Developer's Guide to Internationalization

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Tables

Preface

This book is intended for people who need to write internationalized software for the Solaris environment. Knowledge of the C programming language is assumed. All operating system information pertains to SunOS, while all window system information pertains to OpenWindows 3.3.

Book Organization

The material in this book is organized as follows:

- Chapter 1 introduces key notions of internationalization and localization.
- Chapter 2 describes some of the cultural differences that global software design must resolve.
- Chapter 3 describes internationalization features that the operating system offers to users and application developers.
- Chapter 4 describes correct international coding practices and provides C language examples.
- Chapter 5 discusses writing internationalized window system code.
- Chapter 6 describes how to create and install translated user messages for internationalized applications.
- Appendix A lists accepted locale names for language and territory.
- Appendix B illustrates localized keyboard layouts.
- Appendix C is a glossary of special terms used throughout the book.

Conventions

The following conventions are used in the procedures and examples throughout this guide.

- System prompts and error messages are printed in `listing font`.
- Information you type as a command or in response to prompts is shown in **`boldface listing font`**. Type everything shown in boldface exactly as it appears in the text.
- Parts of a command shown in *italic text* refer to a variable that you have to substitute from a selection. It is up to you to make the correct substitution.
- Dialogues between you and the system are enclosed in boxes:

```
$ pwd
/home/machine/scotty
$
```

- Sections of program code are enclosed in boxes:

```
#include <stdio.h>
main()
{
    printf(“Hello world!\n”);
    exit(0)
```

- Control characters are shown by the word “Control-” followed by the appropriate character, such as Control-D. To enter a control character, hold down the key marked Control and press the appropriate key.
- The standard prompt signs are the dollar sign ($) or percent sign (%) for ordinary users, and the sharp (#) when a command must be executed by `root` or the superuser.
- When commands are mentioned in the text for the first time, a reference to the manual page for the command is often given with the section number in parentheses: *command*(section). For example, `grep`(1) is the `grep` command described in section 1 of the manual pages.

Introduction 1

What Is Internationalization?

Internationalization is a way of designing and producing software that can easily be adapted to local markets. Internationalized products can be **localized** or adapted to different languages and cultures with minimal effort.

How Is Localization Different?

Internationalization is the process of making software portable between languages or regions, while localization is the process of adapting software for specific languages or regions. International software can be developed using interfaces that modify program behavior at run time in accordance with specific cultural requirements. Localization involves establishing on-line information to support a language or region, called a *locale*.

Unlike software that must be completely rewritten before it can work with different native languages and customs, internationalized software does not require rewriting. It can be ported from one locale to another without change. Solaris is internationalized, providing the infrastructure and interfaces you need to create internationalized software. Chapters 3 and 4 describe what facilities are available and how to use them.

Basic Steps in Internationalization

An internationalized application's executable image is portable between languages and regions. To internationalize software, you:

- Use the interfaces described in this book to create software whose environment can be modified dynamically without the software needing to be recompiled.
- Separate all printable and displayable messages that the user sees from the executable image. Keep these message strings in a message database.

Message strings are translated for a language and region (called a locale) as part of the localization process. Related databases that specify formats for time, currency, and numbers are translated at the same time.

To use a localized version of a product, users set an environment variable. The product then displays user messages in their translated form, and also formats date, time, and currency according to the locale-specific conventions. Thus, users gain control of their software's language and behavior.

Advantages of Internationalization

Creating internationalized software automatically expands the market for your product. If you follow the steps outlined in this book, your product will be compatible with all the languages and cultures supported by Solaris.

Many computer firms, including Sun Microsystems and Digital Equipment, obtain around half of their revenue from outside the United States. Profit margins are often higher abroad, so contribution to net income can be even higher. Localization costs can be high, particularly for translation, but often repay themselves quickly in higher sales.

Conforming to Standards

Many standards bodies are developing guidelines for internationalized software. Practices described in this book conform to various ANSI, IEEE, ISO, and X/Open standards. International standards are still evolving, so not all interfaces can be guaranteed forever.

Internationalization Levels

SunSoft defines four levels of internationalization, described below. Levels are not necessarily hiearchical, but do indicate difficulty of implementation.

Level 1—Text and Codesets

Software that is level-1 compliant is "8-bit clean" and can therefore use the ISO 8859-1 (also called ISO Latin-1) codeset. Historically, many programmers assumed their application needed only the ASCII character set. Because the ASCII codeset employs only seven bits out of an 8-bit byte, the most significant bit was often used to store information about the character. For example, setting the most significant bit "on" might indicate that the character is highlighted. The ISO Latin-1 codeset employs all eight bits. Software that uses the most significant bit for its own purposes is not level-1 compliant.

Level 2—Formats and Collation

Software is level-2 compliant if its formatting and collation methods are locale sensitive. Many different formats are employed throughout the world to represent date, time, currency, numbers, and units. Also, some alphabets have more letters than others, and the order in which letters are sorted within national alphabets varies from language to language. Programs that leave the format design and sorting order to the localization center in a particular country are considered level-2 compliant.

Level 3—Messages and Text Presentation

User-visible text in a level-3 compliant application should be easily translatable into the languages of various target markets. User-visible text includes help text, error messages, property sheets, buttons, text on icons, and so forth. A common way to provide message translation is to put text to be translated into a separate file with messages indexed either by string contents (the POSIX/UniForum method) or by number (the XPG-3 method). Software using separate message files for messaging—rather than encapsulating user messages in the binary—is considered level-3 compliant

Solaris provides two different but incompatible methods of translating text. The first and preferred method is the POSIX and UniForum proposed standard using the `gettext()` function.

The second method of message translating is XPG-3 style *message catalogs*. XPG is the X/Open Portability Guide. XPG-3 provides the `catgets()` function to obtain translated message strings from a message database. The XPG message translation standard is not recommended and is not used in Solaris.

Whichever mechanism is used for messages translation—and only one method should be used in any one application— level-3 compliant software should strive to avoid compound messages (messages consisting of separately composed parts) because word order is different in many languages.

Level 4—Asian Language Support

Level-4 compliant software provides support for East Asian languages, which often require multi-byte codesets because of their large character inventory. Solaris provides such support with the Extended Unix Code (EUC). This is a method for switching between multiple codesets, three of which may in turn be multi-byte codesets.

Formats and Conventions Overview 2

Different countries in the world use completely different conventions for writing date, time, numbers, currency, delimiting words and phrases, and quoting material. You should not code these conventions directly into your application; an internationalized product in conjunction with a localization package will be sensitive to the appropriate formats. Chapters 4 and 5 provide guidance on coding practices used to achieve these formats.

Formatting Differences

Time Formats

The following table shows some of the ways to write 11:59 p.m.

Table 2-1 International Time Formats

Locale	**Format**
Canadian	23:59
Finnish	23.59
German	23.59 Uhr
Norwegian	Kl 23.59
U.K.	11:59 PM

Time is represented by both a 12-hour clock and a 24-hour clock—sometimes known as ''railroad time''. The hour and minute separator can be either a colon (:) or a period (.). Some countries attach letters to the time indicating that this is a time, but these are not strictly necessary.

Time zone splits occur between and within countries. Although a time zone can be described in terms of how many hours it is ahead of or behind Greenwich Mean Time (GMT), this number is not always an integer. For example, Newfoundland is in a time zone that is half an hour different from the adjacent time zone.

Daylight Savings Time (DST) starts and ends on different dates that can vary from country to country.

Date Formats

This table shows some of the date formats used around the world. Of course, there is a good deal of variation even within countries, so these formats are not the final truth.

Table 2-2 International Date Formats

Locale	**Convention**	**Example**
Canadian (English)	yyyy-mm-dd	1989-08-13
Canadian (French)	yyyy-mm-dd	1989-08-13
Danish	dd/mm/yy	13/08/89
Finnish	dd.mm.yyyy	13.08.1989
French	dd/mm/yy	13/08/89
German	dd.mm.yy	13.08.89
Italian	dd.mm.yy	13.08.89
Norwegian	dd.mm.yy	13.08.89
Spanish	dd-mm-yy	13-08-89
Swedish	yyyy-mm-dd	1989-08-13
UK-English	dd/mm/yy	13/08/89
US-English	mm-dd-yy	08-13-89

Numbers

Decimal and Thousands Separators

The United Kingdom and the United States are two of the few places in the world that use a period to indicate the decimal place. Many other countries use a comma instead. The decimal separator is also called the *radix* character. Likewise, while the UK and US use a comma to separate thousands groups, many other countries use a period for this instead, and some countries separate thousands groups with a thin space. This table shows some commonly used numeric formats.

Table 2-3 International Numeric Conventions

Locale	**Large Number**
Canadian (French)	4 294 967 295,00
Canadian (English)	4 294 967 295,00
Danish	4.294.967.295,00
Finnish	4.294.967.295,00
French	4.294.967.295,00
German	4 294 967 295,00
Italian	4.294.967.295,00
Norwegian	4.294.967.295,00
Spanish	4.294.967.295,00
Swedish	4.294.967.295,00
UK-English	4,294,967,295.00
US-English	4,294,967,295.00

Data files containing locale-specific formats will be misinterpreted when transferred to a system in a different locale. For example, a file containing numbers in a French format will not be useful to a UK-specific program.

List Separators

There are no particular locale conventions that specify how to separate numbers in a list. They are sometimes comma-delimited in the UK and the US, but often spaces and semicolons are used. Certainly, international software never “hardwires” any number delimiter.

Currency

Currency units and presentation order vary greatly around the world. This table shows monetary formats in some countries.

Table 2-4 International Monetary Conventions

Locale	**Currency**	**Example**
Canadian (English)	Dollar ($)	$1 234.56
Canadian (French)	Dollar ($)	1 234.56$
Danish	Kroner (kr)	kr.1.234,56
Finnish	Markka (mk)	1.234 mk
French	Franc (F)	F1.234,56
German	Deutsche Mark (DM)	1,234.56DM
Italian	Lira (L)	L1.234,56
Japanese	Yen (¥)	¥1,234
Norwegian	Krone (kr)	kr 1.234,56
Spanish	Peseta (Pts)	1.234,56Pts
Swedish	Krona (Kr)	1234.56KR
UK-English	Pound (£)	£1,234.56
US-English	Dollar ($)	$1,234.56

Note that local and international symbols for currency can differ. For example, the designation for the French Franc is “F” in France but this is often written as ‘‘FRF’’ internationally to distinguish it from other Francs, such as the Swiss Franc or the Polynesian Francs.

Be aware also that a *converted* currency amount may take up more or less space than the original amount. To illustrate: $1,000 can become L1.307.000.

Word and Letter Differences

Word Delimiters

Usually, words are separated by a space character. In Japanese and Thai, however, there is often no delimiter between words.

