Tru64 UNIX

Writing Software for the International Market

Part Number: AA-RH9YC-TE

September 2002

Product Version: Tru64 UNIX Version 5.1B or higher

This manual discusses how to create international software and describes the tools provided on the Tru64 UNIX operating system that help you write international programs.

Hewlett-Packard Company
Palo Alto, California

Contents

5 Creating Internationalized X, Xt, and Motif Applications

6 Creating Locales

7 Programming Considerations for International Applications

A Summary Tables of Worldwide Portability Interfaces

B Setting Up and Using User-Defined Character Databases

Figures

Tables

About This Manual

HP Tru64 UNIX internationalization tools and routines allow you to write programs for use in a number of nations. These tools and routines enable you to write programs with the following features:

- An interface that appears to be designed for a nation's users
- Source code that is independent of specific native languages and customs

Audience

This manual is intended for experienced applications developers who are writing programs for multinational or non-English language use. Translators who translate the messages displayed by international programs will also find this manual useful.

New and Changed Features

This manual was written for Tru64 UNIX Version 5.1B. The manual has been restructured. The Tru64 UNIX *Writing Software for the International Market* manual now contains information specific to programming international applications. Material on using the international features of Tru64 UNIX has been moved to a companion manual, *Using International Software*. This manual also includes changes related to the following:

- Chapter 2 has been updated with information on Unicode and dense code locales, `*.UTF-8` locales, and enhanced support for the euro currency sign that includes ISO 8859–15 (Latin-9) and UTF-8 locales and additional bitmap fonts.
- Information and guidelines for writing translatable message files has been added to Chapter 3.

Organization

This manual is organized as follows:

Chapter 1	Introduces the basic concepts and procedures for writing programs that meet the needs of international users.
Chapter 2	Discusses techniques for handling character sets, cultural data, and language in an internationalized application.

Chapter 3	Explains how to extract and translate text for messages and how to generate and access message catalogs.
Chapter 4	Describes the `curses` Library routines for writing, removing, and reading wide-character data.
Chapter 5	Discusses how to use GUI programming libraries (X, OSF/Motif, and DECwindows Extensions to OSF/Motif) when writing internationalized programs.
Chapter 6	Discusses the source files for a locale, how to write library methods, and how to build and test locales.
Chapter 7	Discusses miscellaneous programming topics that apply to the creation of international applications. Topics include input methods, user-defined characters, sorting, creating reference pages, data file codeset conversion, and font renderers.
Appendix A	Lists and summarizes internationalized functions for locale initialization, character classification, case conversion, character collation, date and time interpretation, text strings, number conversion, multibyte characters, and string manipulation.
Appendix B	Describes support for user-defined characters (UDCs) in Chinese, Japanese, and Korean, including information on `cedit` and UDC fonts.
Appendix C	Describes DECterm programming features and restrictions.
Appendix D	Contains complete source files for the sample locale discussed in Chapter 6.
Glossary	Defines terms and acronyms used in this manual.

Related Documentation and Standards

This manual focuses on internationalization from the perspective of the application programmer. A companion manual, *Using International Software*, focuses on the user of international applications. That manual is part of the operating system documentation set. It describes setup requirements for using applications in different language environments and how to use operating system commands in a multilanguage working environment.

The following manuals in the operating system documentation set provide information about using the C compiler and other program development tools on a Tru64 UNIX system. If you are developing internationalized applications, see these manuals for general programming information.

- *Programmer's Guide*
- *Programming Support Tools*

The Tru64 UNIX *Documentation Overview* manual provides information on all of the documentation provided with the operating system.

The Tru64 UNIX documentation is available on the World Wide Web at the following URL:

http://www.tru64unix.compaq.com/docs/

The following manual, published by O'Reilly and Associates, Inc., is also a good reference:

Programmer's Supplement for Release 6 of the X Window System

The following standards or draft standards apply to software components discussed in this manual. This manual refers to some of these standards.

- *ANS X3.159 Programming Language C*
- *ISO/IEC 646: 1983*

  Information processing —- ISO 7-bit coded character set for information interchange.
- *ISO 6937: 1983*

  Information processing —- Coded character sets for text communication.
- *ISO 8859-1: 1987*

  Information processing —- ISO 8-bit single-byte coded graphic character sets – Latin alphabet No. 1.
- *ISO/IEC 9899: 1990*

  Information technology —- programming languages —- C.
- *ISO/IEC 9945–1: 1990*

  Information technology – Portable operating system interface (POSIX) – Part 1: System application programming interface (API) [C Language].
- *ISO/IEC 9945–2: 1993*

  Information technology —- Portable operating system interface (POSIX) – Part 2: Shells and Utilities.
- *ISO/IEC 10646:2001*

  Information Technology — Universal Multiple-Octet Coded Character Set (UCS) 2001. The Basic Multilingual Plane defined by this standard is identical with the main body of Unicode character encoding.
- *Code for Information Interchange, JIS X0201–1976*; Japanese national standard.
- *Code of the Japanese Graphic Character Set for Information Interchange, JIS X0208–1990*; Japanese national standard.

- *Code of the Supplementary Japanese Graphic Character Set, JIS X0212–1990*; Japanese national standard.
- *Chinese Character Input Standard, GB18030–2000*; National Standards Bureau of China, Beijing, 2001.
- *Codes of Chinese Graphic Characters for Information Interchange, Primary Set (GB2312–80)*; National Standards Bureau of China, Beijing, 1980.
- *Standard Codes of Common Chinese Characters for Information Interchange, CNS 11643*; Taiwan, 1986, 1992.
- *Standard Codes of Korean Characters for Information Interchange, KSC 5601*; Korea, 1987.
- *Thai Industrial Standard, TIS 620-2533*; Standard for a primary set of graphic characters used for Thai information interchange.
- The Open Group UNIX CAE specifications, specifically:
  - *Commands and Utilities, XCU Issue 5*
  - *Systems Interfaces and Headers, XSH Issue 5*
  - *System Interface Definitions, Issue 5*
  - *Networking Services, Issue 5*
  - *X/Open Curses, XCURSES, Issue 4 Version 2*
- *The Unicode Standard*, Version 3.0 and Version 3.1
- *XllR6 Specification* (including X Input and Output Methods)

Programming for the World: A Guide to Internationalization (O'Donnell, Sandra Martin, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1994) provides information about cultural and linguistic requirements around the world and the changes needed in computer systems to handle those requirements.

Articles in *Digital Technical Journal*, Volume 5 Number 3 (published Summer 1993) cover topics related to product internationalization.

Reader's Comments

HP welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

A Reader's Comment form is located on your system in the following location:

`/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate HP technical support office. Information provided with the software media explains how to send problem reports to HP.

Conventions

The following conventions are used in this manual:

`%`	A percent sign represents the C shell system prompt.
`$`	A dollar sign represents the system prompt for the Bourne and Korn shells.
`#`	A number sign represents the superuser prompt.
`% cat`	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|]	In syntax definitions, brackets indicate items that are optional. Vertical bars separating items inside brackets indicate that you choose one item from among those listed.
{ | }	In syntax definitions, braces indicate items that are required. Vertical bars separating items inside

	braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
⋮	A vertical ellipsis indicates that a portion of an example that would normally be present is not included.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.
Ctrl/*x*	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash, for example, Ctrl/c.
Alt *x*	Multiple key or mouse button names separated by spaces indicate that you press and release each in sequence, for example, Alt Space.

1

Overview of International Software Development

Internationalization refers to the process of developing software programs without prior knowledge of the language, **cultural data**, or character-encoding schemes that the programs are expected to handle. In system terms, internationalization refers to the provision of interfaces that let programs produce varying output, depending on the specific environment in which they are run. The mnemonic **I18N** is frequently used as an abbreviation for internationalization.

This manual describes operating system interfaces and utilities that help you develop internationalized programs. These interfaces and utilities conform to specifications in the X/Open UNIX standard, which allows for implementation-defined behavior in certain areas. This manual identifies those software characteristics that are specific to the operating system.

The following sections provide an overview of the operating system interfaces and utilities used for international program development:

- Language. Section 1.1 is a general description of language requirement implementation. Section 1.2 defines language in software application terms.
- Cultural Data. Section 1.3 defines cultural data and Section 1.1.1 defines the implementation of cultural or local requirements in a computer system.
- Character Sets. Section 1.4 defines character sets in terms of internationalization.

1.1 Language Announcement

Language announcement is the mechanism by which language, cultural data, and codeset requirements are set either for the system as a whole, by an application, or by individual users. Language announcement is performed by setting a locale name in a set of reserved environment variables. System administrators can set the default values for these variables for different shell environments; see the Tru64 UNIX *System Administration* manual for information about setting locale defaults for shells. Users can also set locale variables on a per-process basis.

Typically, internationalized programs read locale variables at run time and use them to attach settings to locale categories in the programs' operational environment. However, programs can also set these categories internally when appropriate. Therefore, the binding to a particular locale need not be general for all parts of a program. Within one execution cycle, different parts of the program can request different localizations.

1.1.1 Localization

Localization refers to the process of implementing local requirements within a computer system. Some of these requirements are addressed by **locales**. Each locale is a set of data that supports a particular combination of **native language**, cultural data, and codeset. The type of information a locale can contain and the interfaces that use a locale are subject to standardization. However, where locales reside on the system and how they are named can vary from one vendor to another.

There is more to localization than providing locales. For example, the localization process means making sure that translations are available for software messages; appropriate fonts and measurement systems are supported and available for display and printing devices; and, in some cases, additional software is written to handle local requirements.

The mnemonic **L10N** is frequently used as an abbreviation for localization.

See Chapter 3 for information on creating and using localized data and message files in application programs. See Chapter 5 for information on localization and graphical user interfaces.

See Chapter 2 for information on the programming aspects of local implementation and Chapter 6 for a description of locales, the primary tool for localizing software.

1.2 Language

An internationalized program makes no assumptions about the language of character **data** (text) that the program is designed to handle.

Language has implications for processing text for such things as character handling and word ordering. The operating system provides interfaces that allow internationalized programs to manipulate text according to the language requirements of individual users.

Language differences require the separation of message text from program code. The operating system provides facilities that allow message text to be separated from the code, translated into different languages, and accessed by the program at run time. Chapter 3 explains how an internationalized

program that uses the **worldwide portability interfaces** (WPI) generates and accesses messages.

An internationalized program that uses X and Motif interfaces can separate message text from program code in the following ways:

- By defining menu items, titles, text fields, and messages in user interface language (UIL) files
- By specifying titles and font lists in application resource files
- By specifying help messages in files that the Help widget uses

For information about separating message text from program code for X and Motif interfaces, see the following manuals:

- *X Window System Toolkit*
- *Common Desktop Environment: Internationalization Programmer's Guide*

1.2.1 Character Classification

Character classification information describes the characteristics associated with each valid character code; that is, whether the code defines an alphabetic, uppercase, lowercase, punctuation, control, space, or other kind of character. Character classification functions and internationalized regular expressions use this information to determine character classes.

1.2.2 Case Conversion

Case conversion refers to information that identifies the possible alternative case of each valid character code. Case conversion functions use this information to change characters from uppercase to lowercase or from lowercase to uppercase. In some languages, case is not a characteristic of all of the letters, or even of any characters.

1.2.3 Message Catalogs

A message catalog is a file or storage area that contains program messages, command prompts, and responses to prompts for a particular language. Motif applications also use resource files and UIL files in addition to, or in place of, message catalogs for text and other values that can vary from one locale to another. Chapter 3 describes the messaging system.

1.3 Cultural Data

Cultural data refers to the conventions of a geopolitical area, called **territory** in this manual. Cultural data includes such things as date, time, and currency formats.

An internationalized program cannot assume how cultural data formats are set in advance and uses system facilities to determine formats at run time. This capability is provided through a language information database (or **langinfo database**) that programs can query for the required formats of cultural data items.

1.3.1 Language Information

Language information refers to localization data that describes the format and setting of cultural data that can vary from one locale to another. The information stored in a langinfo database includes the appropriate formats and characters for date and time, currency, and numeric values.

1.4 Character Sets

A **character set** is a set of alphabetic or other characters used to construct the words and other elementary units of a native language or computer language. A coded character set (or **codeset**) is a set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation.

For a program to be able to handle text recorded in different codesets, the program cannot make assumptions about the size or bit assignment of character encodings. In particular, the program cannot assume that any part of an area used to store a character is available for other uses.

1.4.1 Collating Sequence

The **collating sequence**, or ordering of characters, may be implicit in underlying hardware but can be defined for software to conform to the way language is used in a particular territory. Many languages have complex rules for sorting. The following list describes some of these collating rules:

- A single letter is not necessarily represented by a single character

  In traditional Spanish, for example, the character combination `ch` sorts between the characters `c` and `d`.

- A single character can be equivalent to a combined set of characters

  For example, the ß character is equivalent to `ss` in standard and Swiss German and to `sz` in Austrian German.

- Accented letters do not always follow unaccented letters

  In many languages, this is true only if the words that contain those letters are otherwise identical. In other languages, a particular accented letter may be considered unique and sort after a letter that is different from the unaccented counterpart.

