Support for Internationalization 3

This chapter describes features of Solaris that provide the foundation for internationalization support. Chapter 1, "Writing Internationalized Code" gives examples of international coding practices, and Chapter 1, "Internationalizing OpenWindows" 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 identified above do 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 3 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.

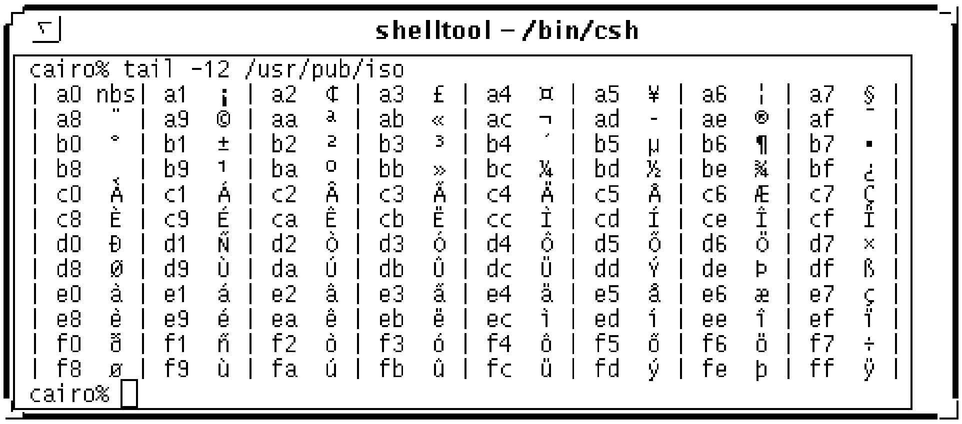
The PC-AT101 and PC-AT102 keyboards are currently available in:

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 Figure 3-1:

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 the SunOS system 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 the SunOS system 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 the tasks described 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 the OpenWindows environment, 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 space bar. 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 the Solaris system for x86 is Control-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 Control-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 valid only for the ISO-8859-1 code set and not for the optional IBM® DOS 437 code set. In the OpenWindows environment, these tables are valid because only the ISO-8859-1 code set is supported.

Compose Key Sequences

Table B-1 on page 67 shows how to produce special ISO Latin-1 characters using Compose key sequences.

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 support only 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 work with 8-bit data.

Dumb Terminals

Dumb terminals set to 8-bit space-parity mode 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 spools PostScript® files. It can also translate the following formats to PostScript for spooling and printing:
  - `troff`
  - TeX
  - regular text
  - Tektronix® 4014
  - Diablo® 630
  - `plot`(5)

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. The SunOS system 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 **multibyte** code sets for language processing. SunOS uses the EUC encoding scheme to represent multibyte 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 multibyte, 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 multibyte, 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

Code set	EUC Representation
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 single-byte special characters are:

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, the SunOS system 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 the SunOS system 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.

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
```

Multibyte Library Routines

SunOS provides four library routines to convert characters and strings from EUC representation to wide character representation and back again. The SunOS system 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 functions are in `libc`.

The SunOS system provides library routines in `libw` to replace or supplement character and string routines in `libc`. Compile multibyte 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 multibyte capable, you should increase buffer size fourfold in order to preserve efficiency, since `wchar_t` is four bytes long.

Table 3-3 International Library Routines *(1 of 5)*

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 3-3 International Library Routines *(2 of 5)*

Library Routine	Description
`isspace()`	Is white space
`isupper()`	Is uppercase
`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()`	Return structure containing monetary format
SunOS Messaging	
`bindtextdomain()`	Associate path name with message domain
`textdomain()`	Open message catalog domain
`gettext()`	Get message from catalog

Table 3-3 International Library Routines *(3 of 5)*

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 multibyte character
`mbtowc()`	Multibyte to wide character
`wctomb()`	Wide character to multibyte character
`mbstowcs()`	Multibyte string to wide character string
`wcstombs()`	Wide character string to multibyte 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 3-3 International Library Routines *(4 of 5)*

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 multibyte char from stream, convert to wide char
`getwchar()`	Get multibyte char from stdin, convert to wide char
`fgetws()`	Get multibyte string from stream, convert to wide char
`getws()`	Get multibyte string from stdin, convert to wide char
`fputwc()`	Convert wide char to multibyte char, put to stream
`putwchar()`	Convert wide char to multibyte char, put to stdin
`fputws()`	Convert wide char to multibyte string, put to stream
`putws()`	Convert wide char to multibyte string, put to stdin
`ungetwc()`	Push a wide char back into input stream
Wide Ctype	

Table 3-3 International Library Routines *(5 of 5)*

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 the SunOS system, 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 files shared via the NFS® system 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 system 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 Confederation Helvetica. See Appendix A, "Language and Territory Names" 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. Few SunOS system 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 is empty until a localization package containing system message translations is installed. Note that many application packages 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.