Word Order

The order of words in phrases and sentences varies between languages. For instance, the order of the words ‘‘cat’’ and ‘‘black’’ in ‘‘a black cat’’ is reversed in the equivalent Spanish phrase, ‘‘uno gato negro’’. And in French, the negatives ‘‘ne’’ and ‘‘pas’’ surround the word they negate, as in the phrase ‘‘I do not speak,’’ which in French is ‘‘Je ne parle pas’’.

Sort Order

Sorting order for particular characters is not the same in all languages. For example, the character “ö” sorts with the ordinary “o” in Germany, but sorts separately in Sweden, where it is the last letter of the alphabet.

Character Sets

Number of Characters

While the English alphabet contains only 26 characters, some languages contain many more characters. Japanese, for example, can contain over 40,000 characters; Chinese even more.

Western European Alphabets

The alphabets of most western European countries are similar to the standard 26-character alphabet used in English-speaking countries, but there are often some additional basic characters, some marked (or accented) characters, and some ligatures.

Japanese

Japanese text is composed of three different scripts mixed together: Kanji ideographs derived from Chinese, and two phonetic scripts (or syllabaries), Hiragana and Katakana.

Although each character in Hiragana has an equivalent in Katakana, Hiragana is the most common script, with cursive rather than block-like letter forms. Kanji characters are used to write root words. Katakana is mostly used to represent ''foreign'' words—words ''imported'' from languages other than Japanese.

There are tens of thousands of Kanji characters, but the number commonly used has been declining steadily over the years. Now only about 3500 are frequently used, although the average Japanese writer has a vocabulary of merely 2000 Kanji characters. Nonetheless, computer systems must support more than 7000 because that is what the Japan Industry Standard (JIS) requires. In addition, there are about 170 Hiragana and Katakana characters. On average 55% of Japanese text is Hiragana, 35% Kanji, and 10% Katakana. Arabic numerals and Roman letters are also present in Japanese text.

Although it is possible to avoid the use of Kanji completely, most Japanese readers find text containing Kanji easier to understand.

Korean

Korean is similar to Japanese in that Chinese-based ideograms, called Hanja, are mixed together with a phonetic alphabet, Hangul. Hanja is used mostly to avoid confusion when Hangul would be ambiguous.

Hangul characters are formed by combining ten basic vowels and fourteen consonants, two to five of which compose one syllable. Hangul characters are often arranged in a square like the four on a pair of dice, so that the group takes up the same space as a Hanja character.

Korean requires over 6000 Hanja characters, plus about 96 Hangul characters.

Chinese

Chinese usually consists entirely of characters from the ideographic script called Hanzi. In the People's Republic of China (PRC) there are about 7000 commonly used Hanzi characters, although emerging standards number Hanzi in the tens of thousands. In the Republic of China (ROC or Taiwan) current standards require more than 13,000 characters; 6000 others have been recently standardized but are considered rare.

If a character is not a root character, it usually consists of two or more parts, two being most common. In two-part characters, one part generally represents meaning, and the other represents pronunciation. Occasionally both parts represent meaning. The radical is the most important element, and characters are traditionally arranged by radical, of which there are several hundred. The same sound can be represented by many different characters, which are not interchangeable in usage.

Some characters are more appropriate than others in a given context—the appropriate one is distinguished phonetically by the use of tones. By contrast, spoken Japanese and Korean lack tones.

There are several phonetic systems for representing Chinese. In mainland China the most common is pinyin, which uses Roman characters and is widely employed in the west for place names such as Beijing. The Wade-Giles system is an older phonetic system, formerly used for place names such as Peking. In Taiwan zhuyin (or bopomofo), an extensive phonetic alphabet with unique letter forms, is often used instead.

Commercial applications, particularly those that deal with people's names, need to consider the impact of code set expansion. Many people in the ROC have names containing characters that do not exist in any standard code set. Space needs to be provided in unassigned code sets to deal with this issue.

Codesets for x86

The default codeset on Solaris for x86 is ISO-8859-1. IBM® DOS 437 codeset is provided as an option in text mode; however, it is only provided at internationalization level 1. That is, if you choose to download IBM DOS 437 codeset by typing :

```
loadfont -c 437
```

```
pcmapkeys -f /usr/share/lib/keyboards/437/en_US
```

there will be no support for non-standard U.S. date, time, currency, numbers, units, and collation. There will be no support for non-English message and text presentation, and no multi-byte character support. Therefore, non-Windows users should only use IBM DOS 437 codeset in the default C locale.

- You must be in the text mode to download the IBM codeset, not the graphics mode.
- If you are not using the standard U.S. PC keyboard, replace `en_US` with the keyboard map related to your keyboard.
- To download the default codeset in text mode, type:

```
loadfont -c 8859
pcmapkeys -f /usr/share/lib/keyboards/8859/en_US
```

- See the `loadfont` **(1)** and `pcmapkeys` **(1)** manual pages.

Keyboard Differences

Not all characters on the US keyboard appear on other keyboards. Similarly, other keyboards often contain many characters not visible on the US keyboard. However, the Compose key can be used to produce any character in the ISO Latin-1 code set on any keyboard that supports it. See Appendix B for a list of different keyboard layouts.

Other Differences

Punctuation

Both the position and the type of punctuation symbols can vary between languages. In Spanish, “¿” and “¡” appear at the beginnings of sentences, while in Finnish colons (:) can occur inside words.

Symbols

Commonly used symbols in one culture often have no meaning in another culture. Because the common US rural mailbox, for example, does not exist in other countries, it would not make a universal MailTool icon.

Measurements

While most countries now use the metric system of measurement, the United States, parts of Canada, and the United Kingdom (albeit unofficially) still use the imperial system. The symbols for feet (‘) and inches (“) are not understood in all countries.

Gender

The spelling of adjectives, articles, and nouns are gender-dependent in some languages. In French, for example, “un petit gamin” and “une petite gamine” both mean “a cute kid”. The first expression, however, refers to a boy, and the second expression, to a girl. Also, neuter objects in English (“a computer” for example) have gender in other languages (“un ordinateur” is a masculine noun in French).

Titles and Addresses

Mr., Miss, Mrs., and Ms. are common titles in the US but are not used in many other countries.

Address formats differ from country to country. In many countries, the postal code includes letters as well as numbers.

Paper Sizes

Within each country a small number of paper sizes are commonly used, normally with one of those sizes being much more common than the others. Most countries follow ISO Standard 216 ‘‘Writing paper and certain classes of printed matter—Trimmed sizes—A and B series.’’

Internationalized applications should not make assumptions about the page sizes available to them. Solaris provides no support for tracking output page size; this is the responsibility of the application program itself.

Table 2-5 Common International Page Sizes

Paper Type	**Dimensions**	**Countries**
ISO A4	21.0 cm by 29.7 cm	Everywhere except US
ISO A5	14.8 cm by 21.0 cm	Everywhere except US
JIS B4	25.9 cm by 36.65 cm	Japan
JIS B5	18.36 cm by 25.9 cm	Japan
US Letter	8.5 inch by 11 inches	US and Canada
US Legal	8.5 inch by 14 inches	US and Canada

Standard paper trays distributed with LaserWriter and LaserWriter II support US letter, US legal, and A4 paper sizes. The SPARCprinter's paper tray supports all these, in addition to B5.

Summary

International code does not contain any implicit cultural assumptions. Instead, it uses standardized interfaces that make use of installed localization packages. Read Chapters 4 and 5 to learn what these standard interfaces are and how you can use them to create internationalized products.

Support for Internationalization 3

This chapter describes features of Solaris that provide the foundation for internationalization support. Chapter 4 gives examples of international coding practices, and Chapter 5 discusses window system specifics.

Here are some of the ways that the SunOS operating system supports international software applications:

- Data paths are 8-bit clean in order to support ISO 8859 code sets, and so that multi-byte characters can survive intact.
- Keyboard drivers and mapping tables are provided for a variety of code sets. This allows software to cope with many European and Asian languages.
- Printers, modems, and terminals are supported that can handle ASCII, ISO Latin-1, and EUC code sets.
- System locales are included for French, German, Italian, Swedish, and Japanese, offering level 2 internationalization for these locales.
- Standard C library routines provide support for writing software that can be easily localized.
- Wide character library routines provide programming support for Asian language applications.
- Applications can use either the X/Open or the `gettext()` message system for access to text that needs translation.

Note that only dynamically linked libraries provide international support. Many of the features above will not work for statically linked programs.

3

Keyboards and Peripherals

Keyboards

The Type-5 and Type-4 keyboards are currently available in 18 versions: 15 for Roman alphabets and three for Asian languages:

Table 3-1 Available SPARC Keyboards

Belgium / France	Canada	Canada (French)
Denmark	Germany	Italy
Netherlands	Norway	Portugal
Spain	Sweden / Finland	Switzerland (French)
Switzerland (German)	United Kingdom	United States
Japan	Korea	Taiwan

International keyboards are normally delivered with country kits, but are also available separately.

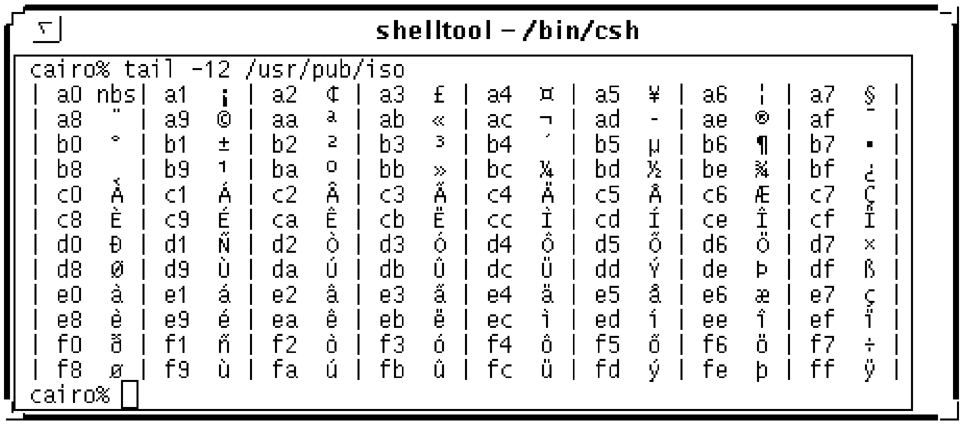
The PC-AT101 and PC-AT102 keyboards are currently available in:

Table 3-2 Available PC Keyboards

Belgium / France	Canada	Denmark
Germany	Italy	Netherlands
Norway	Portugal	Spain
Sweden / Finland	Switzerland (French)	Switzerland (German)
United Kingdom	United States	

The lower half of the ISO Latin-1 code set contains all characters from the ASCII code set, while the upper half includes accented and special characters, shown in the figure below.