- Characters can be sorted in multiple ways for the same language

  The ideographic characters in Asian languages have sort orders based on pronunciation and on two visually recognized components (radicals, which are pictograms for elements of meaning, and the number of strokes).

Each locale contains information about collating sequences that informs string comparison functions about the relative ordering of characters defined in the associated codeset. Internationalized regular expressions also use the collating sequence for implementing character ranges, collating symbols, and equivalence classes.

1.4.2 Characters and Strings

A **character** is a sequence of one or more bytes that represent a single graphic symbol or control code. Do not confuse the term character with the C programming language `char` data type, which represents an object large enough to store any member of the basic execution character set and which is usually mapped as an 8-bit value. Unlike the `char` data type in C, a character can be represented by a value that is one or more bytes. The expression **multibyte character** is synonymous with the term character; that is, both refer to character values of any length, including single-byte values.

A **character string** or string is a contiguous sequence of bytes terminated by and including the null byte. A string is an array of type `char` in the C programming language. The null byte is a value with all bits set to zero (`0`).

A **wide character** is an integral type that is large enough to hold any member of the extended execution character set. In program terms, a wide character is an object of type `wchar_t`, which is defined in the header files `/usr/include/stddef.h` (for conformance to the X/Open XSH specification) and `/usr/include/stdlib.h` (for conformance to the ANSI C standard). The locations of these header files are determined by standards organizations; however, the definitions themselves are implementation specific. For example, implementations that support only single-byte codesets (not the case for Tru64 UNIX) might define `wchar_t` as a byte value.

A **wide-character string** is a contiguous sequence of wide characters terminated by and including the null wide character. A wide-character string is an array of type `wchar_t`. The null wide character is a `wchar_t` value with all bits set to zero (`0`).

An empty string is a character string whose first element is the null byte. Similarly, an empty wide-character string is a wide-character string whose first element is the null wide character.

1.4.3 Portable Character Set

The Portable Character Set (PCS) is supported in both compile-time (source) and run-time (executable) environments for all locales. The PCS contains the following characters:

- The 26 uppercase letters of the English language alphabet:

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- The 26 lowercase letters of the English language alphabet:

  a b c d e f g h i j k l m n o p q r s t u v w x y z

- The 10 decimal digits:

  0 1 2 3 4 5 6 7 8 9

- The following 32 **graphic characters**:

  ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ‘ { | } ~

- The space character, plus **control characters** that represent the horizontal tab, vertical tab, and form feed
- In addition to the preceding characters, the execution version of the PCS contains control characters that represent alert, backspace, carriage return, and newline

The PCS as defined by X/Open is similar to the basic source and basic execution character sets defined in *ISO/IEC 9899: 1990*, except that the X/Open version also includes the dollar sign ($), commercial at sign (@), and grave accent (`) characters.

Some locales (for example, **ISO 646** variants) may make substitutions for one or more of the preceding characters. In such cases, the substituted character has the same syntactic meaning as the character it replaces in the PCS. An example of a character substitution might be the British pound sign (£) for the number sign (#) that is the default.

The definition of a character set that is portable across all codesets is particularly relevant to encoding formats that support a limited set of native languages. This is typical for most of the character encoding formats developed for UNIX systems. In other words, the codeset used for a Chinese locale must include all the PCS characters in addition to characters that are part of the Chinese language. However, that same codeset probably would not include characters needed to support Russian or Icelandic. Similarly, the codeset used for the Russian language probably would not include any Chinese characters but must include all the PCS characters. Therefore, no matter what the locale setting, programs can assume that characters in the PCS are available.

1.4.4 Universal Character Set

The **Universal Character Set** (UCS), as specified by the Unicode and ISO/IEC 10646 standards, is supported on the operating system. The UCS specifies a repertoire of characters that can be used by all major languages and standardizes the rules used by languages to process characters. This character set supports the philosophy that applications should be able to manipulate characters in any language by using the same encoding format and set of rules. Thus, operating system support of UCS can include **file code** and internal **process code** conversion without having to include multiple algorithms.

The ISO/IEC 10646 standard support two character sizes (16 bits and 32 bits) that require different parsing schemes for data input and output. UCS encoding that an implementation parses in 16-bit units (2 octets) is called UCS-2. UCS encoding that an implementation parses in 32-bit units (4 octets) is called UCS-4. UCS-4 expands the number of characters that can be supported and is more efficiently manipulated as internal process code on larger computer systems.

The standards define a number of universal transformation formats (UTFs) used by the byte-oriented protocols that handle file data. We recommend UTF-8 and UTF-32 for use on the operating system. The operating system supports the following universal transformation formats:

- UTF-8

  UTF-8 is the standard method for transforming UCS-4 process encoding into a sequence of 8-bit bytes and ensuring interchange transparency for characters in C0 code positions (0 to 31), the SPACE character (32), and the DEL character (127). The operating system provides codeset converters and locales for UTF-8.

- UTF-16

  UTF-16 uses the surrogate character extension technique defined in the Unicode standard and represents characters in 16-bit units. UTF-16 is a superset of UCS-2, but does not support full representation of the UCS-4 code space. UTF-16 does support all characters currently defined for languages covered by both standards.

  Byte orientation in file code can differ and, depending on the platform on which the file was generated, can be little-endian (LE) or big-endian (BE). UTF-16 uses a byte order mark (BOM), which is not part of the file text data, to indicate byte orientation. The Unicode standard also defines UTF-16LE and UTF-16BE, which do not include a BOM, for little-endian and big-endian orientations, respectively. The operating system supports UTF-16, UTF-16LE, and UTF-16BE through codeset converters. Because UCS-2 is a subset of UTF-16, the operating system supports UCS-2 with UTF-16 codeset converters. The `UCS-2` codeset

converter name is recognized as an alias for `UTF-16*`, but with a restricted character repertoire.

The operating system normally expects UTF-16, rather than UTF-16LE or UTF-16BE. On input, the system looks for a BOM. If one is not found, the converter assumes UTF-16BE. On output, the system automatically inserts a BOM. If your application expects little-endian or big-endian byte orientation on input or output, you may have to explicitly state byte orientation. See `iconv_intro`(5) for more information on setting byte orientation.

- UTF-32

  UTF-32 allows character representation in 4-byte encoding units. UTF-32 is a restricted subset of UCS-4 in that the range of character values is restricted to U+0000 to U+10FFFF, the same range as UTF-16. Keep in mind that private-use ranges above U+10FFFF will be removed from future versions of ISO/IEC 10646 to promote interoperability between ISO/IEC 10646 and Unicode standard encoding formats.

  UTF-32 uses a BOM to indicate little-endian or big-endian byte orientation. As with UTF-16, the Unicode standard defines UTF-32LE and UTF-32BE, which do not include BOMs. The operating system default for input and output is also the same as UTF-16.

  Use the `UCS-4` codeset converter to process UTF-32.

The operating system supports UCS-4 with codeset converters and locales. The locales and some library functions allow applications to use UCS-4 as internal process code. The codeset converters allow file data to be converted to encoding formats supported by fonts and other software resident on the system.

See Section 2.2 for more information about Unicode, locales, and related encoding formats. See Chapter 4 for a description of the `curses` Library and information on support of wide-character format and multibyte characters.

2

Developing Internationalized Software

This chapter explains how the requirements of localization (language, codeset, and cultural differences) change the way you implement basic coding operations. A sample application that applies the suggested program development techniques from this chapter is provided in the `/usr/examples/i18n/xpg4demo` directory. See the `README` file in that directory for an introduction to the application and how you can compile and run the application with different locales. Parts of the `xpg4demo` application are used as examples in this and other chapters.

One of the primary functions of most computer programs is to manipulate data, which can involve interaction between the program and a computer user. In commercial situations, it is important that such interactions take place in the native language of each user. Cultural data should also observe the correct customs.

When you write programs to support multilanguage operation, you must consider the fact that languages can be represented within the computer system by one or more codesets. Because of the requirements of different languages, characters in codesets may vary in both size (8 bits, 16 bits, and so on) and binary representation.

You can satisfy the requirements of codesets and data by writing programs that make no hard-coded assumptions about language, cultural data, or character encodings. Such programs are said to be internationalized. Data specific to each supported language, territory, and codeset combination are held separately from the program code and can be bound to the run-time environment by language-initialization functions.

The operating system provides the following facilities for developing internationalized software, defining localization data, and announcing specific language requirements:

- Locales that contain language, codeset, and cultural definitions for each language (Section 2.1)
- Library functions that handle extended character codes and that provide language- and codeset-independent character classification, case conversion, number format conversion, and string collation (Section 2.2)
- Library functions that let programs dynamically determine cultural and language-specific data (Section 2.3)

- A message system that allows program messages to be held apart from the program code, translated into different languages, and retrieved by a program at run time (Section 2.4)
- An initialization function that binds a program at run time to the linguistic and cultural requirements of each user (Section 2.5)

The discussion and examples in this chapter focus on functions provided in the Standard C Library. See Chapter 4 for information on using functions in the `curses` Library. See Chapter 5 for information about using functions in the X and Motif libraries.

2.1 Using Locales

The operating system supports **Unicode** and **dense code** locales. The Unicode locales are installed in `/usr/i18n/lib/nls/ucsloc/`. Dense code locales are installed in `/usr/i18n/lib/nls/loc`. The active default is determined by the symbolic link, `/usr/i18n/lib/nls/dloc`. For example, the Japanese locale filename, `/usr/lib/nls/loc/ja_JP.eucJP`, is a symbolic link to `/usr/i18n/lib/nls/dloc/ja_JP.eucJP`, where `/dloc` is a symbolic link to either `/ucsloc` for the Unicode version or `/loc` for the dense code version of the Japanese locale.

If you are superuser, you can switch between Unicode and dense code locales by changing the setting of the symbolic link, as described in `l10n_intro`(5), or you can use the Configure International Software utility from the SysMan Menu. You can also use the utility to change a default system locale and specify an input method for those Asian locales that support multiple input methods. See the online help for Configure International Software for more information.

Unicode locales conform to Unicode and ISO/IEC 10646 standards and use UTF-32 as the wide-character encoding. Under UTF-32 wide character encoding, `wchar_t` values represent the same characters regardless of the locale and, because Unicode standards prevail, implementation is consistent across platforms.

Locales whose names end in `.UTF-8` use file code and UTF-32 internal process code (`wchar_t` encoding) defined in the **ISO 10646** and Unicode standards.

Other, non-UTF-8 Unicode locales use traditional UNIX and proprietary codesets for the file code while using UTF-32 as the internal process code. A subset of these Unicode locales have a `@ucs4` modifier; however, they are the same as the locales without the `@ucs4` modifier. The `@ucs4` subset is provided for backward compatibility and may be removed in the future. You cannot choose `@ucs4` locales from the CDE Login Menu; you must specify the locale name in the `LANG` environment variable.

The universal.UTF-8 locale is also available (for use by applications rather than end users). This locale supports the complete set of characters in the Universal Character Set (UCS).

See Unicode(5) for more information about encoding formats.

For .UTF-8 locales, file code may include characters encoded in more than 1 byte; therefore, use these locales in applications that can process multibyte data. Design new applications based on multibyte .UTF-8 locales, which incorporate a large character repertoire, to enable the application to expand future character support without changing the character set.

Dense code locales use dense code for wide-character encoding to minimize table size (that is, codepoints are assigned consecutively with no empty positions). Under dense code locales, a wchar_t value for one locale may not represent the same character in another locale and, thus, is locale specific. Dense code locales are appropriate for applications that have no dependencies on the internal process code or, because dense code locales are slightly more efficient than Unicode locales, for applications whose primary goal is better performance.

All valid codepoints in multibyte character sets are mapped to valid codepoints in Unicode, including unmapped codepoints that are mapped to Unicode codepoints in the private use area. Thus, dense code locales are equivalent to Unicode locales. In general, the same charmaps and locale source can be used for Unicode and dense code locales. However, Unicode and dense code characters that are not defined in the LC_COLLATE section may be sorted differently.

A Unicode locale exists for each dense code locale. (However, not all Unicode locales have a dense code version.) For Latin-1 locales (ISO8859–1), the dense code and Unicode locales are identical because Latin-1 characters are the same as the first 256 characters in Unicode. Keep in mind that the same locale name can refer to a Unicode locale or to a dense code locale, depending on the setting of the symbolic link. Thus, if running an application in a locale is problematic, check the symbolic link.

Because Unicode locales use consistent values for characters in wchar_t form, the link to Unicode locales can increase consistency across locales and platforms. However, some users may prefer the older, dense code locales that use proprietary algorithms to convert characters to wchar_t form, or an application may have dependencies on dense code wchar_t encoding.

2.2 Using Codesets

In the past, most UNIX systems were based on the 7-bit **ASCII** codeset. However, most non-English languages include characters in addition to those contained in the ASCII codeset.

The X/Open UNIX standard does not require an operating system to supply any particular codesets in addition to ASCII. The standard does specify requirements for the interfaces that manipulate characters so that programs are able to handle characters from whatever codeset is available on a given system.

The first group of the International Standards Organization (ISO) codesets covered only the major European languages. In this group, several codesets allow for the mixing of major languages within a single codeset. All of these codesets are a superset of the ASCII codeset and allow systems to support non-English languages without invalidating existing software that is not internationalized. The Tru64 UNIX operating system always includes a locale for the United States that uses the **ISO 8859–1** (ISO Latin–1) codeset.

