*messagenum*`:]`*defaultstring*

The *catalog* field is in the format described in "Specifying MNLS Catalogs" on page 234. *messagenum* is the message number in the catalog to use as the format. *defaultstring* specifies the string to use if the catalog lookup fails for any reason.

An important feature of **pfmt()** is its ability to refer to format arguments in format-specified order just as **printf()** does. See "Variably Ordered Referencing of printf() Arguments" for details.

Using fmtmsg()

fmtmsg() is a comprehensive formatter using the MNLS catalogs and "standard" formats. You probably won't need to use it; most applications should get by with **pfmt()**, **gettxt()**, and **printf()**. Consult the **fmtmsg(**3**)** reference page for details.

Internationalizing File Typing Rule Strings With MNLS

You can internationalize the strings defined in the LEGEND and MENUCMD rules in the File Typing Rule (FTR) file. To internationalize these rules, precede the string with the following:

`:[`*catalogname*`:]`*msgnumber*`:`

catalogname is optional and should be a valid MNLS catalog; *msgnumber* is the line number in *catalogname*. If you omit *catalogname*, the *uxsgidesktop* catalog is used by default.

You can use these rules to create your own FTR catalog. For example, an entry looks like this:

```
LEGEND :mycatalog:7:Archive 8mm Tape Drive
```

This entry uses line 7 from the catalog, *mycatalog*, as the LEGEND for this FTR. If *mycatalog* is not available, or line 7 is not accessible from *mycatalog*, "Archive 8mm Tape Drive" is used as the LEGEND.

```
LEGEND :7:Archive 8mm Tape Drive
```

This entry uses line 7 from the *uxsgidesktop* catalog, if available. Otherwise, "Archive 8mm Tape Drive" is used.

The next example,

```
MENUCMD \‘mycatalog:9:Eject Tape\’ /usr/sbin/eject /dev/tape
```

displays line 9 from *mycatalog*, if available. Otherwise "Eject Tape" is displayed on the menu that pops up when you click an icon that uses this FTR.

You can internationalize strings in the command part of MENUCMD and CMD rules by using *gettxt* or any other convenient policy detailed in this section. For example

```
CMD OPEN xconfirm –t "Tape tool not available"
```

can be internationalized to

```
CMD OPEN xconfirm –t "‘gettxt mycatalog:376 ’Tape tool not available’‘"
```

In this example, *gettxt* is invoked to access line 376 from the catalog, *mycatalog*, and the string returned by *gettxt* is passed to *xconfirm* for display. If line 376 from *mycatalog* is not accessible, then *gettxt* returns the string "Tape tool not available."

For more information about FTRs, see the *Indigo Magic Desktop Integration Guide*.

Variably Ordered Referencing of printf() Arguments

printf() and its variants can now refer to arguments in any specified order. Consider the following scenario: an application has chosen "house" from a list of objects and "white" from a list of colors. The application wishes to display this choice. The code might look like this:

```
char *obj, *color;
... /* make choices */ ...
printf("%s %s\n", color, obj);
```

The **printf()** call produces this:

```
white house
```

Even once we make sure that *obj* and *color* are localized strings, we are not quite finished. If our locale is Spanish, the **printf()** yields:

```
blanca casa
```

That is incorrect grammar; in Spanish, it should be:

```
casa blanca
```

The solution to this problem is *variably ordered referencing* of **printf()** arguments. The syntax of **printf()** format strings has been expanded to deal with this.

The original definition of **printf()** is that each conversion specification %*T* (where *T* represents any of the **printf()** conversion characters) is implicitly matched to an argument value by position. In order to deal with variably ordered strings, **printf()** allows an argument position index *D* to appear in the conversion specification following the %, so that where a format string contains %*T* , it can now contain %*D*$*T*. The value *D*, set off by a currency symbol ($), selects the argument from the argument list to be used. This means you can write

```
printf("2nd parameter is %2$s; the 1st is %1$s", p1, p2)
```

The *second* parameter is printed *first*, with the first parameter printed second. For example:

```
char *store = "Macy's";
char *obj = "a cup";

printf("At %1$s, I bought %2$s.\n", store, obj);
printf("I bought %2$s at %1$s.\n", store, obj);
```

This code displays

```
At Macy's, I bought a cup.
I bought a cup at Macy's.
```

In English, we are able to come up with strings suitable for either word order; in some other language, we might not be so lucky. Nor can we predict which order such languages might prefer. So the developer has no way of knowing how to create traditional **printf()** format strings suitable for all languages.

Developers should therefore use message catalogs for their **printf()** format strings that take linguistic parameters, and allow localizers to localize the format strings as well as text strings. This means that the localizer has much greater ability to create intelligible text. An internationalized version of the above code appears in Example 6-4.

Example 6-4 Internationalized Code

```
/* internationalized (XPG/4) version */
char *form = catgets(msgd, set, formNum,
                  "At %1$s, I bought %2$s.\n");
char *store = catgets(msgd, set, storeNum, "Macy's");
char *obj = catgets(msgd, set, objNum, "a cup");

printf(form, store, obj);
```

The unlocalized (default) version would produce

```
At Macy's, I bought a cup.
```

A localized version might produce

```
Compré una tasa en Macy's.
```

In practice, variably ordered format strings are found only in message catalogs and not in default strings. The default string usually simply uses the parameters in the order they're given, without the new variable-order format strings.

Internationalization Support in X11R6

X11R6 internationalization support is provided on the X client side; that is, the application must take care of such support instead of relying on the X server. No server changes are necessary, and the protocol is unchanged. Full backward compatibility is preserved, so a new internationalized application can run on an old server.

Note: X11R6 internationalization refers to features in X11R5 and X11R6.

X uses existing internationalization standards to do its internationalization support; there are no X-specific interfaces to set and change locale. Internationalized X applications receive no help from X when attempting multilingual support. No locales or special process states are peculiar to X.