Figure 3-1 Upper Half of ISO 8859-1



shelltool – /bin/csh

```
cairo% tail -12 /usr/pub/iso
| a0 nbs| a1  ¡ | a2  ¢ | a3  £ | a4  ¤ | a5  ¥ | a6  ¦ | a7  § |
| a8  ¨ | a9  © | aa  ª | ab  « | ac  ¬ | ad  - | ae  ® | af  ¯ |
| b0  ° | b1  ± | b2  ² | b3  ³ | b4  ´ | b5  µ | b6  ¶ | b7  · |
| b8  ¸ | b9  ¹ | ba  º | bb  » | bc  ¼ | bd  ½ | be  ¾ | bf  ¿ |
| c0  À | c1  Á | c2  Â | c3  Ã | c4  Ä | c5  Å | c6  Æ | c7  Ç |
| c8  È | c9  É | ca  Ê | cb  Ë | cc  Ì | cd  Í | ce  Î | cf  Ï |
| d0  Ð | d1  Ñ | d2  Ò | d3  Ó | d4  Ô | d5  Õ | d6  Ö | d7  × |
| d8  Ø | d9  Ù | da  Ú | db  Û | dc  Ü | dd  Ý | de  Þ | df  ß |
| e0  à | e1  á | e2  â | e3  ã | e4  ä | e5  å | e6  æ | e7  ç |
| e8  è | e9  é | ea  ê | eb  ë | ec  ì | ed  í | ee  î | ef  ï |
| f0  ð | f1  ñ | f2  ò | f3  ó | f4  ô | f5  õ | f6  ö | f7  ÷ |
| f8  ø | f9  ù | fa  ú | fb  û | fc  ü | fd  ý | fe  þ | ff  ÿ |
cairo%
```

The PROM monitor and OpenWindows 3.3 provide fonts for western Europe, standardized as ISO 8859-1, also called ISO Latin-1. These ISO fonts use the full 8-bit address space of a byte. The system is supposed to come up in 8-bit mode by default, but in case it does not, simply type `stty cs8 -istrip` in a shell or command window.

When SunOS boots on a SPARC system, it automatically recognizes the keyboard type. If you plug an alternate keyboard into a running system, SPARC hardware generates an interrupt and returns you to the PROM monitor. After the PROM monitor's > or `ok` prompt, simply type `c` or `go` for "continue". Then choose "Refresh" from the OpenWindows Workspace Utilities menu, and execute the `loadkeys` command in a shell or command window.

When SunOS boots on an x86 system, the system is supposed to come up in 8-bit mode by default. If it does not, simply type `stty pass8` in a shell or command window.

Native Language Keyboards

The SPARC Type-5 and Type-4 keyboards generate 8-bit characters, or events. System translation tables generate appropriate character codes based on these events. The SunOS translation tables are in `/usr/share/lib/keytables`. OpenWindows keyboard tables are in `$OPENWINHOME/etc/keytables`.

The PC-AT101 and PC-AT102 keyboards generate 8-bit characters, or events. System translation tables generate appropriate character codes based on these events. The SunOS translation tables are in `/usr/share/lib/keyboards`. OpenWindows keyboard tables are in `$OPENWINHOME/etc/keytables`.

If at all possible, use the operating system or window system's keyboard translation tables. Applications that read events directly from the keyboard must perform their own key mappings, which isn't easy. Moreover, such applications need to be revised every time new keyboards are devised.

Generating Characters Not on a U.S. Keyboard

Although non-English characters like the German ä or the French ê are not present on a keyboard designed for use in American English, most of these characters can be generated. This allows users to write French letters on American systems, for example. There are three ways to generate characters for which there are no keycaps (explicit symbols on the keyboard):

- Deadkeys (x86 systems only)
- Compose sequences (SPARC and x86 systems)
- The decimal representation of the character (x86 systems only)

Deadkeys (x86)

The *deadkey* was invented by typewriter manufacturers. For example, imagine you need the French character ê. A French typewriter does not have a key for this character, but it has keys for both e and ^. When the key `^` is pressed, a circumflex is printed but the typewriter carriage does not move. When the `e` key is then pressed, the letter "e" is printed on the same spot as the circumflex and an ê is formed. This technique works very similarly on a terminal. The only difference is that when `^` is pressed, *nothing* happens until `e` is pressed, after which the character ê appears on the screen.

In text mode, a utility that can be used to assign deadkeys, `pcmapkeys`, is supplied. This utility is used to do everything discussed in this section. To define `^` as a deadkey and try the other examples listed below, type the command:

```
pcmapkeys -f /usr/share/lib/keyboards/dead/circumflex
```

In OpenWindows, the utility `Xmodmap` may be used to remap keys. See the `Xmodmap` (1) manual page for additional information. Now when you press ^, nothing appears on the screen. When an e is typed next, the letter ê appears. To use the ^ character alone, press ^ first and then the spacebar. If a sequence of two characters is typed that does not make sense, no character is sent to the application that is currently being used, and the machine beeps to indicate that an erroneous combination was typed.

Using the Compose Key (SPARC and x86)

Compose Key on SPARC

The regular SPARC Type-5 or Type-4 keyboard can produce all characters in the standard ISO 8859-1 (Latin-1) code set by means of the Compose key. Such characters are typically composite characters that include diacritical marks. To produce a composite character on the US-English keyboard, first press the Compose key. Next, press the key for the desired diacritical mark, and then the key for the desired alphabetical character. You may type the diacritical mark and the alphabetical character in either order.

For example, to produce **à** press the Compose key, then type `a` and `‘`(order doesn’t matter). When testing software, make sure to try all these combinations on your keyboard. If your software manages the keyboard directly, you should also try all special keys on the 13 European SPARC keyboards and perhaps the Asian SPARC keyboards as well.

Compose Key on x86

On x86 systems, the default COMPOSE key sequence for Solaris for x86 is CTRL SHIFT F1. (Many MS-DOS® (DOS) users will be familiar with it.) When in COMPOSE mode, the system expects two more characters to be typed by the user to generate a character. Press CTRL SHIFT F1 followed by `n ~` to produce the Spanish ñ (the n in mañana) on the screen. If you press the COMPOSE key sequence followed by pressing `!` twice, an inverted exclamation sign appears on the screen.

In text mode, both the value of the COMPOSE key and the list of COMPOSE key sequences and the characters they generate can be specified in a file that is then processed by the `pcmapkeys` command (see `pcmapkeys` (1)). In text mode, the following tables are only valid for the ISO-8859-1 codeset and not for the optional IBM DOS 437 codeset. In OpenWindows, these tables are valid because OpenWindows only supports the ISO-8859-1 codeset.

Compose Key Sequences

Here is a table shows how to produce special ISO Latin-1 characters using Compose key sequences. .

Table 3-3 Compose Key Sequences

Compose Key	**Sequence**	**Result**	**Description**
space	space		no-break space
!	!	¡	inverted exclamation
c	/	¢	cents
l	-	£	pounds sterling
o	x	¤	currency symbol
y	-	¥	yen
|	|	¦	broken bar
s	o	§	section
“	“	¨	umlaut/diaeresis
c	o	©	copyright
-	a	ª	feminine ordinal
<	<	«	left guillemet
-	|	¬	not sign
-	-	-	soft hyphen
r	o	®	registered

Table 3-3 Compose Key Sequences

Compose Key	**Sequence**	**Result**	**Description**
^	-	¯	macron
^	0	°	degree
+	-	±	plus-minus
^	2	^{2}	superscript 2
^	3	^{3}	superscript 3
\	\	´	prime / acute accent
/	u	µ	mu / micro
P	!	¶	pilcro / paragraph
,	,	¸	cedilla
^	.	·	middle dot
^	1	^{1}	superscript 1
_	o	º	masculine ordinal
>	>	»	right guillemet
1	4	$^{1}/_{4}$	quarter
1	2	$^{1}/_{2}$	half
3	4	$^{3}/_{4}$	three quarters
?	?	¿	inverted question
A	`	À	A grave
A	‘	Á	A acute
A	^	Â	A circumflex
A	~	Ã	A tilde
A	“	Ä	A umlaut
A	*	Å	A angstrom
A	E	Æ	AE ligature
C	,	Ç	C cedilla
E	`	È	E grave

Table 3-3 Compose Key Sequences

Compose Key	Sequence	Result	Description
E	‘	É	E acute
E	^	Ê	E circumflex
E	“	Ë	E umlaut
I	`	Ì	I grave
I	‘	Í	I acute
I	^	Î	I circumflex
I	“	Ï	I umlaut
D	-	Ð	Eth
N	~	Ñ	N tilde
O	`	Ò	O grave
O	‘	Ó	O acute
O	^	Ô	O circumflex
O	~	Õ	O tilde
O	“	Ö	O umlaut
x	x	×	multiply
O	/	Ø	O slash
U	`	Ù	U grave
U	‘	Ú	U acute
U	^	Û	U circumflex
U	“	Ü	U umlaut
Y	‘	Ý	Y acute
T	H	Þ	Thorn
s	s	ß	ess zed / digraph s
a	`	à	a grave
a	‘	á	a acute
a	^	â	a circumflex

Table 3-3 Compose Key Sequences

Compose Key	**Sequence**	**Result**	**Description**
a	~	ã	a tilde
a	“	ä	a umlaut
a	*	å	a angstrom
a	e	æ	ae ligature
c	,	ç	c cedilla
e	`	è	e grave
e	‘	é	e acute
e	^	ê	e circumflex
e	“	ë	e umlaut
i	`	ì	i grave
i	‘	í	i acute
i	^	î	i circumflex
i	“	ï	i umlaut
d	-	ð	eth
n	~	ñ	n tilde
o	`	ò	o grave
o	‘	ó	o acute
o	^	ô	o circumflex
o	~	õ	o tilde
o	“	ö	o umlaut
-	:	÷	divide
o	/	ø	o slash
u	`	ù	u grave
u	‘	ú	u acute
u	^	û	u circumflex
u	“	ü	u umlaut

Table 3-3 Compose Key Sequences

Compose Key	Sequence	Result	Description
y	‘	ý	y acute
t	h	Þ	thorn
y	“	ÿ	y umlaut

Decimal Representation

A third method of generating characters is using their decimal representation. Every character corresponds to a unique number. Up to 256 different characters can be used (although some terminals only support 128). When the COMPOSE key is used, followed by three digits, the character that is internally represented by the three-digit number (in decimal) is generated. This feature is also derived from the DOS system. Press the COMPOSE key sequence, followed by `065`, and an `A` appears on the screen. 65 is the decimal value used by computers to store the uppercase letter A. Press the COMPOSE key sequence followed by `136` and the letter ê appears. If you type:

```
pcmapkeys -d
```

all deadkeys and compose sequences are disabled.

Using the Floating Accent Keys

On some keyboards, certain keys appear with an empty box (❑) underneath the diacritical mark. These are referred to as *floating accent* keys. They allow you to type in a composite character without using the Compose key. Type the floating accent key first, followed by the key for the letter to be accented.

Modems

Modems set to 8-bit no-parity mode will work with 8-bit data.