Subsets that are installed as part of Worldwide Language Support (WLS) support localized variants of the operating system and may include locales based on additional ISO codesets. For example, the optional language variant subsets included with the operating system to support Czech, Hungarian, Polish, Russian, Slovak, and Slovene provide locales based on the ISO 8859-2 (Latin-2) codeset.

The following is a complete list of ISO codesets provided with the WLS, including the languages that they support and the reference pages where they are discussed in more detail:

- ISO 8859–1, Latin-1

  Languages of Western Europe and North America, including Catalan, Danish, Dutch, English/Great Britain, English/United States, Finnish, Flemish/Belgium, French/Belgium, French/Canada, French/Swiss, French, German/Swiss, German/Germany, Icelandic, Italian, Norwegian, Portuguese, Spanish, and Swedish

  See `iso8859-1`(5)

- ISO 8859–2, Latin-2

  Languages of Eastern Europe, including Czech Republic, Hungarian, Polish, Slovak, and Slovene

  See `iso8859-2`(5)

- ISO 8859–4, Latin-4

  Lithuanian

  See `iso8859-4`(5)

- ISO 8859–5, Latin/Cyrillic

  Russian

  See `iso8859-5`(5)

- ISO 8859–7, Latin/Greek

Greek

See `iso8859-7`(5)

- ISO 8859–8, Latin/Hebrew

  Hebrew/Israel (uses the ISO Hebrew codeset)

  See `iso8859-8`(5)

- ISO 8859–9, Latin-5

  Turkish

  See `iso8859-9`(5)

- ISO 8859–15, Latin-9

  Catalan/Spain, Danish, Dutch, English/Great Britain, English/United States, Finnish, Flemish/Belgium, French/Belgium, French/Canada, French/Swiss, French, German/Swiss, German/Germany, Icelandic, Italian, Norwegian, Portuguese, Spanish/Spain, and Swedish. ISO 8859–15 (and UTF-8) support the **euro** monetary character.

  See `iso8859-15`(5)

The operating system does not include support for the ISO 8859–3 (Latin-3) and ISO 8859–6 (Latin-6) codesets.

Another ISO codeset supported by utilities on a standard operating system is **ISO 6937: 1983**. This codeset, which accommodates both 7-bit and 8-bit characters, is used for text communication over communication networks and interchange media, such as magnetic tape and disks.

The codesets discussed up to this point address the requirements of languages whose characters can be stored in a single byte. Such codesets do not meet the needs of Asian languages, whose characters can occupy multiple bytes. The operating system supplies the following codesets through installed subsets that support Asian languages and countries:

- Japanese
  - Japanese Extended UNIX Code (the default)

    See `eucJP`(5)
  - Shift JIS

    See `shiftjis`(5)
  - DEC Kanji

    See `deckanji`(5)
  - Super DEC Kanji

    See `sdeckanji`(5)
- Korean

  - DEC Korean

    See deckorean(5)
  - Korean Extended UNIX Code

    See eucKR(5)
- Thai
  - Thai API Consortium/Thai Industrial Standard

    See TACTIS(5)
- Simplified Chinese
  - DEC Hanzi

    See dechanzi(5)
  - GBK and GB18030

    See GBK(5) and GB18030(5)
- Traditional Chinese
  - DEC Hanyu

    See dechanyu(5)
  - Taiwanese Extended UNIX Code

    See eucTW(5)
  - BIG-5 (and the variant, Shift BIG-5)

    See big5(5) and sbig5(5)
  - Telecode

    See telecode(5)

These codesets are supplied when you install Asian language variant subsets of the operating system software. Also supplied are a specialized terminal driver and associated utilities that must be available on your system to support the input and display of Asian characters at run time.

Codesets developed for PC systems are commonly called code pages. There are PC code pages that correspond to most of the language-specific codesets developed for UNIX systems. The operating system supports PC codesets mostly through converters that can change file data from one type of encoding format to another. The CP850 codeset supports English/United States and is used with data that contains accented characters generated on a PC using the CP850 code page for character encoding. This character encoding is usually the default for MS-DOS and Windows operating systems in Europe. See code_page(5).

The Unicode and ISO/IEC 10646 standards specify the Universal Character Set (UCS), which allows character units to be processed for all languages,

including Asian languages, using the same set of rules. The operating system supports the UCS-4 (32–bit) encoding of this character set in process code.

Other encoding formats defined by the Unicode standard, the ISO/IEC 10646 standard, or both include the following:

- UCS-2, a 16-bit encoding counterpart to UCS-4
- A number of universal transformation formats (UTF-8, UTF-16, and UTF-32) that transform UCS encoding into sequences of bytes for handling by byte-oriented protocols

The operating system supports these different formats through locales, codeset converters, or both. Because UCS-2 is a subset of UTF-16, the operating system supports UCS-2 with UTF-16 codeset converters. The operating system supports UCS-4 with both codeset conversion and locales.

The following locales use UTF-32 as internal processing code:

- `universal.UTF-8`

  Use this locale in applications to convert data in UTF-8 file format to UCS-4 process code and to test any UCS-4 character to determine if it is included in one of the following `LC-CTYPE` classes: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, or `xdigit`. In this locale, the `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC`, and `LC_TIME` definitions match those of the POSIX (C) locale. Your application can use this locale, along with the `fold_string_w( )` function, to process the full range of characters defined by the Unicode and ISO/IEC 10646 standards.

  This locale differs from most others because it does not provide access to local cultural conventions.

- *`language_territory`*`.UTF-8`

  These locales limit classification information to the characters in a particular native language, make country-specific data available to your application, and assume file data follows UTF-8 encoding rules. The operating system locales that support the euro monetary symbol use either UTF-8 or ISO 8859–15 codesets.

  The Unicode UTF-8 codeset supports Catalan/Spain, Czech Republic, Danish, Dutch, English/Great Britain, English/United States, Finnish, Flemish, French/Belgium, French/Canada, French/Swiss, German/Swiss, German, Greek, Hungarian, Icelandic, Italian, Japanese, Korean, Lithuanian, Norwegian, Polish, Portuguese, Russian, Slovak, Slovene, Spanish, Swedish, Turkish, simplified Chinese (Hanzi), and traditional Chinese (Hanyu). See `Unicode`(5).

- *`native_locale_name`*

These locales use UTF-32 as internal processing code. The codeset portion of the *native_locale_name* (for example, ISO8859–1) specifies the file code. Also, the locale provides classification information for the native language characters, but not for the full set of UTF-32 characters. Country specific information is available to the application; the LC_COLLATE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, and LC_TIME category definitions match the definition in *native_language_name*.

- *native_language_name*@ucs4

  These locales are provided for compatibility with existing applications that use the @ucs4 locales. They function the same as the *native_locale_name* locales, but the list of locales provided is not as complete as the *native_language_name* locales.

See Section 2.5 for information on locale categories, such as LC_TIME. See Unicode(5) and Section 2.1 for information on locales and comparisons of data handling. See euro(5) for more information on the euro monetary symbol.

See Unicode(5) for detailed information about support for UCS-2, UCS-4, UTF-8, UTF-16, and UTF-32. For information on how codesets are supported for a particular **local language**, see the reference page for that language. Reference pages for languages, particularly Asian languages, might note additional codesets that are not supported in a locale but for which there is a codeset converter.

The following sections discuss important issues that affect the way you write source code when your program must process characters in different codesets:

- Ensuring data transparency (Section 2.2.1)
- Using in-code literals (Section 2.2.2)
- Manipulating characters that span multiple bytes (Section 2.2.3)
- Converting between multibyte-character and wide-character data (Section 2.2.4)
- Rules for multibyte characters in source and executable codesets (Section 2.2.5)
- Classifying characters (Section 2.2.6)
- Converting characters (Section 2.2.7)
- Comparing strings (Section 2.2.8)

2.2.1 Ensuring Data Transparency

As discussed in Section 2.2, internationalized software must accommodate a wide variety of character-encoding schemes. Programs cannot assume that

a particular codeset is on all systems that conform to requirements in the X/Open UNIX CAE specifications, nor that individual characters occupy a fixed number of bits.

Because of the historical dependence of UNIX systems on 7-bit ASCII character encoding, some programs use the most significant bit (MSB) of a byte for their own internal purposes. This was a dubious programming practice, although quite safe when characters in the underlying codeset always mapped to the remaining 7 bits of the byte. In the world of international codesets, the practice of using the most significant bit of a byte for program purposes must be avoided.

2.2.2 Using In-Code Literals

When you write internationalized software, avoid using in-code literals. Consider, for example, the following conditional statement:

```
if ((c = getchar()) == \141)
```

This condition assumes that lowercase a is always represented by a fixed octal value, which may not be true for all codesets. Use a function, instead of an in-code literal. Consider the following statement that uses a `getchar( )` function to substitute a character constant for the octal value:

```
if ((c = getchar()) == 'a')
```

However, because the `getchar( )` function operates on bytes, the statement would not work correctly if the next character in the input stream spanned multiple bytes. To avoid this problem, substitute the `getwchar( )` function for the `getchar( )` function. The `getwchar( )` function, as used in the example, works correctly with any codeset because a is a member of the PCS and is transformed into the same wide-character value in all locales.

```
if ((c = getwchar()) == L'a')
```

The X/Open UNIX standard specifies that each member of the source character set and each escape sequence in character constants and string literals is converted to the same member of the execution character set in all locales. Thus, you can safely use any of the characters in the PCS as a character constant or in string literals. Non-English language characters are not included in the PCS and may not translate correctly when used as literals. Consider the following example:

```
if ((c = getwchar()) == L'à')
```

The accented character à may not be represented in the codeset's source character set or execution character set. Also, the binary value of the accented character may not be translatable from one set to the other. When source files specify non-English language characters in constants, the results

are undefined. In cases such as this, it can be helpful to employ a consistent use of Unicode locales.

The following example illustrates how to construct a test for a constant that for whatever reason may be a non-English language character. The constant has been defined in a message catalog with the symbolic identifier MSG_ID. Statements in the example retrieve the value for MSG_ID from the message catalog, which is locale specific and bound to the program at run time.

```
.
.
.
char *schar;      1
wchar_t wchar;    2
.
.
.

schar = catgets(catd,NL_SETD,MSG_ID,"a"); 3
if (mbtowc (&wchar,schar,MB_CUR_MAX) == -1) 4
        error();
if ((c = getwchar()) == wchar) 5
.
.
.
```

1 Declares a pointer to schar as char.

2 Declares the variable wchar to be of type wchar_t.

3 Calls the catgets( ) function to retrieve the value of MSG_ID from the message catalog for the user's locale.

The catgets( ) function returns a value as an array of bytes so the value is returned to the schar variable. If the accented character is not available in the locale's codeset, the test is made against the unaccented base character (a).

4 Tests to make sure the value contained in schar represents a valid multibyte character. If the value is a valid multibyte character, the program converts it to a wide-character value and stores the results in the variable wchar.

If schar does not contain a valid multibyte character, the program signals an error.

5 Codes the conditional statement to include the value contained in wchar as the constant.

See Chapter 3 for more information about message catalogs and the catgets( ) function. See Section 2.2.4 for information about converting multibyte characters and strings to wide-character data that your program can process.

2.2.3 Manipulating Characters That Span Multiple Bytes

The operating system provides all the interfaces (such as `putwc( )`, `getwc( )`, `fputws( )`, and `fgetws( )`) that are needed to support codesets with characters that span multiple bytes. Language variant subsets of the operating system must be installed to supply the locales and facilities that make this support operational. On systems where such locales are not available, or are available but not bound to the program at run time, the `*ws*( )` and `*wc*( )` functions are merely synonyms for the associated single-byte functions (such as `putc( )`, `getc( )`, `fputs( )`, and `fgets( )`).

2.2.4 Converting Between Multibyte-Character and Wide-Character Data

On an internationalized system, data can be encoded as either multibyte character or wide-character data.

Multibyte encoding is typically used when data is stored in a file or generated for external use or data interchange. Multibyte encoding has the following disadvantages:

- Characters are not represented by a fixed number of bytes for each character, even in the same codeset. Thus, the size of a character in a multibyte data record can vary from one character to the next.
- The parsing rules for retrieving character codes from a multibyte data record are locale dependent.

Because of these disadvantages, wide-character encoding, which allocates a fixed number of bytes for each character, is typically used for internal processing by programs; in fact, internal process code is another way of referring to data in wide-character format. The size of a wide character varies from one system implementation to another. On Tru64 UNIX systems, the size for a wide character is set to 4 bytes (32 bits), a setting that optimizes performance for the HP Alpha processor.

Library routines that print, scan, input, or output text can automatically convert data from multibyte characters to wide characters or from wide characters to multibyte characters, as appropriate for the operation. However, applications almost always have additional statements or requirements for which conversion to and from multibyte characters needs to be explicit.