This section covers the following topics:

- "Limitations of X11R6 in Supporting Internationalization" discusses vertical text, character sets, and Xlib interface changes.
- "Resource Names" covers encoding of resource names.
- "Getting X Internationalization Started" describes initialization of Xlib and toolkit programming.
- "Fontsets" explains specifying, creating, and using fontsets.
- "Text Rendering Routines" discusses the *XmbDrawText()*, *XmbDrawString()*, and *XmbDrawImageString()* functions.
- "New Text Extents Functions" describes a few new extents-related functions, including *XFontSetExtents*.

Limitations of X11R6 in Supporting Internationalization

Since X is locale-independent, there are some limitations on its ability to support internationalization. The X protocol and Xlib specification, together with ANSI C and POSIX restrictions, have led to certain choices being made in X11R6. These are described in the following paragraphs.

Vertical Text

There is no built-in support for vertical text. Applications may draw strings vertically only by laying out the text manually.

Character Sets

In previous releases of X, there was no general support for character sets other than Latin 1. X11R6, however, does allow other character sets.

X11R6 includes the definition of the *X Portable Character Set*, which is required to exist in all locales supported by Xlib. There is no encoding defined for this set; it is only a character set. The set—which is similar to printable ASCII plus the newline and tab—consists of these characters:

```
abcdefghijklmnoqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
<space> <tab> <newline>
```

The *Host Portable Character Encoding* is the encoding of the X Portable Character Set on the Xlib host. This encoding is part of X, and is thus independent of locale—the coding remains the same for all locales supported by the host.

Strings used or returned by Xlib routines are either in the Host Portable Character Encoding or a locale-specific encoding. The Xlib reference pages specify which encodings are used where. Some string constructs (such as *TextProperty*) contain information regarding their own encoding.

Xlib Interface Change

Full use of X11R6's internationalization features means calling some new routines supplied in the X11R6 Xlib. While all old Xlib applications work with the new Xlib, developers should change their code in places. These are described below.

Resource Names

Resource names are compiled into programs. Because of that, their encoding must be known independent of locale. Trying to add a level of indirection here results in a problem: you're always left with something compiled that can't be localized. Resource names therefore use the X Portable Character Set. The names may be anything; at least they'll mean something to the application author. (If the names were numbers, for example, they would be meaningless to everybody.)

Getting X Internationalization Started

Xlib's internationalization state, like that of *libc*, needs to be initialized.

Initialization for Toolkit Programming

If you're using Xt (with a widget set such as IRIS IM, Motif, or XaW), then don't use **setlocale()**. Instead, use

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

If you're using a toolkit other than Xt, call **setlocale()** as early as possible after execution begins.

Initialization for Xlib Programming

Initialize Xlib's internationalization state after calling **setlocale()**. Xlib is being initialized, not a server or server-specific object, so a server connection is not necessary.

Example 6-5 Initializing Xlib for a Locale

```
if ( setlocale(LC_ALL, "") == NULL )
    exit_with_error();
if ( ! XSupportsLocale() )
    exit_with_other_error();
if ( XSetLocaleModifiers("") == NULL)
    give_warning();
```

XSetLocaleModifiers() is required only for input. Just as passing an empty string to **setlocale()** honors the user's environment, so does passing an empty string to **XSetLocaleModifiers()**.

Fontsets

In X11R5 and X11R6, unlike previous releases of X, a string may contain characters from more than one codeset. There are several methods for determining which codeset a given character is in; which method is appropriate depends on the locale and the encoding used.

For information on installing and using fontsets with an application, refer to Chapter 5, "Working With Fonts."

Such multiple-codeset strings usually cannot be rendered using a single font. A *fontset* is a collection of fonts suitable for rendering all codesets represented in a locale's encoding. A fontset includes information to indicate which locale it was created in. Applications create fontsets for their own use; when a program creates a fontset, it is told which of the requested fonts are unavailable.

Example: EUC in Japanese

To render strings encoded in EUC in Japanese, an application would need fonts encoded in 8859-1, JIS X 208, and JIS X 201. The application doesn't need to know which characters in a string go with which font, since it doesn't deal with locale specifics. So it creates a fontset that is made from a list of user-specified fonts (under the assumption that the localizer has provided an appropriate list). Rendering is then done using that fontset. The locale-aware rendering system chooses the appropriate fonts for each character being rendered, from the supplied list. You can find additional information about EUC in "Asian Languages."

Specifying a Fontset

A fontset specification is just a string, enumerating XLFD names of fonts. (See *X Logical Font Description Conventions*, an MIT X Consortium standard, as well as "Font Names" on page 171.) This string can include wild card characters. For example, a specification of 16-point "fixed" fonts might be as follows:

```
char *fontSetSpecString = "*fixed-medium-r-normal*150*";
```

Based on the fonts available, a particular server might expand this to a string such as:

```
-jis-fixed-medium-r-normal--16-150-75-75-c-160-jisx0208.1983-0
-sony-fixed-medium-r-normal--16-150-75-75-c-80-iso8859-1
-sony-fixed-medium-r-normal--16-150-75-75-c-80-jisx0201.1976-0
```

Specifying the fontset by simply enumerating the fonts is perfectly acceptable:

```
char *fontSetSpecString =
"-jis-fixed-medium-r-normal*150-75-75*jisx0208.1983-0,\
-sony-fixed-medium-r-normal*150-75-75*iso8859-1,\
-sony-fixed-medium-r-normal*150-75-75*jisx0201.1976-0";
```

A German locale would work with only the ISO font; a Japanese locale might use all three; a Chinese locale would have trouble with this fontset.

The developer should specify a default fontset suitable for the default locale. Furthermore, developers should ensure that the application accepts localized fontset specifications via resources (or message catalogs) or command line options. Localizers are responsible for providing default fontset specifications suitable for their locales.

Creating a Fontset

Creating fontsets in X is simply a matter of providing a string that names the fonts, as described above.

Example 6-6 Creating a Fontset

```
XFontSet fontset;
char *base_name;  /* should get from resource */
char **missingCharsetList;
int missingCharsetCount;
char *defaultStringForMissingCharsets;

base_name = "*fixed-medium-r*150*"; /* use resources! */

fontset = XCreateFontSet(display, base_name,
                         &missingCharsetList,
                         &missingCharsetCount,
                         &defaultStringForMissingCharsets);
```

The locale in effect at create time is bound to the fontset. Fontsets are freed with **XFreeFontSet()**.