Dumb Terminals

Dumb terminals set to 8-bit space-parity mode will work with 8-bit data, although they are unlikely to display the proper characters.

Printers

Support in the SunOS system for native language printing includes:

- Transmission of 8-bit characters by `lp`. The serial line must also be 8-bit clean, and the printer must support the ISO Latin-1 character set, for the characters to come out properly.
- The SunOS `lp` subsystem will spool PostScript® files. It can also translate the following formats to PostScript for spooling and printing:
  - `troff` to PostScript
  - TeX to PostScript
  - regular text to PostScript
  - Tektronix 4014 to PostScript
  - Diablo 630 to PostScript
  - `plot`(5) to PostScript

All these conversions are 8-bit clean. Be aware that standard paper size around the world varies widely. Internationalized applications do not assume any particular set of page sizes. SunOS provides no support for tracking the output page size; this is the responsibility of the application program itself.

Character and Code Sets

The ISO Latin-1 character set is used to represent European characters sets. ISO Latin-1 uses eight bits (one byte) to represent each character, allowing for 255 characters. It is compatible with the 7-bit ASCII code set in that all ASCII characters have identical encodings in the ISO Latin-1, when the most significant bit in ISO Latin-1 is set to 0. ISO Latin-1 can be thought of as a superset of ASCII for purposes of text representation.

East Asian characters, however, cannot fit into a single byte. Consequently Chinese, Japanese, and Korean all require **multi-byte** code sets for language processing. SunOS uses the EUC encoding scheme to represent multi-byte characters. With EUC, one to four bytes may be used to store a character.

EUC characters are not necessarily convenient to process by standard functions because of their variable length nature, so SunOS provides functions to convert EUC characters to **wide characters**. In SunOS, wide characters are four bytes long, and can be processed using wide character library routines. The next sections examine EUC and wide characters in detail.

Extended UNIX Code (EUC)

SunOS has adopted the EUC from USL's Multi-National Language Supplement (MNLS). EUC is used primarily for storing data in files.

EUC is comprised of four code sets, three of which may be multi-byte, and two of which must be announced by 8-bit control codes known as single-shift characters. EUC's primary code set (code set 0) is used for ASCII. The three supplementary code sets (code sets 1, 2, and 3) can be assigned to different code sets by the locale administrator.

Code set 0 is single byte, with the most significant bit set to zero. The supplementary code sets can be single- or multi-byte, with the most significant bit set to one. Code sets 2 and 3 have a preceding single-shift character, known as SS2 and SS3 respectively, where SS2 = 0x8E (10001110) and SS3 = 0x8F (10001111). There is no SS1.

Differentiating between code sets is done as follows: If the high bit is 0, the code set is ASCII. If the high bit is 1, the byte is checked for SS2 or SS3 to determine code set. The length (in bytes) of characters from that code set is retrieved from the `LC_CTYPE` locale database governing character classification associated with the current locale.

Code set	EUC Representation
slot 0	0xxxxxxx
slot 1	1xxxxxxx *or*
	1xxxxxxx 1xxxxxxx *or*
	1xxxxxxx 1xxxxxxx 1xxxxxxx
slot 2	SS2 1xxxxxxx *or*
	SS2 1xxxxxxx 1xxxxxxx *or*
	SS2 1xxxxxxx 1xxxxxxx 1xxxxxxx
slot 3	SS3 1xxxxxxx *or*
	SS3 1xxxxxxx 1xxxxxxx *or*
	SS3 1xxxxxxx 1xxxxxxx 1xxxxxxx

Whether code sets 1, 2, and 3 are single-byte, double-byte, or triple-byte depends on the locale. Code set 1 could be used to represent ISO Latin-1, but this is not the usual practice in East Asian locales.

EUC divides the code set space into graphic and control characters. Graphic characters are those that can be displayed. Special characters include control characters, unassigned characters, and the space and delete characters. Control characters are characters other than graphic characters, whose occurrence may initiate, modify, or stop a control operation. The following table indicates the single-byte special characters.

Special Character	**EUC Representation**
Space	`00100000`
Delete	`01111111`
Control codes (Primary)	`000xxxxx`
Control codes (Supplementary)	`100xxxxx`

SS2 and SS3 are examples of supplementary control codes.

Wide Characters

EUC is intended primarily for external data storage, and its encoding schemes provide reasonably compact representations for data storage. However, EUC is not very convenient for internal processing—its variable length nature complicates constructing homogenous character arrays, for example. To assist convenient internal processing, SunOS provides a wide character format, plus a collection of library functions for operating on wide characters. Additional functions are available to convert from EUC format to wide characters and from wide characters back to EUC format.

Wide characters are the ANSI C data type `wchar_t`, defined in SunOS as `typedef long`. EUC code sets with one, two, or three bytes get mapped to wide characters as shown below. Four bytes are enough to represent the entire Chinese character set defined by the Chinese National Standard CNS 11643-86, with space left over for user-defined characters.

Table 3-4 EUC and Wide Character Representation

Code set	EUC Representation	Wide Character Representation
0	0xxxxxxx	00000000 00000000 00000000 0xxxxxxx
1	1xxxxxxx 1yyyyyyy	00110000 00000000 00xxxxxx xyyyyyyy
2	SS2 1xxxxxxx 1yyyyyyy 1zzzzzzz	00010000 000xxxxx xyyyyyyy yzzzzzzz
3	SS3 1xxxxxxx 1yyyyyyy 1zzzzzzz	00100000 000xxxxx xyyyyyyy yzzzzzzz

Wide characters provide a standard character size, and are useful for indexing, interprocess communication, memory management, and other tasks that use character counts and known array sizes. Wide characters are stateless and unambiguous within a given locale.

Note that with the SunOS model, the single byte ISO Latin-1 character is represented in wide character form as follows:

```
00110000 00000000 00000000 0xxxxxxx
```

This is the only way in which a single byte character can lose its sense of ‘‘single-bytedness.’’

Multi-byte Library Routines

SunOS provides four library routines to convert characters and strings from EUC representation to wide character representation and back again. SunOS also provides a set of library routines to perform standard operations on wide characters.

Use `mbtowc()` to convert EUC representation to a wide character, and `wctomb()` to convert a wide character to EUC representation. For strings (arrays of characters), use `mbstowcs()` to convert EUC representation strings to wide character strings and `wcstombs()` to convert wide character strings to EUC representation strings. All these are in `libc`.

SunOS provides library routines in `libw` to replace or supplement character and string routines in `libc`. Compile multi-byte programs using the `-lw` option to the linker.

A wide character standard I/O package is available; its routines have the letter `w` in front of `c` (for character-based routines) or `s` (for string-based routines). There is the `wsprintf()` function for wide character formatting, and the `wsscanf()` function for wide character input interpretation. All the `is*()` functions are duplicated by `isw*()` functions for wide characters, and all the `str*()` functions by `ws*()` functions for wide character string operations.

When making applications multi-byte capable, you should increase buffer size four-fold in order to preserve efficiency, since `wchar_t` is four bytes long.

Table 3-5 International Library Routines

Library Routine	**Description**
Locale Management	
`setlocale()`	set or query language or locale
`nl_langinfo()`	obtain various language or locale information
Character Type	
`isalnum()`	is letter or digit
`isalpha()`	is a letter
`isascii()`	is 7-bit ASCII character
`iscntrl()`	is control code
`isdigit()`	is a digit
`isgraph()`	is visible
`islower()`	is lower-case
`isprint()`	is printable
`ispunct()`	is a punctuation mark

Table 3-5 International Library Routines

Library Routine	**Description**
`isspace()`	is white space
`isupper()`	is upper-case
`isxdigit()`	is a hexadecimal digit
`toascii()`	convert to ASCII
`tolower()`	convert to lower-case
`toupper()`	convert to upper-case
String Collation	
`strcoll()`	compare two strings
`strxfrm()`	transform string for comparison
Date and Time	
`strftime()`	convert date and time to string
Formatted Output	
`printf()`	print formatted string
`fprintf()`	format string to file stream
`sprintf()`	format string in memory
Formatted Input	
`scanf()`	scan formatted string
`fscanf()`	scan string from file stream
`sscanf()`	scan string in memory
Monetary Format	
`localeconv()`	returns structure containing monetary format
SunOS Messaging	
`bindtextdomain()`	associate path name with message domain
`textdomain()`	open message catalog domain
`gettext()`	get message from catalog

Table 3-5 International Library Routines

Library Routine	**Description**
`dgettext()`	get message from catalog domain
X/Open Messaging	
`catopen()`	open message catalog (X/Open)
`catgets()`	get message from catalog (X/Open)
`catclose()`	close message catalog (X/Open)
Regular Expressions	
`regexpr(3G)`	regular expression handler (is EUC multibyte-capable, but no wide character interface has been provided)
Multi-byte Handling	
`mblen()`	get length of multi-byte character
`mbtowc()`	multi-byte to wide character
`wctomb()`	wide character to multi-byte character
`mbstowcs()`	multi-byte string to wide character string
`wcstombs()`	wide character string to multi-byte string
Wide Characters	
`wscat()`	concatenate wide char strings
`wsncat()`	concatenate wide char strings to length *n*
`wsdup()`	duplicate wide char string
`wscmp()`	compare wide char strings
`wsncmp()`	compare wide char strings to length *n*
`wscpy()`	copy wide char strings
`wsncpy()`	copy wide char strings to length *n*
`wschr()`	find character in wide char string
`wsrchr()`	find character in wide char string from right
`wslen()`	get length of wide char string
`wscol()`	return display width of wide char string

Table 3-5 International Library Routines

Library Routine	**Description**
`wsspn()`	return span of one wide char string in another
`wscspn()`	return span of one wide char string not in another
`wspbrk()`	return pointer to one wide char string in another
`wstok()`	move token through wide char string
Wide Formatting	
`wsprintf()`	generate wide char string according to format
`wsscanf()`	interpret wide char string according to format
Wide Numbers	
`wstol()`	convert wide char string to long integer
`wstod()`	convert wide char string to double precision
Wide Strings	
`wscasecmp()`	compare wide char strings, ignore case differences
`wsncasecmp()`	compare wide char strings to length *n* (ignore case)
`wscoll()`	collate wide char strings
`wsxfrm()`	transform wide char string for comparison
Wide Standard I/O	
`fgetwc()`	get multi-byte char from stream, convert to wide char
`getwchar()`	get multi-byte char from stdin, convert to wide char
`fgetws()`	get multi-byte string from stream, convert to wide char
`getws()`	get multi-byte string from stdin, convert to wide char
`fputwc()`	convert wide char to multi-byte char, put to stream
`putwchar()`	convert wide char to multi-byte char, put to stdin
`fputws()`	convert wide char to multi-byte string, put to stream
`putws()`	convert wide char to multi-byte string, put to stdin
`ungetwc()`	push a wide char back into input stream
Wide Ctype	

Table 3-5 International Library Routines

Library Routine	**Description**
`iswalpha()`	is wide character letter
`iswupper()`	is wide character upper-case
`iswlower()`	is wide character lower-case
`iswdigit()`	is wide character digit
`iswxdigit()`	is wide character hex digit
`iswalnum()`	is wide character alphanumeric
`iswspace()`	is wide character white space
`iswpunct()`	is wide character punctuation
`iswprint()`	is wide character printable
`iswgraph()`	is wide character graphic
`iswcntrl()`	is wide character control
`iswascii()`	is wide character ASCII
`isphonogram()`	is wide character phonogram
`isideogram()`	is wide character ideogram
`isenglish()`	is wide char in English alphabet from sup code set
`isnumber()`	is wide character digit from supplementary code set
`isspecial()`	is special wide character from sup code set
`towupper()`	convert wide character to upper-case
`towlower()`	convert wide character to lower-case
Codeset Info	
`getwidth()`	get code set information on EUC and screen width
`euclen()`	get EUC byte length
`euccol()`	get EUC character display width
`eucscol()`	get EUC string display width
`csetlen()`	return number of bytes for an EUC code set
`csetcol()`	return columns needed to display EUC code set

Naming Rules

In SunOS, the following objects must be composed of ASCII characters.