The following example is from a program module that reads records from a database of employee data. In this case, the programmer wants to process the data in fixed-width units, so uses the `mbstowcs( )` function to explicitly convert an employee's first and last names from multibyte character to wide-character encoding.

```
/*
 * The employee record is normalized with the following format, which
 * is locale independent:  Badge number, First Name, Surname,
 * Cost Center, Date of Join in the 'yy/mm/dd' format. Each field is
 * separated by a TAB. The space character is allowed in the First
 * Name and Surname fields.
 */
static const char *dbOutFormat = "%ld\t%S\t%S\t%S\t%02d/%02d/%02d\n";
static const char *dbInFormat = "%ld %[^\t] %[^\t] %S %02d/%02d/%02d\n";
.
.
.

            sscanf(record, dbInFormat,
                   &emp->badge_num,
                   firstname,
                   surname,
                   emp->cost_center,
                   &emp->date_of_join.tm_year,
                   &emp->date_of_join.tm_mon,
                   &emp->date_of_join.tm_mday);
            (void) mbstowcs(emp->first_name, firstname, FIRSTNAME_MAX+1);
            (void) mbstowcs(emp->surname, surname, SURNAME_MAX+1);
.
.
.
```

See Section A.9 for a complete list of functions that work directly with multibyte data.

2.2.5 Rules for Multibyte Characters in Source and Execution Codesets

Both the source and execution character set variants of the same codeset can contain multibyte characters. The encodings do not have to be the same, but the source and execution variants both observe certain rules in codesets that meet X/Open requirements. PC code pages and UCS-based codesets may adhere to some or most of these rules, but the codesets native to any UNIX system that conforms to X/Open standards must adhere to all of them.

- The characters defined in the Portable Character Set must be present in both sets.
- The existence, meaning, and encoding of any additional members are locale specific.
- A character may have a state-dependent encoding. A string of characters may contain a shift-state character that affects the system's interpretation of the following bytes until another shift-state character is encountered.
- While in the initial shift state, all characters from the basic character set retain their usual interpretation and do not alter the shift state.
- The interpretation for subsequent bytes in the sequence is a function of the current shift state.

- A byte with all bits set to zero is interpreted as a null character, independent of the shift state.
- A byte with all bits zero must not occur in the second or subsequent bytes of a multibyte character.

The source variant of a codeset must observe the following additional rules:

- A comment, string literal, character constant, or header name must begin and end in the initial shift state
- A comment, string literal, character constant, or header name must consist of a sequence of valid multibyte characters

The C language compiler supports trigraph sequences when you specify the `-std1` or `-std` flag on the `cc` command line. Trigraph sequences, which are part of the ANSI C specification, allow users to enter the full range of basic characters in programs, even if their keyboards do not support all characters in the source codeset. The following trigraph sequences are currently defined, each of which is replaced by the corresponding single character:

Trigraph Sequence	Single Character
??=	#
??(	[
??/	\
??'	^
??<	{
??)	]
??!	|
??>	}
??-	~

2.2.6 Classifying Characters

Another feature of program operation that depends on the locale is character classification; that is, determining whether a particular character code refers to an uppercase alphabetic, lowercase alphabetic, digit, punctuation, control, or space character.

In the past, many programs classified characters according to whether the character's value fell between certain numerical limits. For example, the following statement tests for all uppercase alphabetic characters:

```
if (c >= 'A' && c <= 'Z')
```

This statement is valid for the ASCII codeset, in which all uppercase letters have values in the range `0x41` to `0x5a` (A to Z). However, the statement is not valid for the ISO 8859–1 codeset, in which uppercase letters occupy the ranges `0x41` to `0x5a`, `0xc0` to `0xd6`, and `0xd8` to `0xdf`. In the EBCDIC codeset, character values are different again and, in this case, even the uppercase English language letters have a different encoding.

When you write internationalized programs, classify characters by calling the appropriate internationalization function. For example:

```
if (iswupper (c))
```

Internationalization functions classify wide-character code values according to `ctype` information in the user's locale. See Section A.2 for a complete list and description of character classification functions.

2.2.7 Converting Characters

As an example of what not to do in an internationalized program, consider the following statements, which perform case conversion of ASCII characters by converting the character in `a_var` first to lowercase and then to uppercase:

```
a_var |= 0x20;
⋮

a_var &= 0xdf;
```

The preceding statements are not safe to use in internationalized programs because the statements assume ASCII-coded character values and because they can convert invalid values.

The correct way to handle case conversion is to call the `towlower( )` function for conversion to lowercase and the `towupper( )` function for conversion to uppercase. For example:

```
a_var = towlower(a_var);
⋮

a_var = towupper(a_var);
```

These functions use information specified in the user's locale and are independent of the codeset in which characters are defined. The functions return the argument unchanged if input is invalid. See Section A.3 for more detailed discussion of case conversion functions.

2.2.8 Comparing Strings

UNIX systems provide functions for comparing character strings. The following statement, for example, compares the strings `s1` and `s2`, returning an integer greater than, equal to, or less than zero, depending on whether the value of `s1` is greater than, equal to, or less than the value of `s2` in the machine-collating sequence:

```
⋮
int cmp_val;
char *s1;
char *s2;
⋮

cmp_val = strcmp(s1, s2);
⋮
```

Many languages, however, require more complex collation algorithms than a simple numerical sort. For example, multiple passes may be required for the following reasons:

- Ordering accented characters within a particular character class for a language (for example, a, á, à, and so on)
- Collating certain multiple character sequences as a single character (for example, the Welsh character ch, which collates after c and before d)
- Collating certain single characters as a 2-character sequence (for example, the German character sharp s, which collates as ss)
- Ignoring certain characters during collation (for example, hyphens in dictionary words)

String comparison in an international environment depends on the codeset and language. This dependency means that additional functions are required to compare strings according to collating sequence information in the user's locale. These functions include the following:

- `strcoll( )`

  This function uses collation information defined in the user's locale rather than performing a simple numeric comparison as does the `strcmp( )` function.

- `wcscoll( )`

  This function performs the same operation as `strcoll( )`, except that it operates on wide characters.

- `wcsxfrm( )`

This function transforms a wide-character string by using collating sequence information in the user's locale so that the resulting string can be compared using the `wcscmp( )` function.

If two strings are being compared only for equality, you can use `strcmp( )` or `wcscmp( )`, which are faster in most environments than `wcscoll( )`.

2.3 Handling Cultural Data

Cultural data refers to items of information that can vary between languages or territories.

For example:

- In the United Kingdom and the United States, a period represents the **radix character** and a comma represents the thousands separator in decimal numbers. In Germany, the same two characters in decimal numbers have the opposite meaning.
- In the United States, the date October 7, 1986 is represented as 10/7/1986. In the United Kingdom, the same date is represented as 7/10/1986. This example indicates that cultural data items can vary even when the same language is spoken.
- Date delimiters, as well as the order of year, month, and day, can vary among countries. In Germany, for example, the date October 7, 1986 is represented as 7.10.1986 rather than as 7/10/1986.
- Currency symbols can vary both in the characters used and where they are placed in a currency value; that is, currency symbols can precede, follow, or be embedded in the value.

  The euro character that is used as the currency symbol by European countries belonging to the Economic and Monetary Union is supported only by Unicode (`*.UTF-8`) or Latin-9 (`*.ISO8859-15`) locales and associated fonts. See `euro`(5) for complete information about support for the euro currency symbol.

  To enter the euro character from the keyboard, you must be working in a Latin-9 or UTF-8 locale and the appropriate keymap must be active. To display the euro character, you must be working in a Latin-9 or UTF-8 locale and the appropriate font must be active. To activate the required locale and the appropriate keymap and font, log in to a Latin-9 or UTF-8 locale, or use `setenv` to set the `LANG` environment variable, and start a new `dtterm`. See the reference pages for `locale`(1) and `dtterm`(1).

You cannot make assumptions about cultural data when writing internationalized programs. Your program must operate according to the local customs of users. The X/Open UNIX standard specifies that this

requirement be met through a database of cultural data items that a program can access at run time, plus a set of associated interfaces. The following sections discuss this database and the functions used to extract and process its data items.

2.3.1 The langinfo Database

The language information database, named `langinfo`, contains items that represent the cultural details of each locale supported on the system. The `langinfo` database contains the following information for each locale, as required by the X/Open UNIX standard:

- Codeset name
- Date and time formats
- Names of the days of the week
- Names of the months of the year
- Abbreviations for names of days
- Abbreviations for names of months
- Radix character (the character that separates whole and fractional quantities
- Thousands separator character
- Affirmative and negative responses for yes/no queries
- Currency symbol and its position within a currency value
- Emperor/Era name and year (for Japanese locales)

2.3.2 Querying the langinfo Database

You can extract cultural data items from the `langinfo` database by calling the `nl_langinfo( )` function. This function takes an *`item`* argument that is one of several constants defined in the `/usr/include/langinfo.h` header file. The function returns a pointer to the string with the value for *`item`* in the current locale.

The following example is a call to `nl_langinfo( )` that extracts the string for formatting date and time information. This value is associated with the constant `D_T_FMT`.

```
nl_langinfo(D_T_FMT);
```

2.3.3 Generating and Interpreting Date and Time Strings That Observe Local Customs

Programs often generate date and time strings. Internationalized programs generate strings that observe the local customs of the user. You can meet this requirement by calling the `strftime( )` or `wcsftime( )` function. Both functions indirectly use the `langinfo` database. In addition, the `wcsftime( )` function converts date and time to wide-character format.

In the following example, the `strftime( )` function generates a date string as defined by the `D_FMT` item in the `langinfo` database:

```
⋮
setlocale(LC_ALL, ""); [1]
⋮

clock = time((time_t*)NULL); [2]
tm = localtime(&clock); [3]
⋮

strftime(buf, size, "%x", tm); [4]
puts(buf); [5]
⋮
```

[1] Binds the program at run time to the locale set for the system or individual user.

[2] Calls the `time( )` subroutine to return the time value to the `clock` variable. The time value returned is relative to Coordinated Universal Time.

[3] Calls the `localtime( )` function to convert the value contained in `clock` to a value that can be stored in a `tm` structure, whose members represent values for year, month, day, hour, minute, and so forth.

[4] Calls `strftime( )` to generate a date string formatted as defined in the user's locale from the value contained in the `tm` structure.

The `buf` argument is a pointer to a string variable in which the date string is returned. The `size` argument contains the maximum size of `buf`. The `"%x"` argument specifies conversion specifications, similar to the format strings used with the `printf( )` and `scanf( )` functions. The `"%x"` argument is replaced in the output string by a representation appropriate for the locale.

[5] Calls the `puts( )` function to copy the string contained in `buf` to the standard output stream (`stdout`) and to append a newline character.

Consider the following example of how to use `strftime( )` and `nl_langinfo( )` in combination to generate a date and time string. Assume that the preceding example's calls to the `setlocale( )`, `time( )`, and `localtime( )` interfaces have been made in this example. However, the following example includes a call to `nl_langinfo( )` that has replaced the format string argument in the call to `strftime( )`.

```
⋮
strftime(buf, size, nl_langinfo(D_T_FMT), tm);
puts(buf);
⋮
```

To convert a string to a date/time value (that is, the reverse of the operation performed by `strftime( )`), you can use the `strptime( )` function. The `strptime( )` function supports a number of conversion specifiers that behave in a locale-dependent manner.

2.3.4 Formatting Monetary Values

The `strfmon( )` function formats monetary values according to information in the locale that is bound to the program at run time. For example:

```
strfmon(buf, size, "%n", value);
```

This statement formats the double-precision floating-point value contained in the `value` variable. The `"%n"` argument is the format specification that is replaced by the format defined in the run-time locale. The results are returned to the `buf` array, whose maximum length is contained in the `size` variable.

The `money` program demonstrates how the `strfmon( )` function works. When you install a Worldwide Language Support subset, the source file for this sample program is installed in the `/usr/i18n/examples/money` directory.

2.3.5 Formatting Numeric Values in Program-Specific Ways

To perform your own conversions of numeric quantities, monetary or otherwise, you can use specific formatting details in the user's locale. The `localeconv( )` function, which has no arguments, returns all the number formatting details defined in the locale to a structure declared in your program. For example:

```
struct lconv *app_conv;
```

You can use the following features, which are contained in the `lconv` structure, in program-defined routines:

- Radix character
- Thousands separator character
- Digit grouping size
- International currency symbol
- Local currency symbol
- Radix character for monetary values
- Thousands separator for monetary values
- Digit grouping size for monetary values
- Positive sign
- Negative sign
- Number of fractional digits to be displayed
- Parenthesis symbols for negative monetary values

2.3.6 Using the langinfo Database for Other Tasks

Functions in addition to the ones discussed so far use the `langinfo` database to determine settings for specific items of cultural data. For example, the `wscanf( )`, `wprintf( )`, and `wcstod( )` functions determine the appropriate radix character from information in the `langinfo` database.

2.4 Handling Text Presentation and Input

As you create applications, you need to consider the user's native language in three particular areas:

- The way program messages are defined and accessed (Section 2.4.1)
- How the program presents output text (Section 2.4.2)
- How the program processes input text (Section 2.4.3)

2.4.1 Creating and Using Messages

Programs need to communicate with users in their own language. This requirement places some constraints on the way program messages are defined and accessed. More specifically, messages are defined in a file that is independent of the program source code and are not compiled into object files. Because messages are in a separate file, they can be translated into different languages and stored in a form that is linked to the program at run time. Programs can then retrieve message text translations that are appropriate for the user's language.