Using a Fontset

Fontsets are used when rendering text with X11R6 **Xmb** or **Xwc** text rendering routines. These routines are described in "Text Rendering Routines."

Text Rendering Routines

X11R6 includes text rendering routines that understand multibyte and wide-character strings. These routines are analogous to the X11R4 text rendering routines **XDrawText()**, **XDrawString()**, and **XDrawImageString()**. The old routines continue to operate, but do not take fontsets, and don't know how to handle characters longer than one byte.

- **XmbDrawText()** and **XwcDrawText()** take lists of *TextItems*, each of which contains (among other things) a string. The strings are rendered using fontsets. These routines allow complex spacing and fontset shifts between strings.
- **XmbDrawString()** and **XwcDrawString()** render a string using a fontset. These routines render in foreground only and use the raster operation from the current graphics context.
- **XmbDrawImageString()** and **XwcDrawImageString** also render a string using a fontset. These routines fill the background rectangle of the entire string with the background, then render the string in the foreground color, ignoring the currently active raster operation.

Consult the appropriate reference pages for more details on these routines.

New Text Extents Functions

X11R6 provides MB and WC versions of **width** and **extents** interrogation routines, supplying the maximum amount of space required to draw any character in a given fontset. These routines depend on fontsets to interpret strings and use locale-specific data.

The *XFontSetExtents* structure contains the two kinds of extents a string can have:

```
typedef struct {
    XRectangle max_ink_extent;
    XRectangle max_logical_extent;
} XFontSetExtents;
```

max_ink_extent gives the maximum boundaries needed to render the drawable characters of a fontset. It considers only the parts of glyphs that would be drawn, and gives distances relative to a constant origin. *max_logical_extent* gives the maximum extent of the *occupied space* of drawable characters of a fontset. The occupied space of a character is a rectangle specifying the minimum distance from other graphical features; other graphics generated by a client should not intersect this rectangle. *max_logical_extent* is used to compute interline spacing and the minimum amount of space needed for a given number of characters.

Here are descriptions of a few of the new extents-related functions (consult the appropriate reference pages for details):

- **XExtentsOfFontSet()** returns an *XFontSetExtents* structure for a fontset.
- **XmbTextEscapement()** and **XwcTextEscapement()** take a string and return the distance in pixels (in the current drawing direction) to the origin of the next character after the string, if the string were drawn. Escapement is always positive, regardless of direction.
- **XmbTextExtents()** and **XwcTextExtents()** take a string and return information detailing the overall rectangle bounding the string's image and the space the string occupies (for spacing purposes).
- **XmbTextPerCharExtents()** and **XwcTextPerCharExtents()** take a string and return ink and logical extents for each character in the string. Use this for redrawing portions of strings or for word justification. If the fontset might include context-dependent drawing, the client cannot assume that it can redraw individual characters and get the same rendering.
- **XContextDependentDrawing()** returns a Boolean telling whether a fontset might include context-dependent drawing.

Internationalization Support in Motif

Your applications can use Motif's internationalization capabilities. Refer to the chapter titled "Internationalization" in the *OSF/Motif Programmer's Guide* for information about the following topics:

- issues in internationalized applications
- compound strings, fonts, and text display
- localizing applications
- advanced topics in internationalization

There are some important points to remember when you internationalize and localize your application:

- At the top of your **main** program, issue the call

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

- Translate your app-defaults and install it in */usr/lib/X11/$LANG/app-defaults*.
- Motif uses font sets and font lists to display text. Specify a font list in your application defaults file using the following format:

  `*fontList:` *font-list-string*`:`

Be sure to separate elements in the *font-list-string* as follows:

- Separate single fonts with a comma (,).
- Separate elements within a font set with a semicolon (;).
- End the string with a colon (:).

An example of specifying a Japanese *fontList* is as follows:

```
*fontList: 7x14;--mincho-*--14-*;--14-*:
```

User Input

This section explains the translation of physical user events into programmatic character strings or special keyboard data (such as "backspace"). This kind of work should be done by toolkits. If you can use a toolkit to manage event processing for you, do so, and blissfully ignore this section. If you are writing a toolkit text object, or are writing a truly extraordinary application, then this section is for you.

This section on user input covers these topics:

- "About User Input and Input Methods" presents an overview of user input and input methods.
- "About X Keyboard Support" covers X keyboard support, including keys, keycodes, keysyms, and composed characters.
- "Input Methods (IMs)" describes opening and closing input methods, and IM styles.
- "Input Contexts (ICs)" explains an IM styles, IC values, pre-edit and status attributes, and creating and using ICs.
- "Events Under IM Control" describes differences in processing events under IM control including *XFilterEvent()* and *LookupString* routines.

About User Input and Input Methods

Just as internationalized programs cannot assume that data is in ASCII, they cannot assume that user input will use any specific keyboard. Keyboards change from country to country and language to language; internationalized software should never assume that a certain position on the keyboard is bound to a certain character, or that a given character will be available as a single keystroke on all keyboards.

No useful physical keyboard—not even one specifically designed for multilingual work—could possibly contain a key for every character we would ever wish to type. Certainly there are characters commonly used in other areas of the world that are not present on most USA keyboards. So methods have been invented that provide for input of almost any known character on even the most naïve keyboards. These schemes are referred to as *input methods* (IMs).

Input methods vary significantly in design, use, and behavior, but there is a single API that developers use to access them. The object is for the application simply to ask for an IM and let the system check the locale and choose the appropriate IM.

Some IMs are complex; others are very simple. The API is designed to be a low-level interface, like Xlib. Usually, only toolkit text object authors must deal with the IM interfaces. However, some applications developers are unable to use toolkit objects, so the concepts are described here.

Reuse Sample Code

A sample program demonstrating some of the concepts in this section is given in Chapter 11 of the *Xlib Programming Manual, Volume One*. Looking carefully at that code may be easier than starting from scratch.