- User name, group name, and passwords
- System name
- Names of printers and special devices
- Names of terminals (`/dev/tty`*)
- Process ID numbers
- Message queues, semaphores, and shared memory labels

The following may be composed of ISO Latin-1 or EUC characters:

- File names
- Directory names
- Command names
- Shell variables and environment variable names
- Mount points for file systems
- NIS key names and domain names

The names of NFS shared files should be composed of ASCII characters. Although files and directories may have names and contents composed of characters from supplementary code sets, using only the ASCII code set allows NFS mounting across any machine, regardless of localization.

What Is a Locale?

The key concept for application programs is that of a program's *locale*. The locale is an explicit model and definition of a native-language environment. The notion of a locale is explicitly defined and included in the library definitions of the ANSI C Language standard.

The locale consists of a number of categories for which there are language-dependent formatting or other specifications. A program's locale defines its code sets, date and time formatting conventions, monetary conventions, decimal formatting conventions, and collation order.

A locale name is comprised of language, territory, and possibly code set, although territory is dropped when not needed. Code set is usually assumed. For example, German is `de`, an abbreviation for Deutsch, while Swiss German is `de_CH`, `CH` being an abbreviation for Confoederatio Helvetica. See Appendix A for a list of accepted locale names.

Generally the locale name is specified by the LANG environment variable. Locale categories are subordinate to LANG, but may be set separately, in which case they override LANG. If LC_ALL is set, it overrides not only LANG, but all the separate locale categories as well.

Locale Categories

The locale categories are as follows:

`LC_CTYPE`
A directory whose files control the behavior of character handling functions. The `LC_CTYPE/ctype` file specifies character types for `<ctype.h>`.

`LC_TIME`
A readable file that specifies date and time formats, including month names, days of the week, and common full and abbreviated representations.

`LC_MONETARY`
A binary file that specifies monetary formats. Very few SunOS commands or library routines actually use this database.

`LC_NUMERIC`
A tiny file that specifies the decimal separator (or radix character) and the thousands separator.

`LC_COLLATE`
A directory containing files that specify sorting order for a locale, and string conversions required to attain this ordering.

`LC_MESSAGES`
A directory containing message catalogs (user message translations). This locale directory would be empty until a localization package containing system message translations is installed. Note that many application packages would have their own separate LC_MESSAGES directories.

All of these locale categories, with the exception of LC_MESSAGES, are defined in both the X/Open and ANSI C standards. LC_MESSAGES is Sun-specific.

}

Writing Internationalized Code 4

This chapter describes some specific steps that you should take to internationalize applications. The material is divided into four main topics: text and code sets, formatting and collation, user messages, and nonglobal locales.

Linking

Some internationalization components depend on dynamic linking to function correctly. The default when compiling and linking in the Solaris environment is dynamic linking. Take care not to specify static linking.

Text and Code sets

Call `setlocale()`

The SunOS system support the POSIX/ANSI C function `setlocale()`, which initializes language and cultural conventions. Most applications should set the locale category `LC_CTYPE` except those not concerned with character interpretation, such as block I/O to disk or network. To control the dynamic handling of different code sets in an application, add these lines to your code:

```
#include <locale.h>
main() {
  (void) setlocale(LC_CTYPE, "");
}
```

Among other things, this ensures that European accented characters such as ö are correctly identified with an `isalpha()` library call. Note that the empty string argument indicates that the application should set its codeset according to the environment variable `LC_ALL`, `LC_CTYPE`, or `LANG`—in that order of precedence. If none of these environment variables is set, the default locale is `C`, which results in old-style UNIX behavior.

Internally this call changes values in the `__ctype` array of the C library. This in turn affects the behavior of various `ctype`(3) library routines. The `LC_CTYPE` locale category may also affect other functions, including wide-character handling.

In most cases, library packages should rely on the programmer to call `setlocale()` inside the application. If not, the call is re-entrant in the sense that it affects static data structures. Applications that fail to call `setlocale()` would simply fail to get international features.

To set all the above locale categories at the same time, use the `LC_ALL` argument to `setlocale()` instead of just `LC_CTYPE`. In practice, most applications should set the `LC_ALL` category once and for all.

Make Software 8-bit Clean

Programs shouldn't alter the most significant bit of a `char`. The computer industry used this bit for parity many years ago, but it didn't work out well—data got corrupted because software ignored the parity bit. Now standards committees have decided to define 8-bit code sets, which means you have to clean up your code now. Here are some problems to look for.

Code that explicitly uses the most significant bit for its own purposes is said to be "dirty". There may be valid reasons for altering the most significant bit, but dirty code often involves setting and clearing private flags:

```
#define INVERSE 0x80        /* bad practice */
char c;
c |= INVERSE;
```

Find another way to encode this information. A trick used several times in the operating system was to extend this data type to be `unsigned short` or `unsigned int`, and later setting the top bit of the new data type.

Code that assumes characters are only seven bits long is dirty. Here's an example of masking off the most significant bit on the assumption it's just the parity bit:

```
c = *(string+i) & 0x7F;/* bad practice */
```

A useful exercise is to search your code for constants like "0x80", "0x7f", "0200", "0177", "127", and "128". These constants often highlight problematic code immediately, if such bit patterns are used in conjunction with character handling.

Code that assumes a particular character range needs fixing:

```
if (c >= 'a' && c <= 'z')/* bad practice */
```

Rewrite this to:

```
if (islower(c))
```

Use codeset independent routines found in `<ctype.h>` such as `isalpha()`, `isprint()`, and so on. Software should have been using these functions all along, as they were always needed for portability to IBM's EBCDIC codeset. SunOS also provides wide-character equivalents such as `iswalpha()` and `iswprint()`.

Fix code that assumes characters fall in the range 0–127 by extending the range of such tables:

```
static int hashtable[127];                /* bad practice */
```

For example, the above declaration would be better coded as follows:

```
#include <limits.h>
static int hashtable[UCHAR_MAX];
```

`UCHAR_MAX` is defined in `<limits.h>` on all ANSI C conforming systems.

Watch for Sign Extension Problems

One issue that is sometimes invisible to the programmer is the way the C compilers default to using `signed` for all fundamental data types. This can sometimes cause substantial problems in both application and library code.

Code that casts `char` to other lengths may be dirty. Because the `char` data type is signed in SunOS, when a `char` variable holds an 8-bit character that has the most significant bit set, sign extension takes place during assignment. Needless to say, a negative integer might cause problems later on:

```
int i;
char c = 0xa0;
i = c; /* i is now negative */
```

Do not pass raw characters to functions that require `short`, `int`, or `long` arguments. This is bad practice because of the sign extension problem. For example, the following code is incorrect, as it produces a negative integer index into the C library `__ctype` table. This is because the functions are actually macros that generate stubs of in-line code, which assume the argument is an integer, and propagate the sign bit accordingly.

```
char ch;
isascii(ch);
```

The code above could be written like this:

```
unsigned char ch;
isascii(ch);
```

Watch for the use of unadorned `char`s. Unfortunately they have probably been used extensively throughout most code. It is therefore a non-trivial task to change all `char` data to `unsigned char`, especially as this might garner some `lint` or compiler warnings.

So,

```
char ch;
ch = 0xA0;
```

is better written as:

```
unsigned char ch;
ch = 0xA0;
```

On the other hand,

```
char *cp;
while (isspace(*cp)) {
```

is written as:

```
char *cp;
while (isspace((unsigned char)*cp)) {
```

Although all this may sound like a lot of work, in many cases existing code executes correctly in 8-bit mode *without* any changes to the code. You are primarily on the lookout for lazy coding habits that assume ASCII is the only form of character encoding available. When you fix problems, they are usually easy to test using the Compose key of the Type-4, Type-5, PC-AT101, and PC-AT102 keyboard.

Note that the C compiler does not support 8-bit or multi-byte characters in object names—that is, names of routines, variables, and so forth—although it does allow you to initialize 8-bit or multi-byte data in strings.

Employ Standard Code Sets

In certain locales, OpenWindows 3.3 provides support for the ISO 8859-1 standard codeset, also known as ISO Latin-1. Test your software by typing all the Compose sequences in Table 3-1. If your software can display all characters in ISO Latin-1, it can also display all characters on European keyboards. For Asian locales, use code sets supported in the Asian Feature Sets.

Generating PostScript

Code that generates PostScript must use the *nnn* octal form for characters above hexadecimal 7F, since the PostScript interpreter cannot read characters with the eighth bit set. If you are using TranScript 2.1.1, you can obtain the encoding for ISO Latin-1 by using the name `ISOLatin1Encoding` to re-order the font encoding. Note that `ISOLatin1Encoding` is available in TranScript 2.1.1 but not in previous releases of TranScript®.

PostScript Level Two interpreters have the name `ISOLatin1Encoding` as a defined name. See the *PostScript Language Reference Manual*—2nd edition.

Use `ctype` Library Routines

As mentioned above, text processing software must avoid hard-coded character ranges. Upper- and lower-case letters, punctuation marks, numeric digits, and spaces should be defined using library routines under `<ctype.h>`, rather than with hard-coded character ranges.

Table 4-1 Library Routines for Codeset Independence

Routine	**Character is a...**
isalpha(c)	letter
isupper(c)	capital letter
islower(c)	lower-case letter
isdigit(c)	digit from `0`–`9`
isxdigit(c)	hexadecimal digit from `0`–`f`
isalnum(c)	alphanumeric (letter or digit)
isspace(c)	white space character
ispunct(c)	punctuation mark
isprint(c)	printable character
iscntrl(c)	control character
isascii(c)	7-bit character
isgraph(c)	visible graphics character

Avoid Managing the Keyboard

Type-5, Type-4, PC-AT101, and PC-AT102 keyboards have provisions for typing all the ISO Latin-1 characters. If programs use system and window services to read characters, they need only be 8-bit clean. But if programs read `/dev/kbd`, or perform keystroke mapping, they are managing the keyboard. If that's the case, make sure to support these input methods for all keyboard layouts:

- Entering any character using a single keystroke. Note that German keyboards switch Y with Z, while French keyboards switch Q and W with A and Z.