The X/Open UNIX standard specifies the following messaging functions:

- A messaging system that contains a definition of message text source files
- The `gencat` command to generate message catalogs from these source files
- A set of library functions to retrieve individual messages from one or more catalogs at run time

The following example demonstrates how an internationalized program retrieves a message from a catalog:

```
#include <stdio.h>     1

#include <locale.h>    2
#include <nl_types.h>  3
#include "prog_msg.h"      4
main()
{
      nl_catd catd; 5
      setlocale(LC_ALL, ""); 6
      catd = catopen("prog.cat", NL_CAT_LOCALE); 7
      puts(catgets(catd, SETN, HELLO_MSG, "Hello, world!")); 8
      catclose(catd); 9
}
⋮
```

1 Includes the header file for the Standard C Library.

2 Includes the `/usr/include/locale.h` header file, which declares the `setlocale( )` function and associated constants and variables.

3 Includes the `/usr/include/nl_types.h` header file, which declares the `catopen( )`, `catgets( )`, and `catclose( )` functions.

4 Includes the program-specific `prog_msg.h` header file, which sets constants to identify the message set (SETN) and specific messages (HELLO_MSG in the example) that are used by this program module.

A message catalog can contain one or more message sets. Individual messages are ordered within each set.

5 Declares a message catalog descriptor `catd` to be of type `nl_catd`.

This descriptor is returned by the function that opens the catalog. The descriptor is also passed as an argument to the function that closes the catalog.

6 Calls the `setlocale( )` function to bind the program's locale categories to settings for the user's locale environment variables.

The locale name set for the `LC_MESSAGES` category is the locale used by the `catopen( )` and `catgets( )` functions in this example. Because the system administrator or user typically sets only the `LANG` or `LC_ALL`

environment variable to a particular locale name, this operation implicitly sets the LC_MESSAGES variable as well.

7 Calls the catopen( ) function to open the prog.cat message catalog for use by this program.

The NL_CAT_LOCALE argument specifies that the program will use the locale name set for LC_MESSAGES. The catopen( ) function uses the value set for the NLSPATH environment variable to determine the location of the message catalog. The call returns the message catalog descriptor to the catd variable.

8 Calls the puts( ) function to display the message.

The first argument to this call is a call to the catgets( ) function, which retrieves the appropriate text for the message with the HELLO_MSG identifier. This message is contained in the message set identified by the SETN constant. The final argument to catgets( ) is the default text to be used if the messaging call cannot retrieve the translated text from the catalog. Default text is usually in the English language.

9 Calls the catclose( ) function to close the message catalog whose descriptor is contained in the catd variable.

See Chapter 3 for information about creating and using message catalogs.

2.4.2 Formatting Output Text

Successful translation of messages into different languages depends not only on making messages independent of the program source code but also on careful construction of message strings within the program.

Consider the following example:

```
printf(catgets(catd, set_id, WRONG_OWNER_MSG,
               "%s is owned by %s\n"),
               folder_name, user_name);
```

The preceding statement uses a message catalog but assumes a particular language construction (a noun followed by a verb in passive voice followed by a noun). Passive verb constructions are not part of all languages; therefore, message translation might mean printing user_name before folder_name. In other words, the translator might need to change the construction of the message so that the user sees the translated equivalent of "John_Smith owns JULY_REVENUE" rather than "JULY_REVENUE is owned by John_Smith."

To overcome the problems imposed by fixed ordering of message elements, the printf( ) routine format specifiers can apply format conversion to the *n*th argument in an argument list, and not just to the next unused argument. To apply the format conversion extension, replace the % conversion character with the sequence %*digit* $, where *digit* specifies the position of the

argument in the argument list. The following example illustrates how the programmer applies this feature to the format string "%s is owned by %s\n":

```
printf(catgets(catd, set_id, WRONG_OWNER_MSG,
               "%1$s is owned by %2$s\n"),
               folder_name, user_name);
```

The construction of the string "%1$s is owned by %2$s", which is the default value for the WRONG_OWNER_MSG entry in the program's message file, can then be changed by the translator to the non-English language equivalent of the following:

```
WRONG_OWNER_MSG        "%2$s owns %1$s\n"
```

2.4.3 Scanning Input Text

The string construction issues that are discussed for output text in Section 2.4.2 also apply to input text. For example, different countries have different conventions for the order in which users specify the elements of a date, or differ in the characters that are input to delimit parts of monetary strings. The scanf( ) family of functions support extended format conversion specifiers that allow for variation in the way that users enter elements of a string.

Consider the following example:

```
⋮
int day;
int month;
int year;
⋮

scanf("%d/%d/%d", &month, &day, &year);
⋮
```

The format string in this statement is governed by the assumption that all users use a United States format (mm/dd/yyyy) to input dates. In an internationalized program, you use extended format specifiers to support requirements that language may impose on the order of string elements. For example:

```
⋮
scanf(catgets(catd, NL_SETD, DATE_STRING,
              "%1$d/%2$d/%3$d"), &month, &day, &year);
⋮
```

The default "%1$d/%2$d/%3$d" value for the DATE_STRING message is still appropriate only for countries in which users use the format mm/dd/yyyy to enter dates. However, for countries in which the order or formatting would be different, the translator can change the entry in the program's message file. Consider the following languages:

- British English (dd/mm/yyyy):

```
DATE_STRING        "%2$d/%1$d/%3$d"
```

- German (dd.mm.yyyy)

```
DATE_STRING        "%2$d.%1$d.%3$d"
```

2.5 Binding a Locale to the Run-Time Environment

A correct, operational internationalized program must bind to localized data that is appropriate for the user at run time. The `setlocale( )` function performs this task. You can call `setlocale( )` to perform the following operations:

- Bind to locale settings that are already in effect for the user's process
- Bind to locale settings controlled by the program
- Query current locale settings without changing them

The call takes two arguments: *category* and *locale_name*.

The *category* argument specifies whether you want to query, change, or use all or a specific section of a locale. Values for *category* and what they represent are as follows:

- LC_ALL

  This *category* argument specifies all sections of a locale (overrides specifications for specific sections).

- LC_CTYPE

  This *category* argument defines classes and character attributes used in case conversion and similar operations.

- LC_COLLATE

  This *category* argument specifies how to order characters and strings in sorting, or collation, operations.

- LC_MESSAGES

  This *category* argument specifies yes/no responses and program messages.

- LC_MONETARY

  This *category* argument specifies rules and special symbols for use in monetary values.

- LC_NUMERIC

  This *category* argument specifies rules and special symbols used for formatting numeric values.

- LC_TIME

  This *category* argument specifies names and abbreviations for days of the week, months of the year, and other strings and formatting conventions that govern expressions of date and time.

The *locale_name* argument is one of the following values:

- An empty string ("") that binds the program at run time to the locale name set for *category* by the system administrator or user
- A locale name that changes the locale that may already be set for *category*
- NULL that determines the locale name currently set for *category*

2.5.1 Binding to the Locale Set for the System or User

Typically, the system administrator or user sets the LANG or LC_ALL environment variable to the name of a locale. When you set either of these variables, it automatically sets all locale category variables to the same locale name.

Except for the case in which LC_ALL has been used to set all locale categories to a single locale name, system administrators or individual users can set locale category variables to different locale names. Usually, internationalized programs contain the LC_ALL call, which initializes all locale categories in the program to environment variable settings already in effect for the user. For example:

```
setlocale(LC_ALL, "");
```

A standard locale name consists of language_TERRITORY.codeset@modifier, for example, zh_CN.dechanzi@radical, where:

- language represents the human language of the locale (zh is Chinese)
- _TERRITORY is the geographic country or region of the locale (_CN is China, as opposed to TW for Taiwan or HK for Hong Kong)
- .codeset is the coded character set used by the locale (dechanzi)
- @modifier is additional information for localization data of a locale (collation by radical)

Locales often have multiple variants. These variants have the same name as the base locale but include a file name suffix that begins with the at sign (@). Locale variants for support of codesets that are not native to UNIX (such as UCS-4 and CP850), can be assigned to LANG or LC_ALL.

However, locale variants that differ from the base locale in only one locale category should be assigned only to the appropriate locale category. For example, a locale variant designed to support a specific collation sequence, such as `@radical`, would be assigned to `LC_COLLATE`. A locale variant designed to support the euro monetary sign (`@euro`) would be assigned to `LC_MONETARY`. Use the base locale name, not these variants, in assignments to the `LANG` environment variable.

Furthermore, in cases where a base locale name is not being assigned to all locale categories, avoid using the `LC_ALL` environment variable, whose assigned value overrides settings for both `LANG` and the environment variables for specific locale categories.

Many locale-specific files reside in directories whose names are constructed from the language, territory, and codeset portions of a locale name. Commands and other system applications insert the setting of the `LANG` variable into search paths that contain `%L` as one of the directory nodes. This makes it possible for software programs to find the correct set of files, such as fonts, resource files, user-defined character files, and translated reference pages, that should be used with the current locale. An @ suffix related to collation, if included in an assignment to the `LANG` variable, may result in applications being unable to find certain locale-specific files.

2.5.2 Changing Locales During Program Execution

Some internationalized programs may need to prompt the user for a locale name or change locales during program execution. The following example demonstrates how to call `setlocale( )` when you want to explicitly initialize or reinitialize all locale categories to the same locale name:

```
⋮
nl_catd catd; 1
char buf[BUFSIZ]; 2
⋮

setlocale(LC_ALL, ""); 3
catd = catopen(CAT_NAME, NL_CAT_LOCALE); 4
⋮

printf(catgets(catd, NL_SETD, LOCALE_PROMPT_MSG,
               "Enter locale name: ")); 5
gets(buf); 6
setlocale(LC_ALL, buf); 7
⋮
```

1 Declares a catalog descriptor `catd` as type `nl_catd`.

2 Declares the `buf` variable into which the locale name will later be stored.

   To make sure that the variable is large enough to accommodate locale names on different systems, you should set its maximum size to the `BUFSIZ` constant, which is defined by the system vendor in `/usr/include/stdio.h`.

3 Calls `setlocale( )` to initialize the program's locale settings to those in effect for the user who runs the program.

4 Calls `catopen( )` to open the message catalog that contains the program's messages. The function returns the catalog's descriptor to the `catd` variable.

   The `CAT_NAME` constant is defined in the program's own header file.

5 Prompts the user for a new locale name.

   The `NL_SETD` constant specifies the default message set number in a message catalog and is defined in `/usr/include/nl_types.h`. The `LOCALE_PROMPT_MSG` identifier specifies the prompt string translation in the default message set.

6 Calls the `gets( )` function to read the locale name typed by the user into the `buf` variable.

7 Calls `setlocale( )` with `buf` as the *`locale_name`* argument to reinitialize all portions of the locale.

Sometimes a program needs to vary the locale only for a particular category of data. For example, consider a program that processes different country-specific files that contain monetary values. Before processing data in each file, the program might reinitialize a program variable to a new locale name and then use that variable value to reset only the `LC_MONETARY` category of the locale.

3

Creating and Using Message Catalogs

A **message catalog** is a file of localization data that programs can access. While the same definition applies to the `langinfo` database, there are differences between the two.

The localization data elements in the `langinfo` database are used by all applications, including the library routines, commands, and utilities provided by the operating system. The `langinfo` database is generated from the source files that define locales.

In contrast to the `langinfo` database, message catalogs meet the specific localization needs of one program or a set of related programs. Message catalogs are generated from message text source files that contain error and informational messages, prompts, background text for forms, and miscellaneous strings and constants that must vary for language and cultural reasons.

X and Motif applications with graphical user interfaces, usually access X resource files, rather than message catalogs, for the small segments of text that belong to the title bars, menus, buttons, and simple messages for a particular window. Motif applications can also use a user interface language (UIL) file, along with a text library file, to access help, error message, and other kinds of text. However, both X and Motif applications can access text in message catalogs as well.

This chapter focuses on message catalogs.

- Section 3.1.1 contains general guidelines you can apply to defining the contents of message text source files.
- Section 3.1.2 describes message sets, an optional component of message text source files that you use to group messages.
- Section 3.1.3 describes the message entries that comprise a message text source file.
- Section 3.1.4 describes the quote directive and Section 3.1.5 describes comment lines that you use to delimit text or enter nonexecutable comments in message text source files.
- Section 3.1.6 contains style guidelines to use when you create message text.
- Section 3.2 describes how to extract message text from existing programs.

- Section 3.3 describes how to edit and translate message text source files.
- Section 3.4 describes how to generate message catalogs, including the use of the `mkcatdefs` and `gencat` commands, and hints for designing and maintaining message catalogs.
- Section 3.5 describes how to display messages and locale data interactively and from scripts.
- Section 3.6 describes how to access message catalogs from programs, including the use of `catopen( )`, `catclose( )`, and `catgets( )` functions to open, close, and read message catalogs.

See Section 3.1.6 for X and Motif programming guidelines that apply to the translation of message catalog text, regardless of the method used to retrieve and display the text.