GL Input

The old GL function **qdevice()** has a hard-coded view of a keyboard (see */usr/include/gl/device.h* for details). Some flexibility, particularly for Europe, is available if you queue KEYBD instead of individual keys, but the GL has no general solution to non-ASCII input. There is no supported way to input Chinese (for instance) to the old GL.

OpenGL does not contain input code but leaves that to the operating environment, which in IRIX means X.

In short, support for internationalized input means a departure from **qread()**. Under IRIX, that means using mixed-model input, all the more reason to use a toolkit.

About X Keyboard Support

This section provides some background that may help make the following sections easier to understand.

Keys, Keycodes, and Keysyms

When a client connects to the X server, the server announces its range of *keycodes* and exports a table of *keysyms*. Each key event the client receives has a single byte *keycode*, which directly represents a physical key, and a single byte *state*, which represents currently engaged modifier keys, such as Shift or Alt.

Note: The mapping of state bits to modifiers is done by another table acquired from the server.

Keysyms are well defined, and there has been an attempt to have a keysym for every engraving one might possibly find on any keyboard, anywhere. (An *engraving* is the image imprinted on a physical key.) These are contained in */usr/include/X11/keysymdef.h*. Keysyms represent the engravings on the actual keys, but not their meanings. The server's idea of the keysym table can be changed by clients, and clients may receive *KeyMap* events when this remapping happens, but such events don't happen often.

When a client receives a Key event, it asks Xlib to use the keycode to index into its keysym table to find a list of keysyms. (This list is usually very short. Most keys have only one or two engravings on them.) Using the state byte, Xlib chooses a keysym from the list to find out what was engraved on the key the user pressed.

At this point, the client can choose to act on the keysym itself (if, for instance, it was a backspace) or it can ask for a character string represented by the keysym (or both). Generating such a string is tricky; it is discussed in "Input Methods (IMs)," below.

Details on X keyboard support can be found in *X Window System, Third Edition*, from Digital Press. Details on input methods are also available in that book, as well as in the *Xlib Programming Manual, Volume One*.

Composed Characters

There are two ways to compose characters that do not exist on a keyboard: explicit and implicit. It is common for an application to be modal and switch between the two. For example, Japanese input of kana is often done via implicit composition.

Users switch between a mode where input is interpreted as romaji (Latin characters) and a mode where input is translated to kana.

Furthermore, both styles may operate simultaneously. While an application is supporting implicit composition of certain characters, other characters may be composable via explicit composition.

Not every keystroke produces a character, even if the associated keysym normally implies character text. The event-to-string translation routines figure out what result a given set of keystrokes should produce (see "Using XLookupString(), XwcLookupString(), and XmbLookupString()" in this section).

Character composition from the user's aspect is discussed in the compose(5) and composetable(5) reference pages.

Explicit Composition

Explicit composition is requested when the user presses the Compose key and then types a key sequence that corresponds to the desired character. For example, to compose the character ñ under some keymaps, you might press the Compose key and then type `~n`.

Note: The xmodmap(1) reference page tells how to map the XK_Multi_key keysym onto whatever key you want to use as Compose.

Implicit Composition

Implicit composition mimics many existing European typewriters that have "dead" keys: keys that type a character but do not advance the carriage. When a special "dead" key is struck, the system attempts to compose a character using the next character struck. For example, on a keyboard that had a diaeresis (¨) and an O, but no Ö, you would strike ¨ and then o to compose Ö.

Implicit composition support usually comes with some specified way to leave characters uncomposed.

Supported Keyboards

IRIX currently supports 12 keyboard layouts: American, Belgian, Danish, English, French, German, Italian, Norwegian, Portuguese, Spanish, Swedish, and Swiss. All are representable in Latin 1; the American keyboard needs only ASCII.

Input Methods (IMs)

Input methods (IMs) are ways to translate keyboard-input events into text strings. You would use a different input method, for instance, to type on a USA keyboard in Chinese than to type on the same keyboard in English. Nobody would build a keyboard suitable for direct input of the tens of thousands of distinct Chinese characters.

IMs come in two flavors, *front-end* and *back-end*. Both types can use identical application programming interfaces, so you lose no generality by using back-end methods for our examples here.

To use an IM, follow these steps:

1. Open the IM.
2. Find out what the IM can do.
3. Agree upon capabilities to use.
4. Create input contexts with preferences and window(s) specified (see "Input Contexts (ICs)" on page 257).
5. Set the input context focus.
6. Process events.

Although all applications go through the same setup when establishing input methods, the results can vary widely. In a Japanese locale, you might end up with networked communications with an input method server and a *kanji* translation server, with circuitous paths for Key events. But in a Swiss locale for example, it is likely that nothing would occur besides a flag or two being set in Xlib. Since operating in non-Asian locales ends up bypassing almost all of the things that might make input methods expensive, Western users are not noticeably penalized for using Asia-ready applications.

Opening an Input Method

XOpenIM() opens an input method appropriate for the locale and modifiers in effect when it is called (see the XOpenIM(3X11) reference page). The locale is bound to that IM and cannot be changed. (But you could open another IM if you wanted to switch later.) Strings returned by **XmbLookupString()** and **XwcLookupString()** are encoded in the locale that was current when the IM was opened, regardless of current input context.

The syntax is

```
XIM XOpenIM(Display *dpy, XrmDataBase db, char *res_name,
            char *res_class);
```

The *res_name* is the resource name of the application, *res_class* is the resource class, and *db* is the resource database that the input method should use for looking up resources private to itself. Any of these can be NULL. The fragment in Example 6-7 shows how easy it is to open an input method.

Example 6-7 Opening an IM

```
XIM im;
im = XOpenIM(dpy, NULL, NULL, NULL);
if (im == NULL)
    exit_with_error();
```

XOpenIM() finds the IM appropriate for the current locale. If **XSupportsLocale()** has returned good status (see "Initialization for Xlib Programming") and **XOpenIM()** fails, something is amiss with the administration of the system.