- Using a dead key accent followed by a given keystroke to produce any valid ISO 8859-1 character.
- Pressing the Compose key followed by two additional keystrokes to produce any valid ISO character.

The most obvious way to test software that manages the keyboard is to type all the Compose sequences in Table 3-1, and to type all keystrokes on several country kit keyboards. If at all possible, use OpenWindows to manage the keyboard for you. This saves you time, and gives your software a uniform user interface.

Formats and Collation

Many different formats are employed throughout the world to represent date, time, currency, numbers, and units. These formats should not be hard-wired into your code. Instead, programs should call `setlocale()`, then the various locale-specific format routines, leaving format design to localization work for each country or language.

For string collation, sort orders may vary for different languages. Programs should use the `strcoll()` or `strxfrm()` library routine to perform string comparisons, which use locale-specific collation order.

Note – Locale specific collation requires that the application be dynamically linked.

Time and Date Formats

The secret to producing time and date formats valid in many locales is the `strftime()` library routine. First set the program clock by calling `time()`, then populate a `tm` structure by calling `localtime()`. Pass this structure to `strftime()`, along with a format for date and time, plus a holding buffer:

```
#include <locale.h>
#include <libintl.h>
#include <stdio.h>
#include <time.h>
main()
{
  time_t clock, time();
  struct tm *tm, *localtime();
  char buf[128];

  setlocale(LC_ALL, "");
  clock = time((time_t *)0);
  tm = localtime(&clock);
  strftime(buf, sizeof(buf), "%C", tm);
  printf("%s\n", buf);
}
```

Recommended formats are `%c` for the local short form of date and time, or `%C` for the local long form. Also, `%x` produces the local date form (numeric), and `%X` yields the local time form. If you try out the above program, your results will look something like this:

```
% setenv LC_TIME de
% a.out
Montag, 16. März 1992, 19:19:19 Uhr PST
% setenv LC_TIME fr
% a.out
lundi, 16 mars 1992, 19:19:20 PST
```

Unfortunately many often-used combinations of date and time are missing from the standard. Neither short nor long form of the local date is available, and there is no abbreviation for time without seconds or time zone.

Monetary Formats

Use `localeconv`(3) function to obtain currency formats. It reads formatting conventions of the current locale to populate an `lconv` structure, then returns a pointer to the filled-in object.

Unfortunately, this gives you a data structure, but not a string. A library routine is needed that converts a floating-point number to a string containing the appropriate monetary format. Standards committees are working on this. Until they agree on a solution, here is a highly simplified method for printing currency in a locale-independent manner:

```
#include <locale.h>
#include <libintl.h> #include <libintl.h>
#include <stdio.h>
char *pcurrency(amount)
double amount;
{
  char string[512];
  struct lconv *lconv, *localeconv();

  lconv = localeconv();
  sprintf(string, "%s%s%.2f%s%s\n",
  lconv->p_cs_precedes ? lconv->currency_symbol : "",
  lconv->p_sep_by_space ? " " : "",
   amount,
  lconv->p_sep_by_space ? " " : "",
  lconv->p_cs_precedes ? "" : lconv->currency_symbol);
  return(string);
}
main()
{
  double amount;

  (void)setlocale(LC_ALL, "");
  scanf("%lf", &amount);
  printf("%s\n", pcurrency(amount));
}
```

Replace `strcmp()` With `strcoll()`

Alphabetic ordering varies from one language to another. For example, in Spanish ñ immediately follows n, and digraphs ch and ll immediately follow c and l, respectively. In German the ligature ß is collated as if it were ss. Swedish has additional unique characters following z. Danish and Norwegian have additional characters æ, ø following z.

The traditional library routine for comparing strings, `strcmp()`, remains unchanged. Because it uses ASCII order, `strcmp()` places ’a’ after ’Z’ even in English. This ordering is often unacceptable.

By contrast, the new library routines `strcoll()` and `strxfrm()` can produce any sort order you want. Use `strcoll()` to compare strings, or `strxfrm()` to transform strings to ones that collate correctly.

Fortunately `strcoll()` takes the same parameters and returns the same values as `strcmp()`. Unfortunately `strcoll()` does a lot more work, and is consequently slower. To speed up applications that compare strings frequently, use `strxfrm()` to store transformed strings into arrays that collate more efficiently.

This program reads standard input, builds a binary tree in the correct order using `strcoll()` to compare strings, then prints out the binary tree. This code may be used for tasks such as listing files in a subwindow.

```
#include <locale.h>
#include <stdio.h>
#include <string.h>
struct tnode {  /* node of binary tree */
char *line;
int count;
struct tnode *left, *right;
};

main()  /* collate: sort a list of lines using strcoll() */
{
  struct tnode *root, *tree();
  char line[BUFSIZ];
  root = NULL;
  (void)setlocale(LC_ALL, "");
  while (fgets(line, BUFSIZ, stdin))
    root = tree(root, line);
  treeprint(root);
}

struct tnode *
tree(p, line)  /* install line at or below tree pointer */
struct tnode *p;
char *line;
{
  char *cp, *malloc(), *strcpy();
  int cond;
  if (p == NULL) {
    p = (struct tnode *)malloc(sizeof(struct tnode));
    if ((cp = malloc(strlen(line)+1)) != NULL)
      strcpy(cp, line);
    p->line = cp;
    p->count = 1;
    p->left = p->right = NULL;
  }
  else if ((cond = strcoll(line, p->line)) == 0)
    p->count++;
  else if (cond < 0)
    p->left = tree(p->left, line);
  else /* cond > 0 */
    p->right = tree(p->right, line);
  return(p);
}
treeprint(p)  /* print tree recursively starting at p */
struct tnode *p;
{
  if (p != NULL) {
    treeprint(p->left);
    while (p->count--)
      printf("%s", p->line);
    treeprint(p->right);
  }
}
```

User Messages and Text Presentation

One of the most critical tasks in software internationalization is providing messages that can be translated easily. Messages are what users see first: help text, button labels, menu choices, usage summaries, error diagnostics, and so forth.

The ease of message localization can vary greatly. In a well-designed application, non-technical people can translate message files into their native languages. In a non-international application, engineers fluent in a language must translate every string inside a program, then recompile the code. There should be no explicit strings in an international application, except those passed to `gettext()`.

Two similar (but incompatible) methods for international messaging in SunOS are: `catgets()` from the X/Open standard, and `gettext()` from the POSIX.1b and UniForum proposals. Both routines provide an interface to message catalogs: text databases that are easy to compose, translate, and access. Because the contents of a message catalog are separate from application code, text can be selected by locale at run-time without altering the code itself.

SunOS supports the existing SVR4 messaging schemes. This includes both the X/Open XPG3 message catalog scheme using `catgets()`, and the SVR4 private scheme using `gettxt()`. However, both schemes are inflexible when it comes to handling messages identified by mnemonic form. This is the rationale for the messaging interface using `gettext()`, which is based on a proposal made by the UniForum Technical Subcommittee on Internationalization. This extension can be run in conjunction with the existing schemes, or can be used as the sole technique for messaging a SunOS application. Applications that call `gettext()` must include the header file `<libintl.h>` and compile with the `-lintl` linker option.

Localized Text Handling

When creating international applications, developers usually write text strings (error messages, text for buttons and menus, and so forth) in their native language, for later translation into other languages. SunOS lets you define any language as the native language, and any other language as the alternate language.

The steps to localize text handling are:

1. Verify that source code uses `textdomain()` and `gettext()`, or else `dgettext()`. These functions accept native language strings as arguments and return equivalent foreign language strings.
2. Extract native language text strings from the `gettext()` functions and store them, with their foreign language equivalents, in a portable message file. This extraction can be done by hand or with the `xgettext` program.

Where Do Messages Reside?

Under SunOS, system messages for libraries and utilities reside in `/usr/lib/locale/`*language*`/LC_MESSAGES/`*domain*`.mo`, where *language* is the specific language—`fr` for French, for example—and *domain* is the specific text domain for that application.

Message files should usually be installed in the same directory hierarchy as the application software. Using SunOS, applications can associate a directory with a message domain by calling `bindtextdomain()`.

Although the message file need not be named after its application, maintenance is easier when the names are similar. The `catopen()` and `textdomain()` library routines actually open the message file. If a message file is missing, users get the untranslated (English or Unixese) message, which is a string contained in the `catgets()` or `gettext()` call.

Using gettext()

It is recommended that programmers use `gettext()` when writing applications for Solaris, even though `gettext()` is not an official standard. SunOS and OpenWindows 3.3 both use `gettext()` exclusively. Here is a short program, `demotext`, that uses both the `gettext()` and `catgets()` functions:

```
#include <locale.h>
#include <libintl.h>
#include <stdio.h>
main()/* demotext.c */
{
  setlocale(LC_ALL, "");
  textdomain("demotext");
  printf("%s\n", gettext("Hello world!"));
  printf("%s\n", gettext("Goodbye."));
}
```

The second line sets the message domain with `textdomain()`, specifying the message domain, which is identical to the message file name. The other lines merely surround literal strings in `printf()` statements with calls to `gettext()`, a routine that searches the message database for key strings, returning the corresponding translated string if it can locate one. Make sure to compile with `cc demotext.c -lintl -o demotext.`

Surround Strings with gettext()

Fortunately `gettext()` is much easier to use than `catgets()`. All you really have to do is go through your programs, enclosing literal strings inside `gettext()` calls. Here's an example of an error message, before:

```
printf("%s: Too few arguments\n", argv[0]);
```

and after:

```
printf(gettext("%s: Too few arguments\n"), argv[0]);
```

Some rearrangement is required here, but not much. Library products should use `dgettext()` in place of `gettext()`, since calling sequence cannot be guaranteed, and different domains may be mixed together at random. The library developer chooses the domain name.

```
printf(dgettext("xview", "Cannot find font.\n"));
```

The call is equivalent to combined `textdomain()` and `gettext()` calls.

In general, you only need to message strings that users see. Do not message strings containing system commands or file names, such as "sort" or "/dev/tty". Be careful when messaging strings inside sprintf(), which is often used to build up path names or command lines. You probably don't need to message strings used only for debugging. Because integers and decimal numbers are not strings, they don't need messaging, either.