3.1 Creating Message Text Source Files

Before creating and using a message catalog, you must first understand the components, syntax, and semantics of a message text source file. A brief overview of a source file example can help provide context for later sections of this chapter, which focus on particular kinds of file entries and processing operations. Example 3–1 contains extracts from a message text source file for the online example, `xpg4demo`.

Example 3–1: Message Text Source File

```
$ /*  1
$  * XPG4 demo program message catalogue.   1
$  *  1
$  */ 1
2
$quote "  3
$set MSGError  4
E_COM_EXISTBADGE        "Employee entry for badge number %ld \  5
already exists"
E_COM_FINDBADGE         "Cannot find badge number %ld"  5
E_COM_INPUT             "Cannot input"  5
E_COM_MODIFY            "Data file contains no records to modify"  5
E_COM_NOENT             "Data file contains no records to display" 5
E_COM_NOTDEL            "Data file contains no records to delete"  5
.
.
.

$set MSGInfo  4
I_COM_NEWEMP            "New employee"    5
I_COM_YN_DELETE         "Do you want to delete this record?"  5
I_COM_YN_MODIFY         "Do you want to modify this record?"  5
I_COM_YN_REPLACE        "Are these the changes you want to make?"  5
.
.
.

$ NOTE - Message contains the format used to display numeric dates
$ The first descriptor, 1$, contains the year
$ The second descriptor, 2$, contains the month
$ The third descriptor, 3$, contains the day
```

Example 3–1: Message Text Source File (cont.)

```
I_SCR_IN_DATE_FMT       "%2$d/%3$d/%1$d"  6
$set MSGString  4
$
$ One-character commands.
$ Note: These should not be translated because they are keywords for the application.
$S_COM_CREATE           "c"    7
S_COM_DELETE            "d"    7
S_COM_EXIT              "e"    7

.
.
.

$ Note: These are column heads and spacing and should be maintained
$ Column one begins at space 1.
$ Column two begins at space 15.
$ Column three begins on space 37.
$ Column four (an abbreviation of Department) begins at space 60.
$ Column five (an abbreviation of Date of Birth) begins at space 68.
$ S_COM_LIST_TITLE is output to underscore headers and should be
$ increased or decreased as appropriate for translation.
S_COM_LIST_TITLE        "Badge          Name                  Surname \
               Dept      DOB\n"   8
S_COM_LIST_LINE         "-----------------------------------------\
--------------------------------\n"   8
.
.
.
$
$ If surname comes before first name, "y" should be specified.
$
S_SCR_SNAME1ST          "n"    9
.
.
.
```

1 Lines that begin with the dollar sign ($), followed by either a space or tab, are comment lines. Section 3.1.5 discusses comment lines.

2 To improve readability, blank lines are allowed anywhere in the file.

3 The quote character delimits message text. Section 3.1.4 discusses quote directives.

4 Identifiers are used to mark the beginning of a message set. There are three sets of messages in this source file: error messages (in the MSGError set), informational messages (in the MSGInfo set), and miscellaneous strings and formats (in the MSGString set). See Section 3.1.2 for more information about defining and removing message sets.

5 Most lines in the source file are message entries, whose components are a unique identifier and a message text string. The first message entry is continued to the next line by using the backslash (\). Other entries contain special character sequences, such as \n (newline), that affect how the message is printed. See Section 3.1.3 for more information

about message entries. Section 3.1.1 also discusses some rules and options that apply to message entries.

6 This type of message entry allows translators to vary the order in which users are prompted to enter date elements. You frequently use message entries to allow format control, although use of program logic to format messages is a better alternative. This line also illustrates the value of providing comments that identify variables to potential translators.

7 This type of message entry defines word abbreviations, which often need special attention to preserve uniqueness from one language to another.

8 This type of message entry defines header lines for menu displays so that translators can adjust the field order and line length to match other adjustments that the program allows for cultural variation. This line also illustrates the value of providing comments to translators who may be unfamiliar with abbreviations or who need to know the amount of spacing in the formatting of columns.

9 This type of message entry defines a constant whose value controls how the program positions name fields. For example, in the `xpg4demo` program, you can change the position of first and last name (surname).

You can use one or more message text source files to create message catalogs (`.cat` files) that programs can access at run time. To create a message catalog from the source file in Example 3–1, perform the following tasks:

1. Use the `mkcatdefs` command to convert symbolic identifiers for message sets and messages to numbers that indicate the ordinal positions of the message sets within the catalog and of messages within each set.
2. Use the `gencat` command to create the message catalog from `mkcatdefs` output.

Section 3.4 discusses the `mkcatdefs` and `gencat` commands.

3.1.1 General Rules

This section contains general guidelines that apply to the syntax of message text source files. Section 3.1.6 contains stylistic guidelines for the content of message text.

A message text source file (`.msg` file) contains sequences of messages. Optionally, you can order these messages within one or more message sets. For a given application, there are usually separate message source files for each localization; for example, there are source files for each locale (each combination of codeset, language, and territory) with which users can run the application.

If you do not quote values for identifiers, specify a single space or tab, as defined by the source codeset, to separate fields in lines of the source file. Otherwise, the extra spaces or tabs are treated as part of the value. Using the character specified in a `quote` directive to delimit all message strings prevents extra spaces or tabs between the identifier and the string from being treated as part of the string (see Section 3.1.4 for a description of the `quote` directive). Quoting message strings is also the only way to indicate that the message text includes a trailing space or tab.

Message text strings can contain ordinary characters plus sequences for special characters, as described in Table 3–1.

Table 3–1: Coding of Special Characters in Message Text Source Files

Description	Symbol	Coding Sequence
Newline	NL (LF)	\n
Horizontal tab	HT	\t
Vertical tab	VT	\v
Backspace	BS	\b
Carriage return	CR	\r
Form feed	FF	\f
Backslash	\	\\
Octal value	ddd	\ddd [a]
Hexadecimal value	dddd	\xdddd [b]

[a] The escape sequence \ddd consists of a backslash followed by one, two, or three octal digits that specify the value of the desired character.
[b] The escape sequence \xdddd consists of a backslash followed by the character x and one, two, three, or four hexadecimal digits that specify the value of the desired character. The hexadecimal coding sequence is an extension to X/Open UNIX CAE specifications and may not be supported on other systems that conform to these specifications.

A backslash in a message file is ignored when followed by coding sequences other than those described in Table 3–1. For example, the sequence `\m` prints in the message as `m`. When you use octal or hexadecimal values to represent characters, include leading zeros if the characters following the numeric encoding of the special character are also valid octal or hexadecimal digits. For example, to print $5.00 when 44 is the octal number for the dollar sign, you must specify `\0445.00` to prevent the `5` from being parsed as part of the octal value.

A newline character normally separates message entries. However, you can continue the same message string from one line to another by entering a backslash before the newline character. In this context, entering a newline character means pressing the Return or Enter key on English language

keyboards. For example, the following two entries are equivalent and do not affect how the string appears to the program user:

```
MSG_ID        This line continues \
to the next line.
MSG_ID        This line continues to the next line.
```

Any empty lines in a message source file are ignored. Thus, you can use blank lines to improve the readability of the file.

3.1.2 Message Sets

Message sets are an optional component within message text source files. You can use message sets to group messages for any reason. In an application built from multiple program source files, you can create message sets to organize messages by program module or, as done for the online example `xpg4demo`, group messages that belong to the same semantic category (error, informational, defined strings).

An advantage of grouping messages by program module is that, should the module later be removed from the application, you can easily find and delete its messages from the catalog.

Grouping messages by semantic category supports message sharing among modules of the same application. When messages are grouped by semantic category, programmers writing new modules or maintaining existing modules for an application can easily determine if a message meeting their needs already exists in the file.

A set directive specifies the set identifier of subsequent messages until another set directive or end-of-file is encountered. Set directives have the following format:

`$SET` *`set_id`* [*`comment`*]

The *`set_id`* variable can be one of the following:

- A number in the range `[1 - NL_SETMAX]`

  The `NL_SETMAX` constant is defined in the `/usr/include/limits.h` file. Numeric set identifiers must occur in ascending order within the source file; however, the numbers need not be contiguous values. Furthermore, set identifier numbers must occur in ascending order from one source file to the next when multiple message source files are processed by the `gencat` command to create a message catalog.

- A user-defined symbolic identifier, such as `MSGErrors`

  When you specify symbolic set identifiers, you must use the `mkcatdefs` command to convert the symbols to the numeric set identifiers required by the `gencat` command.

Any characters following the set identifier are treated as comments.

If the message text source file contains no set directives, all messages are assigned to a default message set. The numeric value for this set is defined by the constant NL_SETD in the /usr/include/nl_types.h file. When a program calls the catgets( ) function to retrieve a message from a catalog that has been generated from sources that do not contain set directives, the NL_SETD constant is specified on the call as the set identifier.

Note

Do not specify NL_SETD in a set directive of a message text source file or try to mix default and user-defined message sets in the same message catalog. Doing so can result in errors from the mkcatdefs or gencat utility. Furthermore, the value assigned to the NL_SETD constant is vendor defined; using NL_SETD as a symbolic identifier in the message text source file can result in mkcatdefs output that is not portable from one system to another.

The rest of this section discusses entries that delete message sets from an existing message catalog. Section 3.4.3 addresses the topic of catalog maintenance more generally.

Message text source files can contain delset directives, which are used to delete message sets from existing message catalogs. The delset directive has the following format:

```
$delset n [ comment]
```

The *n* variable must be the number that identifies the set in the existing catalog to the gencat command. Unlike the case for the set directive, you cannot specify symbolic set identifiers in delset directives. When message files are preprocessed using the mkcatdefs command, you have the option of creating a separate header file that equates your symbolic identifiers with the set numbers and message numbers assigned by the mkcatdefs utility. If you later want to delete one of the message sets, you first refer to this header file to find the number that corresponds to the symbolic identifier for the set you want to delete. This is the number that you specify in the delset directive to delete that set.

Suppose that you are removing program module a_mod.c from an application whose associated message text source file is appl.msg. Messages used only by a_mod.c are contained in the message set whose symbolic identifier is A_MOD_MSGS. The file appl_msg.h contains the following definition statement:

```
⋮
#define A_MOD_MSGS 2
⋮
```

The associated `delset` directive could then be the following:

```
$delset 2   Removing A_MOD_MSG set for a_mod.c in appl.cat.
```

You can specify `delset` directives either in a source file by themselves or as part of a more general message source file revision that includes both `delset` and `set` directives. In the latter case, make sure that multiple directives occur in ascending order according to the specifier.

Assume that the preceding example is contained in a single-directive source file named `kill_mod_a_msgs.msg` and existing message catalogs reside in the `/usr/lib/nls/msg` directory. In this case, the following `ksh` loop would carry out the message set deletion in catalogs for all locales:

```
for i in /usr/lib/nls/msg/*/appl.cat
do
        gencat $i kill_mod_a_msgs.msg
done
```

3.1.3 Message Entries

A message entry has the following format:

msg_id message_text

The *msg_id* can be either of the following:

- A number in the range `[1 - NL_MSGMAX]`

  The constant `NL_MSGMAX` is defined in the `/usr/include/limits.h` file. Message numbers are associated with the message set defined by the preceding `set` directive or, if not preceded by a `set` directive, with the default message set `NL_SETD`, a constant defined in the `/usr/include/nl_types.h` file.

  Message numbers must occur in ascending order within a message set; however, the numbers need not be contiguous values. If message numbers are not in ascending order within a set, the `gencat` command returns an error on attempts to generate a message catalog from the source file.

- A user-defined symbolic name, for example, `ERR_INVALID_ID`

  When a message text source file contains symbolic names, you must use the `mkcatdefs` command to convert the symbolic names to numbers that the `gencat` command can process.

The *message_text* is a string that the program refers to by *msg_id*. You can quote this string if a `quote` directive enables a quotation character before the message entry is encountered. Section 3.1.1 discusses the advantages of quoting message text. Section 3.1.4 lists the rules for `quote` directives.

The total length of *message_text* cannot exceed the maximum number of bytes defined for the `NL_TEXTMAX` constant in the `/usr/include/limits.h` file.

The rest of this section discusses entries that delete specific messages from an existing message catalog. See Section 3.4.3 for a general discussion of message catalog maintenance.

To delete a particular message from an existing message catalog, enter the identifier for the message on a line by itself. This type of entry allows you to delete a message without affecting the ordinal position of subsequent messages. For the message deletion to be carried out correctly, use the following guidelines:

1. Specify a numeric message identifier.

   If you usually use symbolic identifiers in your message text source files, you can obtain the associated numbers from the message header file that is produced when the source file was last processed by the `mkcatdefs` command. Unlike the case for deleting message sets with the `delset` directive, `mkcatdefs` does not generate an error if you use a symbolic message identifier to delete a message; however, you will delete the wrong message if the symbol is not preceded by the same number of message entries as is in the catalog.
2. The identifier cannot be followed by any character other than a newline. If *msg_id* is followed by a space or tab separator, the message is not deleted; rather, the message text is revised to be an empty string.
3. If the catalog contains user-defined message sets, make sure the appropriate `set` directive precedes the entry to delete the message; otherwise, the message may be deleted from the wrong message set. For reasons similar to those noted for message identifiers in step 1, use a numeric rather than symbolic set identifier in the `set` directive.
4. Unless you are replacing all messages in a set, use only the `gencat` command to process the file. To replace all messages in a set, use the `mkcatdefs` utility, which generates a `delset` directive before each `set` directive you specify in the input file. This is helpful when you want to replace all messages in a message set, but it will not produce the results you intend if your input source refers only to one or two messages that you want to delete.