XSetLocaleModifiers() determines configure locale modifiers. The local host X locale modifiers announcer (the XMODIFIERS environment variable) is appended to the modifier list to provide default values on the locale host. The modifier list argument is a null-terminated string containing zero or more concatenated expressions of this form:

`@`*category=value*

For example, if you want to connect Input Method Server *xwnmo*, set modifiers *_XWNMO* as follows:

```
XSetLocaleModifiers("@im=_XWNMO");
```

Or, set environment variable XMODIFIERS to the string `@im=_XWNMO` and execute

```
XSetLocaleModifiers("");
```

Note: The library routines are not prepared for the possibility of **XSupportsLocale()** succeeding and **XOpenIM()** failing, so it's up to application developers to deal with such an eventuality. (This circumstance could occur, for example, if the IM died after **XSupportsLocale()** was called.) This topic is under some debate in the MIT X consortium. If **XSetLocaleModifiers()** is wrong, **XOpenIM()** will fail.

Most of the complexity associated with IM use comes from configuring an input context to work with the IM. Input contexts are discussed in "Input Contexts (ICs)" on page 257.

To close an input method, call **XCloseIM()**.

IM Styles

If the application requests it, an input method can often supply status information about itself. For example, a Japanese IM may be able to indicate whether it is in Japanese input mode or romaji input mode. An input method can also supply pre-edit information, partial feedback about characters in the process of being composed. The way an IM deals with status and pre-edit information is referred to as an IM style. This section describes styles and their naming.

Root Window

The *Root Window* style has a pre-edit area and a status area in a window owned by the IM as a descendant of the root. The application does not manage the pre-edit data, the pre-edit area, the status data, or the status area. Everything is left to the input method to do in its own window, as illustrated in Figure 6-1.

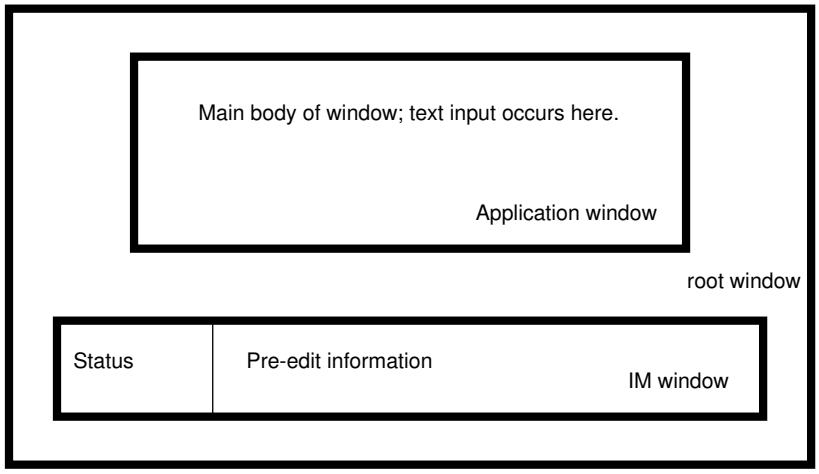

Figure 6-1 Root Window Input

Off-the-Spot

The *Off-the-Spot* style places a pre-edit area and a status area in the window being used, usually in reserved space away from the place where input appears. The application manages the pre-edit area and status area, but allows the IM to update the data there. (The application provides information regarding foreground and background colors, fonts, and so on.) A window using Off-the-Spot input style might look like that shown in Figure 6-2.

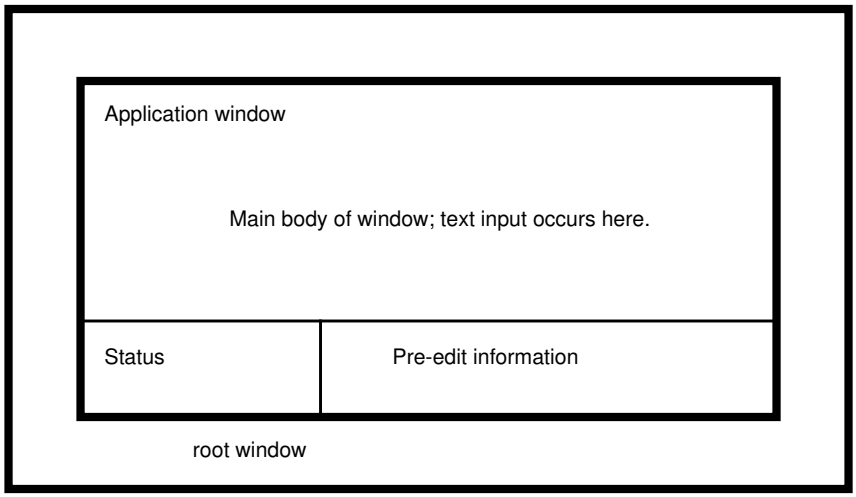

Figure 6-2 Off-the-Spot Input

Over-the-Spot

The *Over-the-Spot* style involves the IM creating a small, pre-edit window over the point of insertion. The window is owned and managed by the IM as a descendant of the root, but it gives the user the impression that input is being entered in the right place; in fact, the pre-edit window often has no borders and is invisible to the user, giving the appearance of On-the-Spot input. The application manages the status area as in Off-the-Spot, but specifies the location of the editing so that the IM can place pre-edit data over that spot.

On-the-Spot

On-the-Spot input is by far the most complex for the application developer. The IM delivers all pre-edit data via callbacks to the application, which must perform in-place editing—complete with insertion and deletion and so on. This approach usually involves a great deal of string and text rendering support at the input generation level, above and beyond the effort required for completed input. Since this may mean a lot of updating of surrounding data or other display management, everything is left to the application. There is little chance an IM could ever know enough about the application to be able to help it provide user feedback. The IM therefore provides status and edit information via callbacks.

Done well, this style can be the most intuitive one for a user.

Setting IM Styles

A style describes how an IM presents its pre-edit and status information to the user. An IM supplies information detailing its presentation capabilities. The information comes in the form of flags combined with OR. The flags to use with each style are as follows:

Root Window	XIMPreeditNothing | XIMStatusNothing
Off-the-Spot	XIMPreeditArea | XIMStatusArea
Over-the-Spot	XIMPreeditPosition | XIMStatusArea
On-the-Spot	XIMPreeditCallbacks | XIMStatusCallbacks

For example, if you wanted a style variable to match an Over-the-Spot IM style, you could write:

```
XIMStyle over = XIMPreeditPosition | XIMStatusArea;
```

If an IM returns *XIMStatusNone* (not to be confused with *XIMStatusNothing*), it means the IM will not supply status information.