Initialized strings require some effort. The one-line initialization statement:

```
char *greeting = "Hello";
```

below must be converted into the two lines:

```
char *greeting;
greeting = gettext("Hello");
```

If strings must be stored in an array, be sure to declare arrays large enough to hold all possible translations. Then call strcpy() as follows:

```
char greeting[BUFSIZ];
strncpy(greeting, gettext("Hello"), sizeof(greeting));
```

Use bindtextdomain()

Many applications do not require root permission for installation, and thus cannot place their messages in /usr/lib/locale. Moreover, most applications need messages in their own directory hierarchy, to simplify export across a network. So most applications should use bindtextdomain() to associate a path name with a message domain. This routine is available in SunOS. Here's a sample invocation:

```
char buf[BUFSIZ];
strcpy(buf, getenv("OPENWINHOME"));
bindtextdomain("xview", strcat(buf, "/locale"));
textdomain("xview");
printf(gettext("Cannot find font."));
```

If the LANG environment had been set to fr, in SunOS this would obtain a translation from $OPENWINHOME/locale/fr/LC_MESSAGES/xview.mo.

Passing a null pointer as the second argument causes bindtextdomain() to return the path name associated with the first argument's message domain. Here's how the path name to message files is constructed under SunOS.

pathname}/$LANG/LC_MESSAGES/{*domain*}.mo

By default the {*pathname*} is `/usr/lib/locale`, although this can be changed with `bindtextdomain()`. `$LANG` comes from the user's environment, and {domain} is supplied by `textdomain()`.

Changing the Text Domain

The following two examples retrieve the same strings but have different effects on the text domain. The first example does not change the current text domain. The second example changes the current text domain to `library_error_strings`, then retrieves the alternate language string of `wrongbutton`.

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

or

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

After writing an application program, create a text domain by extracting `gettext()` strings and placing them in a file with the alternate language equivalent. The following section demonstrates this process.

Create Separate Message Files

Once you have enclosed all user-visible strings inside `gettext()` wrappers, you can run the `xgettext` command on your C source files to create a message file. This produces a readable `.po` file (the portable object) for editing by translators. Running `msgfmt` on this file produces a binary `.mo` file (the message object), which should be installed under `the LC_MESSAGES` directory. Here's a sample interaction on `demotext.c`:

```
% xgettext -m TRNSLT: demotext.c
% cat messages.po
domain "demotext"
msgid "Hello world!"
msgstr "TRNSLT:Hello world!"
msgid "Goodbye."
msgstr "TRNSLT:Goodbye."
% msgfmt demotext.po
% su
Password:
# mv demotext.mo /usr/lib/locale/test/LC_MESSAGES
```

In the portable object, the `msgstr` keyword indicates the search string passed to `gettext()`, while the `msgid` keyword indicates the translated string. Use the `-m` flag with `xgettext` to produce test messages by prepending some obvious string.

If `gettext()` can't find an appropriate string in the message catalog, it returns the string you passed as its first parameter. This means code won't fail if message files are missing, since by default they get indexed source strings. However, you should always deliver message files with an application, because translators may not have access to your source code. If you anticipate difficulty translating `msgid`, insert a comment, or hand-edit `msgstr` to make things clear.

See Chapter 6 for more information on creating message files.

Text Length and Height May Vary

Be aware that translated messages may be of different length and height than the original messages. Translation into certain languages, such as German, often produces longer messages than in English. Some language translations may yield shorter messages. East Asian language ideographs are usually taller than Roman characters.

Window system resource files in OpenWindows 3.3 specify height and width of panel buttons and such. DevGuide 3.3 will employ these facilities. In some cases it's best to use implicit object positioning, letting the window system figure out where to place things. See Chapter 5 for more details.

Try to Avoid Compound Messages

Creating easily translated messages is an art form that involves more than just inserting `gettext()` calls around strings. Remember that word order varies from language to language, so complex messages can be very difficult to translate properly. A common sense guideline is to avoid compound messages with more than two `%s` parts whenever possible.

There are two approaches to messaging: *static* and *dynamic*. Static messaging simply involves looking up strings in a message catalog, with no reordering taking place. Dynamic messaging also involves looking up strings in a message catalog, but those strings are reordered and assembled at run-time. International standards provide an ordering extension to `printf()` for implementing dynamic messaging.

The advantage of static messaging is simplicity. Use it whenever possible. However, avoid splitting strings across two `printf()` statements, which makes messages difficult to translate. You shouldn't need to do this anyway, since you can pass abbreviated strings to `gettext()` to retrieve longer strings in your message catalog. Alternatively, you can use a backslash to hide the line break:

```
printf("This sentence is easy to translate \
because it is included in one printf statement.\n");
```

Translation problems can arise with compound messages, especially when more than one sentence could be produced at run-time. Here is some code that would be very difficult to translate:

```
/* poor practice: multi-part compound message */
printf("%s: Unable to %s %d data %s%s - %s",
    func, (alloc_flg ? "allocate" : "free"),
    count, (file_flg ? "file" : "structure"),
    (count == 1 ? "" : "s"), perror("."));
```

Quite apart from being poor programming practice, this fragment of code would be much clearer to the reader and much easier to translate if it were split into separate print statements inside an `if-else` block that would select the correct message at run-time:

```
if (alloc_flg)
    if (file_flg)
        printf("Unable to allocate %d file\n", count);
    else
        printf("Unable to allocate %d structure\n", count);
else
    if (file_flg)
        printf("Unable to free %d file\n", count);
    else
        printf("Unable to free %d structure\n", count);
```

The issue of making the objects plural is not addressed in this example because, in many languages, pluralization involves more than adding "`s`" to the end of a word.

Dynamic Messaging

Dynamic messaging is used when the exact content or order of a message is not known until run-time. Unless done carefully, dynamic messaging causes translation problems. If the positional dependence of keywords is hard-coded into a program, code needs to be changed before messages can be successfully translated. Obviously, this defeats the purpose of internationalization.

X/Open defines an extension to the `printf()` family that permits changing the order of parameter insertion. SunOS also supports this extension. For example, the conversion format `%1$s` inserts parameter one as a string, while `%2$s` inserts parameter two. The entire format string is parameter zero.

Here's a small example of how these extensions can be used. This `printf` statement has position-dependent keywords because the verb must come before the object.

```
/* poor practice: position-dependent keywords */
printf("Unable to %s the %s\n",
(lock_flg ? "lock" : "find"),
(type_flg ? "page" : "record"));
```

This could produce any of four messages in English:

```
Unable to lock the page.
Unable to find the page.
Unable to lock the record.
Unable to find the record.
```

Here are those four messages translated into German. Note that the auxiliary verb must follow, not precede, the object.

```
Das Programm kann die Seite nicht sperren.
Das Programm kann die Seite nicht finden.
Das Programm kann den Rekord nicht sperren.
Das Programm kann den Rekord nicht finden.
```

German syntax requires different word order, so the program's keywords must be reversed. Here is that `printf` statement properly written for dynamic messaging:

```
printf(gettext("Unable to %s the %s\n"),
(lock_flg ? gettext("lock") : gettext("find")),
(type_flg ? gettext("page") : gettext("record")));
```

The German message catalog would then appear as follows:

```
msgid "Unable to %s the %s\n"
msgstr "Das Programm kann %2$s nicht %1$s\n"
msgid "lock"
msgstr "sperren"
msgid "find"
msgstr "finden"
msgid "page"
msgstr "die Seite"
msgid "record"
msgstr "den Rekord"
```

This example might not work on other vendors' systems because of multiple `gettext()` calls per line.

Please consider carefully the effects of dynamic messaging. You might have to reposition parameters during translation. Often this fact isn't recognized until translation actually begins, by which time it's already too late—the software would have to be laboriously re-released.

Other Languages

SunOS 5.4 provides a `gettext`(1) command to retrieve translated messages from a catalog. This command reads the `TEXTDOMAIN` environment variable for domain name, and the `TEXTDOMAINDIR` environment variable for the path name to the message database.

For PostScript programmers, the NeWS toolkit (TNT 3.3) provides some internationalization facilities, but NeWS itself does not.

Summary of Requirements

This chapter presented the standard X/Open and SunOS library routines for internationalizing software. Use standard X/Open routines whenever possible, but use the SunOS `gettext()` function instead of the X/Open `catgets()` function, because `gettext()` has an easier programming interface than `catgets()`. If you're writing window-based software, consult Chapter 5 for further information on how to internationalize window objects.

Some messages produced by routines in `libc` are localized. The application must be dynamically linkedbe be sure that the localized messages are displayed.

Checklists for Internationalization

Do

- Call `setlocale()` to initialize language and cultural conventions.
- Make software 8-bit clean by verifying that it doesn't alter the most significant bit of 8-bit bytes.
- Watch for sign extension problems, since by default Sun C compilers treat all characters as signed values.
- Use standard code sets such as ISO Latin-1 or EUC.
- Use `ctype(3)` library routines to identify character ranges.
- Remember that many countries use a comma for the decimal separator; both `printf()` and `scanf()` have been changed accordingly.
- Use `strftime()`, not `ctime()`.
- Use the `localeconv()` call if you need to obtain local currency formats.
- Replace calls to `strcmp()` with calls to `strcoll()`.
- Surround user message strings with `gettext()` calls and create separate message files for easy translation.
- Plan for size changes in user messages after translation.
- Use explicit functions to center and align text.
- Allow punctuation characters within words.
- Provide translation notes, where possible (as in code comments).

Don't

- Split user message strings—write out whole messages instead.
- Embed graphics in code.
- Use jargon, abbreviations, and acronyms.
- Make assumptions about word order in strings.
- Assume numbers of bytes or characters.
- Test for alphabetic characters by comparing specific characters— 'A' and 'Z', for instance
- Hard code the decimal separator in parsing or arithmetic calculations.
- Hard code or limit the size of currency fields.
- Assume the size of paper on which application output will be printed.
- Use single-letter commands to represent the first letter of commands (such as "p" for "print").
- Make assumptions about what might be in particular byte locations.

- Reference global external variables used by system libraries, such as:
  ```
  char _cur_locale[][]
  unsigned char _ctype[]
  unsigned char _numeric[]
  char *__time[]
  struct _wctype _wcptr
  int _lflag
  ```
- Use both `catgets`() and `gettext`().

Using X/Open Message Catalogs

Although `catgets()` is the standard X/Open function, its programming interface is not as convenient as `gettext()`. You must pass `catgets()` two extra numeric parameters—one for the catalog descriptor, and another for the message set number. Moreover, `catgets()` requires you to assign a numeric message identifier to each message, which can make code confusing and difficult to maintain. Message numbers can easily get out of sync with their message strings.

Note – Sun recommends that programmers use `gettext`() instead, even though it is not an official standard. SunOS and OpenWindows 3.3 both use `gettext`() exclusively. If source portability is a concern, be advised that `gettext()` source is freely available for integration into your product.

Internationalizing OpenWindows 5

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

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

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

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

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

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

Window Object Layout

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

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

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

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

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

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

Object Size and Positioning

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

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

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

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

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

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

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

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

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

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

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

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

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

The button would then appear at position 20,11.