Consider the following two examples:

- This example uses message text source input processed with the `gencat` command. The command in this example results in the deletion of message 5 from message set 2.

  ```
  $set 2
  5
  ```

- This example uses the same source input. However, in this case, the source is preprocessed with the `mkcatdefs` command. The addition of the `delset` directive results in the deletion of all messages in set 2 from the message catalog.

  ```
  $delset 2
  $set 2
  5
  ```

3.1.4 Quote Directive

A `quote` directive enables or disables a quote character that you use to surround message text strings. The `quote` directive has the following format:

`$quote[` *`character`*`]`

The *`character`* variable is the character to be recognized as the message string delimiter. In the following example, the `quote` directive specifies the double quotation mark as the message string delimiter:

```
$quote "
```

By default, or if a *`character`* is omitted, quoting of message text strings is not recognized.

A source text message file can contain more than one `quote` directive, in which case each directive affects the message entries that follow it in the file. Usually, however, a message file contains only one `quote` directive, which occurs before the first message entry.

3.1.5 Comment Lines

A line beginning with the dollar sign ($) followed by a space or tab is treated as a comment. Neither the `mkcatdefs` nor the `gencat` commands interpret comment lines.

Remember that message files may be translated by individuals who are not programmers. Be sure to include comment lines with instructions to translators on how to handle message entries whose strings contain literals and substitution format specifiers. For example:

```
$ Note to translators: Translate only the text that is within
$ quotation marks ("text text text") on a given line.
$ If you need to continue your translation onto the next line,
```

```
$ type a backslash (\) before pressing the newline
$ (Return or Enter) key to finish the message.
$ For an example of line continuation, see the
$ line that starts with the message identifier E_COM_EXISTBADGE.
.
.
.

$ Note to translator: When users see the following message, a badge
$ number appears in place of the %ld directive.
$ You can move the %ld directive to another position
$ in the translated message, but do not delete %ld or replace %ld with
$ a word.
$
E_COM_EXISTBADGE        "Employee entry for badge number %ld \
already exists"
.
.
.

$
$ Note to translator:  The item %2$d/%1$d/%3$d indicates month/day/year
$ as expressed in decimal numbers; for example, 3/28/81.
$ To improve the appropriateness of this date input format, you can change
$ only the order of the date elements and the delimiter (/).
$ For example, you can change the string to %1$d/%2$d/%3$d or
$ %1$d.%2$d.%3$d to indicate day/month/year or day.month.year
$ (28/3/81 or 28.3.81).
$
I_SCR_IN_DATE_FMT        "%2$d/%1$d/%3$d"
.
.
.
```

The operating system provides the `trans` utility, discussed in Section 3.3, to help translators quickly locate and edit the translatable text in a message source file. This utility does not eliminate the need for information from the programmer on message context and program syntax.

3.1.6 Style Guidelines for Messages

When creating messages and other text strings in the English language, keep the following information in mind:

- Text strings in the English language are usually shorter than equivalent text strings in other languages. When text strings are translated, their length can increase an average of 30 to 40 percent. Expect even larger percentage increases for strings containing fewer than 20 characters.

  The following guidelines address the likelihood that text strings will grow when translated from the English language to another language:

  – If you must limit a text string to one line (for example, 80 characters), make sure the English language text occupies no more than half of the available space. Whenever possible, allow text to wrap to a subsequent line rather than restricting it to an arbitrary length.

  – Do not design a menu, form, screen, or window in which English language text uses most of the available space.

- Design a dialog box so that its components can be moved around. The developers who localize your application may have to reorganize the contents of a dialog box because of text length changes and, for Asian languages, to accommodate Asian character input.
- Do not embed text in a graphic. If text is embedded in a graphic, the entire graphic must be redone when the application is localized. Furthermore, the translated text may cause the graphic to grow in size or to lose visual appeal.

- Nouns in languages other than English may have gender that affects the spelling of the noun itself and associated adjectives and verbs. The way a noun is spelled can also change, depending on whether the noun is the subject or object of a verb, or the object of a preposition. There can be additional grammatical rules, such as those for creating affirmative, negative and imperative verb forms, that are different from the English language. These conditions lead to the following rules:
  - Do not create a message at run time by concatenating different kinds of strings. For example, do not concatenate strings that represent different nouns, adjectives, verbs, or combinations of these.

    If adjectives and verbs can have multiple referents, each with a different gender, the translator may not be able to create a grammatically correct counterpart for all the possible sentences that the user may see. In this case, the developer who is localizing the application may have to redesign the error-handling logic so that the application returns several distinct messages rather than one.
  - Be careful about inserting the same text variable into different strings. Word spelling may have to change if each string represents a different grammatical context. Furthermore, you cannot assume that there is a one-to-one correspondence between English language words and their counterparts in other languages. For example, you can create a negative statement in the English language by creating a text variable that contains the word “not” and inserting that variable into a verb phrase. The message could not be translated to the French language, however, which usually requires two words, “ne” before the verb and “pas” after the verb, to negate meaning.

    Pathnames, file names, and strings that are complete sentences are usually safe to insert into other strings.
  - Avoid using the word “None” as a button label or menu item; this word may be impossible to translate if its referents have different gender.
  - In general, create messages that are complete sentences. Because of differences in grammatical conventions from language to language, building messages from fragments can create translation issues.

If the message is composed of a component that identifies a system entity (a command, utility, error severity level, server, and so forth) and a separate component that contains informational or error text, you can break the rule about starting messages with a verb. In this case, be sure to include comments to the translator in your message source file about how the message components are constructed and about the system entity referenced in the message. Also, use grammatically complete phrases for the informational or error text component. See Section 3.1.5 for information about adding comments to message source files.

– Do not start messages with a verb (unless the message is an imperative where the subject "you" is understood).

  The following messages cannot be translated into some languages because the translator cannot determine the subject of the sentence or the correct form of the verb in the local language:

```
Is a directory.

Could not open file.
```

- Unique identifiers that are based on the first letters of words may not be unique when the words are translated. For example, a common practice in applications that prompt users to choose among several items is to accept a single character as the item identifier. Make sure your application does not require this character to be the first character or first several characters in the item name. The translator should have the option of substituting any character or a number for the item identifier.
- Languages can have syntax rules that require translators to change word order. Therefore, use substitution specifiers as described in Section 2.4.2 so that translators can change the order of message components to meet local language requirements.
- Translations of messages with vague, ambiguous, or telegraphic wording are likely to be incorrect. Use the following guidelines to help ensure accurate translation:
  – Include documentation in the message file, just as you would for a program source file. Provide comments that describe sentence constructions and that clarify any wording that might be misconstrued by a non-native speaker.
  – Include articles (the, a, an) and forms of the verb "to be" where appropriate. Programmers often omit these words to reduce the size of message strings; however, the omission sometimes makes it difficult to distinguish nouns from verbs, subject nouns from predicate nouns, and active voice from passive voice. The message "Maximum parameter count exceeded" illustrates this problem.

- You can include very common contractions, such as “can’t” and “don’t”, but avoid less commonly used contractions, like “should’ve”. If you are using contractions in the English language to conserve line space, be aware that your objective is likely to be lost in translation.
- Avoid using most abbreviations that programmers commonly use in variable names and code comments. In particular, avoid such terms as pkt, msg, tbl, ack, and max. These abbreviations do not appear in a dictionary, and translators may have to guess at what they mean. On the other hand, you can use formal abbreviations for product and utility names and acronyms (such as ANSI or TCP/IP for names of standards, protocols, and so forth that appear in commercial literature).
- Use grammatically correct words. English langugage speakers have a tendency to create new verbs or adjectives out of existing nouns and new nouns out of existing verbs. This practice is confusing to translators, particularly when the intended usage is not one of those noted in an English language dictionary. For example, consider the use of the word “parameter” as an adjective in the message “Invalid parameter delimiter.”
- Avoid using slang or words whose intended meaning is not included in a dictionary. Slang usually has no equivalent in another language or can be misinterpreted. For example, the message “Server hang” may be meaningful to English language speakers who develop software or manage systems, but the meaning of the message may be transformed in another language to “The system lynched the waiter.” The message “The %s server failed.” is more likely to be translated correctly.

- In general, use positional format elements in message files. However, if the message contains only one format flag, a positional element adds no value and tends to confuse the translator. Message files that contain positioning format elements should be heavily commented to help the translator understand the intended result.

3.2 Extracting Message Text from Existing Programs

If you have an existing program that you want to internationalize, the operating system provides the following tools to help you extract message strings into a message source file and to change calls to retrieve messages from a message catalog:

Tool	Description
`extract` command	Interactively extracts text strings from program source files and writes each string to a source message file. The command also replaces each extracted string with a call to the `catgets( )` function.
`strextract` command	Performs string extraction operations in batch.
`strmerge` command	Reads strings from the message file produced by `strextract` and, in the program source, replaces those strings with calls to the `catgets( )` function.

Consider the following call:

```
printf("Hello, world\n");
```

You can use the `extract` command, or the `strextract` command followed by the `strmerge` command, to do the following:

- Create the following entries in a message text source file (assuming that "Hello, world" was the first string extracted):

  ```
  $set 1
  $quote "
  1 "Hello, world\n"
  ```

- Change the `printf( )` call to the following:

  ```
  printf(catgets(cat, 1, 1, "Hello, world\n"));
  ```

Assuming that input to the commands is a program source file named `prog.c`, the commands create the following three new files: `prog.msg` (message text source file), `nl_prog.c` (internationalized version of the program source), and `prog.str` (an intermediate strings file that other utilities can reference). The commands use the following files along with the input source program:

- A patterns file

  This file specifies patterns that the extraction commands use to find strings in the program. You can specify your own patterns file. By default, the extraction commands use the `/usr/lib/nls/patterns` file.

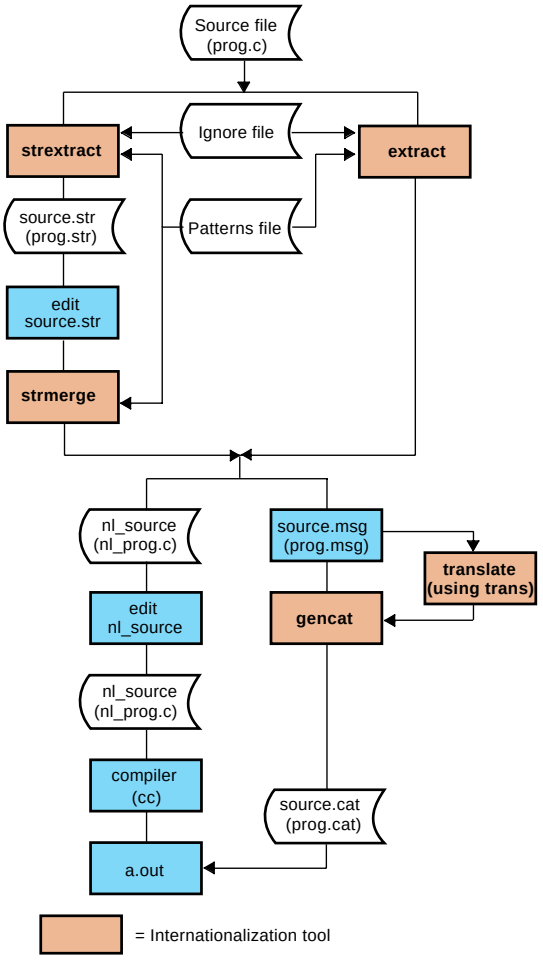
- An optional ignore file

  This file specifies strings that the extraction commands should ignore.

The `extract`, `strextract`, and `strmerge` commands do not perform all the revisions necessary to internationalize a program. For example, you must manually edit the revised program source to add calls to `setlocale( )`, `catopen( )`, and `catclose( )`. In addition, you may need to add routines for multibyte character conversion (for Asian locales) and improve user-defined routines to vary behavior according to values defined in message catalogs or in the `langinfo` database.

Figure 3–1 illustrates the files and tools that help you change an existing program to use a message catalog. For detailed instructions on using the `extract`, `strextract`, and `strmerge` commands, see `extract`(1), `strextract`(1), `strmerge`(1), and `patterns`(4).

Figure 3–1: Converting an Existing Program to Use a Message Catalog



ZK-0045U-AI

3.3 Editing and Translating Message Source Files

You can use any text editor to edit message text source files, provided that the following is true:

- The input device is capable of generating the necessary characters.

- If 8-bit or multibyte characters are required, the editor can transparently handle this data.

The requirement on input devices is satisfied for languages other than Western European by terminal drivers, locales, fonts, and other components that are available with localized software subsets.

The requirement for transparent handling of 8-bit and multibyte data is satisfied by the `ed`, `ex`, and `vi` editors. Localized software subsets may also include enhanced versions of additional editors, such as Emacs, that can handle 8-bit and multibyte characters.