Using Styles

An input method supports one or more styles. It's up to the application to find a style that is supported by both the IM and the application. If several exist, the application must choose. If none exist, the application is in trouble.

Input Contexts (ICs)

An input method may be serving multiple clients, or one client with multiple windows, or one client with multiple input styles on one window. The specification of style and client/IM communication is done via *input contexts*. An input context is simply a collection of parameters that together describe how to go about receiving and examining input under a given set of circumstances.

To set up and use an input context:

1. Decide what styles your application can support.
2. Query the IM to find out what styles it supports.
3. Find a match.
4. Determine information that the IC needs in order to work with your application.
5. Create the IC.
6. Employ the IC.

Find an IM Style

The IM may be able to support multiple styles—for example, both Off-the-Spot and Root Window. The application may be able to do, in order of preference, Over-the-Spot, Off-the-Spot, and Root Window. The application should determine that the best match in this case is Off-the-Spot.

First, discover what the IM can do, then set up a variable describing what the application can do, as shown in Example 6-8.

Example 6-8 Finding What a Client Can Do

```
XIMStyles *IMcando;
XIMStyle  clientCanDo; /* note type difference */
XIMStyle  styleWeWillUse = NULL;

XGetImValues(im, XNQueryInputStyle, &IMcando, NULL);

clientCanDo =
/*none*/ XIMPreeditNone | XIMStatusNone |
/*over*/ XIMPreeditPosition | XIMStatusArea |
/*off*/  XIMPreeditArea | XIMStatusArea |
/*root*/ XIMPreeditNothing | XIMStatusNothing;
```

A client should always be able to handle the case of **XIMPreeditNone | XIMStatusNone**, which is likely in a Western locale. To the application, this is not very different from a *RootWindow* style, but it comes with less overhead.

Once you know what the application can handle, look through the IM styles for a match, as shown in Example 6-9.

Example 6-9 Setting the Desired IM Style

```
for(i=0; i < IMcando->count_styles; i++) {
    XIMStyle tmpStyle;
    tmpStyle = IMcando->support_styles[i];

    if ( ((tmpStyle & clientCanDo) == tmpStyle) )
            styleWeWillUse = tmpStyle;
}

if (styleWeWillUse = NULL)
    exit_with_error();
XFree(IMcando);

/* styleWeWillUse is set, which is what we were after */
```

IC Values

There are several pieces of information an input method may require, depending on the input context and style chosen by the application. The input method can acquire any such information it needs from the input context, ignoring any information that does not affect the style or IM.

A full description of every item of information available to the IM is supplied in *X Window System, Third Edition*. The following is a brief list:

XNClientWindow	Specifies to the IM which client window it can display data in or create child windows in. Set once and cannot be changed.
XNFilterEvents	An additional event mask for event selection on the client window.
XNFocusWindow	The window to receive processed (composed) Key events.
XNGeometryCallback	A geometry handler that is called if the client allows an IM to change the geometry of the window.
XNInputStyle	Specifies the style for this IC.
XNResourceClass, *XNResourceName*	The resource class and name to use when the IM looks up resources that vary by IC.
XNStatusAttributes, *XNPreeditAttributes*	The attributes to be used for any status and pre-edit areas (nested, variable-length lists).

Pre-Edit and Status Attributes

When an IM is going to provide state, it needs some simple X information with which to do its work. For example, if an IM is going to draw status information in a client window in an Off-the-Spot style, it needs to know where the area is, what color and font to render text in, and so on. The application gives this data to the IC for use by the IM.

As with the "IC Values" section, full details are available in *X Window System, Third Edition*.

XNArea	A rectangle to be used as a status or pre-edit area.
XNAreaNeeded	The rectangle desired by the attribute writer. Either the application or the IM may provide this information, depending on circumstances.
XNBackgroundPixmap	A pixmap to be used for the background of windows the IM creates.
XNColormap	The colormap to use.
XNCursor	The cursor to use.
XNFontSet	The fontset to use for rendering text.
XNForeground, *XNBackground*	The colors to use for rendering.
XNLineSpacing	The line spacing to be used in the pre-edit window if more than one line is used.
XNSpotLocation	Specifies where the next insertion point is, for use by *XIMPreeditPosition* styles.
XNStdColormap	Specifies that the IM should use **XGetRGBColormaps()** with the supplied property (passed as an Atom) in order to find out which colormap to use.

Creating an Input Context

Creating an input context is a simple matter of calling **XCreateIC()** with a variable-length list of parameters specifying IC values. Example 6-10 shows a simple example that works for the root window.

Example 6-10 Creating an Input Context With XCreateIC()

```
XVaNestedList arglist;
XIC ic;

arglist = XVaCreateNestedList(0, XNFontSet, fontset,
                              XNForeground,
                              WhitePixel(dpy, screen),
                              XNBackground,
                              BlackPixel(dpy, screen),
                              NULL);

ic = XCreateIC(im, XNInputStyle, styleWeWillUse,
               XNClientWindow, window, XNFocusWindow, window,
               XNStatusAttributes, arglist,
               XNPreeditAttributes, arglist, NULL);
XFree(arglist);

if (ic == NULL)
    exit_with_error();
```

Using the IC

A multi-window application may choose to use several input contexts. But for simplicity, assume that the application just wants to get to the internationalized input using one method in one window.

Using the IC is a matter of making sure you check events the IC wants, and of setting IC focus. If you are setting up a window for the first time, you know the event mask you want, and you can use it directly. If you are attaching an IC to a previously configured window, you should query the window and add in the new event mask.

Example 6-11 Using the IC

```
unsigned long imEventMask;

XGetWindowAttributes(dpy, win, &winAtts);
XGetICValues(ic, XNFilterEvents, &imEventMask, NULL);

imEventMask |= winAtts.your_event_mask;
XSelectInput(dpy, window, imEventMask);
XSetICFocus(ic);
```

At this point, the window is ready to be used.