Input Method

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

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

5

Font Handling

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

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

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

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

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

Translating Messages 6

This chapter explains how to create and install message catalogs for your internationalized application. Other aspects of localization, such as formatting and collation, are provided to you along with any SunOS localization packages you have installed. Your translated user messages together with a localization package allow your internationalized application to function properly for a given locale.

Creating a Message File

Follow these steps to create a message catalog:

1. Use `xgettext`(1) to generate a `.po` (portable message) file. This is a readable text file intended for editing.

2. Translate each `msgid` string in the portable message file, placing the translation on the `msgstr` line directly below it.

3. Run the `msgfmt`(1) program on the portable message file to create a `.mo` (message object) file. This is a data file containing a binary message tree for quick machine access. When a program calls `gettext()` or `dgettext()`, the message routines find the appropriate translated string. Your message object file should be called *domain*`.mo`, where *domain* is the name of the text domain you selected

4. Install the message object file—*domain*`.mo`—into the `LC_MESSAGES` directory. This might be under `/usr/lib/locale/$`*LANG*, but more likely will be the directory specified by your program's `bindtextdomain()` call.

The Portable Message File

First run the xgettext command to extract gettext() strings.

```
% xgettext -d demotext demotext.c
```

The output from xgettext is a portable message (suffix .po) file for each domain specified. For the simple "Hello world" example presented in Chapter 4, you would execute xgettext on the source code demotext.c, resulting in an output file called messages.po. In this example, messages is the default domain name. This file would contain:

```
msgid "Hello World!\n"
msgstr

msgid "Goodbye.\n"
msgstr
```

A translator provides the translations by editing this file, appending the target language text to the msgstr lines. If the target language were French, the portable object file might then look like this:

```
msgid "Hello World!\n"
msgstr “Bonjour, tout le monde!\n”

msgid "Goodbye.\n"
msgstr “Au revoir.\n”
```

Lines beginning with an sharp (#) are comments. Empty lines are ignored. Statement lines contain a directive and value. A directive always starts at the beginning of line, and is separated from its value by white space.

If your application program calls textdomain() or dgettext() to set the message domain, then xgettext produces a *textdomain*.mo file for each text domain in the source code.

The following statements are recognized:

`domain` *domainname*
: States that all following target strings until another `domain` directive are contained in *domainname,* which can be any string up to 255 bytes, containing any character in the portable character set, but excluding slash (/). Until the first `domain` directive, all target strings belong to the default domain, `messages`.

`msgid` *message_identifier*
: Specifies the value of the message identifier associated with the following `msgstr` directive. The *message_identifier* string identifies a target string at retrieval time. Usually this is the untranslated string.

`msgstr` *message_string*
: Specifies the target string associated with the *message_identifier* string declared in the immediately preceding `msgid` directive. Every statement containing a `msgstr` directive must be immediately preceded by a `msgid` statement. Usually *message_string* is the translated message.

Any other form of input-line syntax is illegal. Message strings and message identifiers can contain special characters and escape sequences as defined in the following table:

Description	**Symbol**
<newline>	\n
<tab>	\t
<vertical-tab>	\v
<backspace>	\b
<carriage-return>	\r
<form-feed>	\f
<backslash>	\\
<double-quote>	\"
octal bit pattern	\ddd
hex. bit pattern	\xDD

The escape sequence *ddd* consists of a backslash followed by 1, 2, or 3 octal digits specifying the value of the desired character. The escape sequence *xDD* is similar but introduces a hexadecimal value. If the character following backslash is not specified in the table above, the effect is undefined.

The Message Object File

Running a file in portable message format through the `msgfmt` utility produces a binary version of the file which can be installed in the locale directory (for example `$APPHOME/locale/$`*LANG*`/LC_MESSAGES`). This file can then be accessed by `gettext()` at run time.

After you have saved the edited `.po` file, run the `msgfmt` command.

```
% msgfmt demotext.po
```

The output from `msgfmt` is a message object file whose name is the domain name, suffixed with `.mo`. The `msgfmt` utility generates one `.mo` file for each text domain in the `.po` file(s). In the above example, executing `msgfmt` on the file `messages.po` produces a binary file name `messages.mo`. Install this file into the appropriate `LC_MESSAGES` sub-directory, renaming it to the proper *textdomain* as necessary.

Language and Territory Names A

Language and Territory

An internationalized program makes no assumptions about the language and format of text it is designed to handle. It must work equally well in any locale for data generated internally, text read from or written to files, and messages presented to the user.

To determine locale-specific conventions at run time, programs query system databases, installed in `/usr/lib/locale`, for cultural data. Applications should identify the proper locale using ISO standard names, presented below.

Standard Locale Names

A locale name is always composed of a language name, sometimes with a territory name appended. SunSoft's list of accepted names for languages and territories follows below. Language names (all lower case) came from the ISO 639 standard. Territory names (all upper case) came from the ISO 3166 standard. Language is separated from territory by an underscore, and territory is optional.

Some languages have longer, more mnemonic names. For example, JLE accepts the locale name `japanese`, KLE accepts the locale name `korean`, HLE accepts the locale name `tchinese`, and CLE accepts the locale name `chinese`. Solaris 2.3 continues to support these long names.

Table A-1 Language and Territory Names

Locale	Language / Territory	Locale	Language / Territory
C	Default "C" locale	ar	Arabic
bg	Bulgarian	ca	Catalan
co	Corsican	cs	Czech
cy	Welsh	da	Danish
de	German	de_CH	Swiss German
el	Greek	en	English
en_UK	UK English	en_US	US English
eo	Esperanto	es	Spanish
eu	Basque	fa	Persian
fi	Finnish	fr	French
fr_BE	Belgian French	fr_CA	Canadian French
fr_CH	Swiss French	fy	Frisian
ga	Irish	gd	Scots Gaelic
hu	Hungarian	is	Icelandic
it	Italian	iw	Hebrew
ja	Japanese	ji	Yiddish
kl	Greenlandic	ko	Korean
lv	Latvian	nl	Dutch
no	Norwegian	pl	Polish
pt	Portuguese	ro	Romanian
ru	Russian	sh	Serbo-Croatian
sk	Slovak	sr	Serbian
sv	Swedish	th	Thai
tr	Turkish	zh	Chinese
zh_TW	Chinese in Taiwan		

Compose Key Sequences B

SPARC and x86 keyboards can produce all characters in the standard ISO 8859-1 (Latin-1) codeset by using the Compose key. On a SPARC keyboard, pressing the Compose key should illuminate the light on the key. On an x86 keyboard, the Compose key is control-shift-F1 (hold the Control and Shift keys down and press the F1 key).

Localized keyboards do not all generate the same key codes for ISO Latin-1 characters. However, system translation tables take care of producing the right character code. SunOS finds its tables in `/usr/share/lib/keytables`.

Table B-1 shows how to produce special ISO Latin-1 characters using Compose key sequences.

Table B-1 Compose Key Sequences

Compose		Result	Description
space space			space
!	!	¡	inverted exclamation
c	/	¢	cents
l	-	£	pounds sterling
o	x	¤	currency symbol
y	-	¥	yen
|	|	¦	broken bar
s	o	§	section

Table B-1 Compose Key Sequences

Compose		Result	Description
"	"	¨	umlaut/diaeresis
c	o	©	copyright
-	a	ª	feminine ordinal
<	<	«	left guillemet
-	|	¬	not sign
-	-	-	soft hyphen
r	o	®	registered
^	-	¯	macron
^	0	°	degree
+	-	+-	plus-minus
^	2	$ sup 2$	superscript 2
^	3	$ sup 3$	superscript 3
\	\	´	prime/acute accent
/	u	mu	mu/micro
P	!	¶	pilcro/paragraph
^	.	·	middle dot
,	,	¸	cedilla
^	1	$ sup 1$	superscript 1
_	o	º	masculine ordinal
>	>	»	right guillemet
1	4	1/4	quarter
1	2	1/2	half
3	4	3/4	three quarters
?	?	¿	inverted question
A	‘	À	A grave
A	’	Á	A acute
A	^	Â	A circumflex

Table B-1 Compose Key Sequences

Compose		Result	Description
A	~	Ã	A tilde
A	"	Ä	A umlaut
A	*	Å	A angstrom
A	E	Æ	AE ligature
C	,	Ç	C cedilla
E	‘	È	E grave
E	’	É	E acute
E	^	Ê	E circumflex
E	"	Ë	E umlaut
I	‘	Í	I grave
I	’	Í	I acute
I	^	Î	I circumflex
I	"	Ï	I umlaut
D	-	D-	Eth
N	~	Ñ	N tilde
O	‘	Ò	O grave
O	’	Ó	O acute
O	^	Ô	O circumflex
O	~	Õ	O tilde
O	"	Ö	O umlaut
x	x	x	multiply
O	/	Ø	O slash
U	‘	Ù	U grave
U	’	Ú	U acute
U	^	Û	U circumflex
U	"	Ü	U umlaut
Y	’	Y’	Y acute

Table B-1 Compose Key Sequences

Compose		Result	Description
T	H	Th	Thorn
s	s	ß	Eszett/digraph s
a	‘	à	a grave
a	’	á	a acute
a	^	â	a circumflex
a	~	ã	a tilde
a	"	ä	a umlaut
a	*	å	a angstrom
a	e	æ	ae ligature
c	,	ç	c cedilla
e	‘	è	e grave
e	’	é	e acute
e	^	ê	e circumflex
e	"	ë	e umlaut
i	‘	ì	i grave
i	’	í	i acute
i	^	î	i circumflex
i	"	ï	i umlaut
d	-	d-	eth
n	~	ñ	n tilde
o	‘	ò	o grave
o	’	ó	o acute
o	^	ô	o circumflex
o	~	õ	o tilde
o	"	ö	o umlaut
-	:	-:	divide
o	/	ø	o slash

Table B-1 Compose Key Sequences

Compose		Result	Description
u	‘	ù	u grave
u	’	ú	u acute
u	^	û	u circumflex
u	"	ü	u umlaut
y	’	y’	y acute
t	h	th	thorn
y	"	ÿ	y umlaut

The previous table of Compose key sequences is for ISO 8859-1 only. The ISO 8859 codesets are listed below.

Table B-2 ISO 8859 Standard Codesets

Codeset	Name	Coverage	Approved
8859-1	Latin-1	Western Europe	15 February 1987
8859-2	Latin-2	Eastern Europe	15 February 1987
8859-3	Latin-3	Maltese, Catalan, Galician, Esperanto	15 April 1988
8859-4	Latin-4	Baltic and Nordic region	15 April 1988
8859-5	Cyrillic	Slavic countries	1 December 1988
8859-6	Arabic	Arab countries	15 August 1987
8859-7	Greek	Greece	15 November 1987
8859-8	Hebrew	Israel	1 June 1988
8859-9	Latin-5	8859-1 minus Iceland plus Turkey	15 May 1989