Events Under IM Control

Processing events under input method control is almost the same in X11R6 as it was under R4 and before. There are two essential differences: the **XFilterEvent()** and **X*LookupString()** routines.

Using XFilterEvent()

Every event received by your application should be fed to the IM via **XFilterEvent()**, which returns a value telling you whether or not to disregard the event. IMs asks you to disregard the event if they have extracted the data and plan on giving it to you later, possibly in some other form. All events (not just *KeyPress* and *KeyRelease* events) go to **XFilterEvent()**.

If you compacted the event processing into a single routine, a typical event loop would look something like the code in Example 6-12.

Example 6-12 Event Loop

```
Xevent event;

while (TRUE) {
    XNextEvent(dpy, &event);

    if (XFilterEvent(&event, None))
        continue;

    DealWithEvent(&event);
}
```

Using XLookupString(), XwcLookupString(), and XmbLookupString()

When using an input method, you should replace calls to **XLookupString()** with calls to **XwcLookupString()** or **XmbLookupString()**. The **MB** and **WC** versions have very similar interfaces. The examples below arbitrarily use **XmbLookupString()**, but apply to both versions.

There are two new situations to deal with:

1. The string returned may be long.
2. There may be an interesting keysym returned, an interesting set of characters returned, both, or neither.

Dealing with the former is a matter of maintaining an arena, as in Example 6-13.

To tell the application what to pay attention to for a given event, **XmbLookupString()** returns a status value in a passed parameter, equal to one of the following:

XLookupKeysym	Indicates that the keysym should be checked.
XLookupChars	Indicates that a string has been typed or composed.
XLookupBoth	Means both of the above.
XLookupNone	Means neither is ready for processing.
XBufferOverflow	Means the supplied buffer is too small—call **XmbLookupString()** again with a bigger buffer

XmbLookupString() also returns the length of the string in question. Note that **XmbLookupString()** returns the length of the string in bytes, while **XwcLookupString()** returns the length of the string in characters.

The example below should help show how these functions work. Most event processors perform a switch on the event type; assume you have done that and have received a *KeyPress* event.

Example 6-13 *KeyPress* Event

```
case KeyPress:
{
    Keysym keysym;
    Status status;
    int buflength;
    static int bufsize = 16;
    static char *buf = NULL;

    if (buf == NULL) {
        buf = malloc(bufsize);
        if (buf < 0) StopSequence();
    }

    buflength = XmbLookupString(ic, &event, buf, bufsize,
                            &keysym, &status);

    /* first, check to see if that worked */
    if (status == XBufferOverflow) {
        buf = realloc(buf, (bufsize = buflength));
        buflength = XmbLookupString(ic, &event, buf, bufsize,
                                &keysym, &status);
    }

    /* We have a valid status. Check that */
    switch(status) {
    case XLookupKeysym:
        DealWithKeysym(keysym);
        break;
    case XLookupBoth:
        DealWithKeysym(keysym);
        /* **FALL INTO** charcter case */
    case XLookupChars:
        DealWithString(buf, buflength);
    case XLookupNone:
        break;
    } /* end switch(status) */

} /* end case KeyPress segment */
break; /* we are in a switch(event.type) statement */
```

GUI Concerns

It shouldn't be significantly more difficult to internationalize an application with a graphical user interface than an application without such an interface, but there are a few further issues that must be addressed:

- "X Resources for Strings" covers labeling objects using X resources.
- "Layout" describes creating layouts that are usable after localization.
- "Icons" explains some concerns for localizing icons.

X Resources for Strings

Resource lookup mechanisms in Xlib as well as in toolkits monitor locale environment variables when locating resource files. For string constants that are used within toolkit objects, resources provide a simpler solution than do message catalogs.

These are some common objects that should definitely get their text from resources:

- Labels
- Buttons
- Menu items
- Dialog notices and questions

Any object that employs some sort of text label should be labeled via resources. Since the localizer wants to provide strings for the local version of the application, the *app-defaults* file for the application should specify every reasonable string resource. Reference pages should identify all localizable string resources.

Localizers of an application provide a separate resource file for each locale that the application runs in.

Layout

Layout management is of special interest when you cannot predict how large a button or other label might be. The nature of the problem of layout composition and management does not change, but one must construct the layout management without full knowledge of the final appearance.

It's worth noting that localization efforts can be assumed to be "reasonable" in some sense. For example, X resources have always allowed a user to specify an extremely large font for buttons, but applications correctly choose to let such users live with the results. But it's not always that clear what is reasonable and what isn't; you don't always know what will be difficult to translate succinctly in some locale. So while you need not provide for all combinations of resource specifications, you must make the application localizable.

Three main approaches to the layout problem are described below: dynamic layout, constant layout, and localized layout

Dynamic Layout

Most toolkits provide *form*, *pane*, *rowcolumn*, or other layout objects that calculate layout depending on the "natural" (localized) size of the objects involved. Most use some hints provided by the developer that can regulate this layout. For example, some IRIS IM widgets providing these services are *XmForm*, *XmPanedWindow*, and *XmRowColumn*.

Dynamic layout is probably the simplest way to prevent localization difficulties.

Note: The IRIS IM product is the Silicon Graphics port of the OSF/Motif product, and should not be confused with IM, the abbreviation for Input Methods.

Constant Layout

Under certain circumstances, an application may insist on having a predefined layout. When this is so, the application must provide objects that are constructed to allow localization. A "Quit" button that just barely allows room for the Latin 1 string "Quit" is not likely to suffice when localizers attempt to fit their translations into that small space.

In order to enforce constant layout, the developer incurs the heavy responsibility of making sure the objects are localizable. This means a lot of investigation; the "there, that ought to be enough" approach is chancy at best.

Localized Layout

Some toolkits provide for layout control by run-time reading of strings or other data files. Applications that use such toolkits can easily finesse the layout issue by providing the capability for localization of the layout, as well as localization of the contents of the layout. This provides each localizer maximum freedom in presenting the application to the local users. The application developer is responsible for providing localizers with instructions and the mechanisms necessary to produce layout data.

IRIS IM Localization with editres

IRIX provides an interactive method of laying out widgets for IRIS IM and Xaw (the Athena Widget Set): a utility called *editres*. With *editres*, you can construct and edit resources and see how your widgets will look on the screen; the program even generates a usable app-defaults file for you. But note that if you hard-code any resources into your IRIS IM code, you won't be able to edit them using this method.

Icons

Icons attempt to be fairly generic representations of their antecedents. Unfortunately, it is very difficult for a designer to know what is generic or recognizable in other cultures. Therefore, it is important that any pictographic representations used by an application be localizable.

Graphic representations can be stored as strings representing X bitmaps, as names of data files containing pictographs, or in whatever manner the developer thinks best, so long as the developer provides a way for the localizer to produce and deliver localized pictographs.

Popular Encodings

This section discusses three encodings that are commonly used:

- "The ISO 8859 Family" explains the ISO 8859 family of encodings.
- "Asian Languages" describes Asian language encodings.
- "ISO 10646 and Unicode" covers the ISO 10646 and Unicode.

The ISO 8859 Family

American English is easily representable in 7-bit ASCII. Most other languages are not. For example, the character é is not in ASCII.

Most Western European languages are representable in 8-bit ISO 8859-1, which is commonly known as Latin 1. Latin 1 is a superset of ASCII that includes characters used by several Western European languages (such as ö, £, ñ, ç, ¿).

ISO 8859 comes in nine parts, many of which overlap; all are supersets of ASCII.

The ISO 8859 Character Sets are shown in Table 6-10.

Table 6-10 ISO 8859 Character Sets

Character Set	Common Name	Languages Supported
8859-1	Latin 1	Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish
8859-2	Latin 2	Albanian, Czech, English, German, Hungarian, Polish, Rumanian, Serbo-Croatian, Slovak, Slovene
8859-3	Latin 3	Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish
8859-4	Latin 4	Danish, English, Estonian, Finnish, German, Greenlandic, Lapp, Latvian, Lithuanian, Norwegian, Swedish
8859-5	Latin/Cyrillic	Bulgarian, Byelorussian, English, Macedonian, Russian, Serbo-Croatian, Ukrainian
8859-6	Latin/Arabic	Arabic, English (see ISO 8859-6 specification)
8859-7	Latin/Greek	English, Greek (see ISO 8859-7 specification)
8859-8	Latin/Hebrew	English, Hebrew (see ISO 8859-8 specification)
8859-9	Latin 5	Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish

IRIX contains over 500 Latin 1 fonts, as well as a few fonts for each of the other 8859-encoded character sets, except 8859-6 and 8859-8. Currently, IRIX contains no fonts for use with the 8859-6 or 8859-8 character sets.

To get the list of ISO-8859 fonts, enter the following:

```
xlsfonts
```

Or you can restrict the amount of output, for example, by typing

```
xlsfonts '*8859-2'
```

To see the encoding, use the *xfd* command. For example:

```
xfd -fn -sgi-screen-medium-r-normal--9-90-72-72-m-60-iso8859-1
```

For more information on *xlsfonts* and *xfd*, and installing and using fonts, refer to Chapter 5, "Working With Fonts."

Asian Languages

Asian languages are commonly ideographic and employ large numbers of characters for their representation. For example, Japanese and Korean can be practically encoded in 16 bits. Daily-use Chinese can be, also, but archives and scholars frequently need more, so Chinese is often encoded with up to four bytes per character.

Some Standards

Various Asian character sets have been developed, some of which are considered standard. Encodings for these sets are less standardized. Asian character sets usually require larger-than-byte character types like those described in "Multibyte Characters." Table 6-11 lists some of these standard character sets. Note that some of these character sets have multiple associated codesets, usually designated by appending the year the codeset was adopted to the character set name. (For example, JIS X 208-1983 is different from JIS X 208-1990.)

Table 6-11 Character Sets for Asian Languages

Language	Character Set Standards	Support
Japanese	JIS X 0201.1976-0	*Katakana*
	JIS X 0208.1983-0	*Kanji*, *kana*, Latin, Greek, Cyrillic, symbols, others
	JIS X 0212.1990-0	Supplemental *kanji*, others
Chinese	GB 2312.1980-0	
Korean	KSC 5601.1987-0	Hangul
Taiwan	CNS 11643	

EUC

EUC is *Extended UNIX Code*, an encoding methodology that supports concurrent use of four codesets in one encoding. It employs two special "shift state" bytes:

```
ss1 = 0x8e
ss2 = 0x8f
```

These are used to identify codesets within a string. The EUC encoding scheme uses the following patterns to indicate which codeset is in use at any given time:

```
Codeset #0: 0xxxxxxx
Codeset #1: 1xxxxxxx [ 1xxxxxxx ...]
Codeset #2: ss1 1xxxxxxx [ 1xxxxxxx ...]
Codeset #3: ss2 1xxxxxxx [ 1xxxxxxx ...]
```

So if *ss1* appears in a string, it means that the next character—however many bytes long it is—should be interpreted as a character from codeset #2. If there are multiple characters in a row from codeset #2, each one is preceded by *ss1*. Similarly, *ss2* indicates that the following character belongs to codeset #3. If any other byte whose high bit is 1 appears in the string (without being preceded by *ss1* or *ss2*), it is interpreted as all or part of a character from codeset #1.

In EUC, codeset #1 is always ASCII. The other codesets are implementation- or user-defined. This is why EUC cannot support Latin 1 in Asian locales.

EUC implementations exist (but are not standardized) for all ideographic Asian languages.

ISO 10646 and Unicode

ISO and the Unicode Consortium have jointly developed a character set designed to cover almost every character normally used by any language in the world. The characters have two- and four-byte representations. ISO calls this *ISO IS 10646*. The Unicode Consortium embraces a subset of 10646, called the *Basic Multilingual Plane* (BMP) of 10646, and calls it *Unicode*. The only characters defined in either standard are the characters in the BMP.

It appears that ISO 10646 will grow significantly in acceptance, but widespread use is still some years away.