The operating system includes the `trans` command to assist those who translate message text source files for different locales. The command provides a multiwindow environment so users can see both the original and translated versions of the file. In addition, the command automatically guides users in the file from one translatable string to the next. For more information, see `trans`(1).

See Section 3.1.5 for examples of comments to include in message text source files to ensure that messages are correctly translated.

For examples of translated message text source files, search the `/usr/examples/i18n/xpg4demo/` directory for `*.msg` files, as follows:

```
% cd /usr/examples/i18n/xpg4demo/
% ls *.msg
⋮
```

A translated message catalog is associated with a particular locale and encoding format. Many languages are supported by multiple locales and encoding formats, and this generates a requirement that messages in the same language be available in multiple encoding formats. Although you can use codeset converters to convert message source files, building and installing multiple versions of the same catalog for a single language is expensive. Therefore, the `catopen( )` and `catgets( )` functions support dynamic codeset conversion of message catalogs. A set of `.msg_conv-`*`locale_name`* files in the `/usr/share` directory controls codeset conversion of message catalogs. See `catopen`(3) for detailed information.

3.4 Generating Message Catalogs

The `gencat` command generates message catalogs from one or more message text source files. If the source files contain symbolic rather than numeric identifiers for message sets, message entries, or both, those source files must first be preprocessed by the `mkcatdefs` command. Example 3–2 illustrates interactive processing of message text source files with symbolic identifiers

for a default and nondefault locale. This example provides context for later sections, which discuss each command.

Example 3–2: Generating a Message Catalog Interactively

```
% mkcatdefs xpg4demo xpg4demo.msg | gencat xpg4demo.cat  1
mkcatdefs: xpg4demo_msg.h created   2
% setenv LANG fr_FR.ISO8859-1   3
% mkdir fr_FR   4
% mkcatdefs xpg4demo xpg4demo_fr_FR.msg -h | gencat \
fr_FR/xpg4demo.cat    5
mkcatdefs: no msg.h created   6
```

1 The `mkcatdefs` command specifies the following:

- The root name to use for the header file

  The header file maps symbolic identifiers used in the program to their numeric values in the message catalog.

- The name of the message text source file being processed

The preprocessed message source is piped to the `gencat` command, which specifies the name of the message catalog.

2 The `mkcatdefs` command prints to standard output the name of the header file it creates. The utility appends `_msg.h` to the root name to create a name for the header file.

3 When generating a message file for a nondefault locale, you must set the `LANG` environment variable to the name of the locale that the message catalog will support, in this case, `fr_FR.ISO8859-1`.

4 Because the name of the message catalog opened by the program does not vary by locale name, you must create a directory in which to store each message catalog variant.

5 This line creates the local variant of the message catalog. The header file created by the `mkcatdefs` utility does not vary by locale. The header file has already been created for the default message catalog, so this `mkcatdefs` command includes the `-h` flag to disable creation of another header file. The catalog specified to the `gencat` command is directed to the temporary locale directory. On user systems, you can move this version of the catalog to the `/usr/lib/nls/msg/fr_FR.ISO8859-1` default directory or to a directory that is application specific.

6 The `mkcatdefs` command announces that no header file has been created, as intended.

See the /usr/examples/i18n/xpg4demo/Makefile file for an example of how you can integrate generation of a message catalog into the makefile that builds an application.

3.4.1 Using the mkcatdefs Command

The mkcatdefs command preprocesses one or more message source files to change symbolic identifiers to numeric constants. The utility has the following features:

- Sends preprocessed message source to standard output, so you can either pipe the output to the gencat command as described in Example 3–2 or use the > redirection specifier to print the output to a file
- Creates a header file that maps numbers identifying message sets and messages in the new message catalog with the symbolic identifiers referred to in source programs

  You must include this header file in all the program modules that open this catalog and refer to message sets and messages that use symbolic identifiers.

The advantage of symbolic identifiers is that you can specify them in place of numbers when you code calls whose arguments include message sets and message identifiers. Symbolic identifiers improve the readability of your program source code and make the code independent of the order in which message sets and entries occur in the message catalog. Each time that the mkcatdefs utility processes a message text source file, it produces an associated header file to equate set and message symbols with numbers. Updating your program after a message file revision can be as simple as compiling it with the new header file.

Note

The mkcatdefs command includes two options that are not discussed in this chapter.

The -S option enables symbolic name support in output passed to the gencat command. The dspmsg command (used in shell scripts) has a corresponding -S option to enable use of symbolic names to retrieve messages from message catalogs that were built to include this support. (The catgets( ) function in the libc Library is restricted at run time by the XSH specification of the X/Open UNIX standard to use numeric identifiers, not symbols, to retrieve messages from a catalog.)

The -m option enables automatic generation of a default message string and assigns it to a symbolic name. This feature removes the requirement to specify a default message string in dspmsg

command lines or `catgets( )` calls for display when the command or function cannot retrieve a message from a catalog.

See `mkcatdefs`(1) for more information about these options.

The option of defining symbolic identifiers for message sets and catalogs is not included in the XSH specification, so do not assume that the `mkcatdefs` command is available on all operating systems that conform to this specification. However, the source text message file and header files produced by the `mkcatdefs` command should be portable among systems that conform to the specification.

The `mkcatdefs` command maps numbers to symbol identifiers based on the ordinal position of those symbols in the message source input stream currently being processed. When you are processing changes to an existing catalog, make sure the symbols you specify in the source input to the `mkcatdefs` command are correctly mapped to numeric counterparts for those symbols in the existing message catalog.

In general, consider the `mkcatdefs` utility a tool for regenerating an entire message catalog, not just parts of it. Use the following guidelines:

- For message and message set deletions, specify numeric identifiers in place of symbols at strategic points in the message source input. This technique prevents deletions of message sets and individual messages from affecting the ordinal position of subsequent entries.
- Define new sets at the end of the input source stream (at the end of the last source file if a catalog is generated from a sequence of source files).
- Define new messages for an existing message set at the end of that set.
- Specify source entries for the entire catalog; otherwise, `mkcatdefs` will not produce a complete message header file. You need a complete header file for compiling programs that use both current and new symbols to identify messages. In addition, `mkcatdefs` generates a `delset` directive before each `set` directive you specify in the input source. In other words, `mkcatdefs` expects your input to completely replace all messages in the referenced set.
- If the catalog was generated from multiple source files, specify source files in the same order as they were specified to generate the existing catalog; otherwise, you invalidate headers used to compile all program modules that open the catalog. You can avoid recompiling programs that do not refer to new messages as long as you do not invalidate the symbol-number mapping in the message header file with which those programs were compiled.

- Do not specify NL_SETD in a set directive of a message text source file or try to mix default and user-defined message sets in the same message catalog. Doing so can result in errors from the mkcatdefs or gencat utility.
- Keep in mind that the mkcatdefs utility condenses multiple spaces between the message indentifier and the message text to a single space. This modification ensures compatibility with the UNIX standard and the requirements of other UNIX and LINUX platforms.

3.4.2 Using the gencat Command

The gencat command merges one or more message text source files into a message catalog. For example:

```
# gencat en_US/test_program.cat test_program_en_US.msg
```

The gencat command creates the message catalog if the specified catalog path does not identify an existing catalog; otherwise, the command uses the specified message text source file (or files) to modify the catalog. The gencat command accepts message source data from standard input, so you can omit the source file argument when piping input to gencat from another facility, such as the mkcatdefs command.

The X/Open UNIX standard does not specify file name extensions for message source files and catalogs. On Tru64 UNIX systems, the convention is to use the .msg extension for source files and the .cat extension for catalogs. Because the message catalogs produced by the gencat command are binary encoded, they may not be portable between different types of systems. Message text source files preprocessed by the mkcatdefs command should be portable between systems that conform to X/Open UNIX CAE specifications.

See gencat(1) for more details.

3.4.3 Design and Maintenance Considerations for Message Catalogs

Message sets and message entries are identified at run time by numbers that represent ordinal positions within one version of a message catalog. When you add or delete message sets and entries in an existing catalog, you must be careful not to change the ordinal position specifiers that identify messages.

Consider a message whose English language text "Enter street address: " is identified as 3 : 10 (tenth message of the third message set) in the original generation of a message catalog. That message will have a different identifier in the next version of the catalog if the revised source input to the gencat command performs any of the following operations:

- Inserts message sets at the beginning of the input source
- In the third message set, inserts any messages before the "Enter street address: " entry
- In the third message set, deletes messages before the "Enter street address: " entry without specifying a message deletion directive (a message number followed by no other characters on the line)

Consider the value of adding comments to code to explain restrictions on ordinal positioning to potential translators, as demostrated in the following two program segments:

```
$ Note - Do not reorder message descriptors for columns.

S_COM_LIST_ROW         "%5d        %20s     %20s    %4s     %9s\n"


$ The first descriptor must always be displayed at the beginning of error messages.
$ The second descriptor contains the first name.
$ The third descriptor contains the surname.

S_COM_LIST_ERROR      "%1$s: Error badge number for $2$s   %3$s  incorrect\n"
```

When program source refers to messages by numeric identifiers, any changes in ordinal positions of message sets and message entries require changes to program calls that refer to messages. When a program source file refers to messages by symbolic identifiers, the maintenance cost of ordinal position changes is sharply reduced for each module. In other words, you can synchronize any particular program module with the new version of a message catalog by compiling with the new header file generated by the `mkcatdefs` utility.

The ability to compile program source to synchronize with new message catalog versions does not address issues of complex applications where multiple source files refer to the same message catalog. For such applications, a usual goal is to ensure module-specific maintenance updates. In other words, after an application is installed at end-user sites, you should be able to update a specific module and its associated message catalogs without recompiling and reinstalling all modules in the application. You can achieve this goal in a number of ways. The following design options can help you decide on a message system design strategy that works best for applications developed and maintained at your site:

- One message source file and catalog for each program module
  - Advantages

    This is the easiest strategy to implement for the individual programmer as it eliminates problems that arise when programmers share one source. Source control software, such as the Revision Control System (RCS) and the Source Code Control System

(SCCS), help to manage files that multiple programmers maintain. Sometimes, however, programmers work on different application versions in parallel. This additional layer of complexity is not easy to manage. A one-to-one correspondence between message source files and associated program sources makes it easier to determine whose changes are needed in the message file to build the application for a particular release cycle at a specific point in time.

When the message catalog is module specific, you can replace the entire message catalog when a new binary module is installed at end-user sites. Module replacement minimizes risk to the run-time behavior of other modules in the same application.

– Disadvantages

At run time, the application may need to open and close as many message catalogs as there are modules. Opening a message catalog entails some performance overhead and adds to the number of open file descriptors assigned to the user's process and to the systemwide open file table. There is a systemwide and process-specific maximum for the number of files that can be open simultaneously, and these limits vary from one system to another.

On Tru64 UNIX systems, opened message catalogs are mapped into memory (and the file closed) to improve performance of message retrieval. This operation also means that opening multiple message catalogs has little impact on open file limits. This situation, however, may not exist on other platforms to which you might need to port your application.

- One message source file for each program source and a single catalog for each application

  – Advantages

  This technique has the same advantages as one message source file and catalog for each program as described previously. In addition, the single catalog design eliminates any problems associated with numerous open operations if you port your application to systems other than Tru64 UNIX.

  – Disadvantages

  When you generate a message catalog from multiple source files, maintainability problems can occur if you do not carefully control message set directives. The best rule to follow is to define a fixed number of sets for each source file. For example, define one set for errors, one set for informational displays, and one set for miscellaneous strings. If you allow programmers to change the number of message sets for different versions of their message source files, the message set numbers for subsequent program modules are

likely to change from one version of the catalog to another. This means that other modules whose source code was not changed may have to be included in an update release simply for synchronization with a new version of the message catalog.

There are similar maintainability problems if no source files define message sets or if only some of them do. The `mkcatdefs` and `gencat` commands concatenate input source files so that the end-of-file marker exists only at the end of the last input source file. This means that, if no sets are defined in any file, all messages are considered part of the default message set. (In program calls, the `NL_SETD` constant refers to the default message set.) In this case, adding messages to any source file other than the last one changes the numeric identifiers of messages in all source files that follow on the input stream.

Another disadvantage of the multiple source file to single message catalog design arises when the resulting message catalog is extremely large and memory is limited. As mentioned earlier, message catalogs are mapped into memory when opened so that disk I/O for message retrieval does not impede performance. If the users who run your application typically use software and messages that are associated only with a subset of the available modules, module-specific message catalogs can conserve the total amount of memory used when message catalogs are opened for a particular execution cycle.

Finally, if only some message source files define message sets, message sets can cross source file boundaries. Messages defined in source files that occur later on the input stream are considered part of a message set defined by a source file processed earlier. This arrangement can also result in message entry position changes when new messages are added to different source files.

- Combination strategy

  Depending on your application, it might make sense to have one or more message catalogs that are generated from multiple, module-specific source files and some that are generated from a single source file that is maintained by all programmers.

  For example, if many modules in the application generate messages for the same error conditions, message text consistency is a desirable goal. In this case, generate one message catalog with a single message text source file in which error messages are defined. Use this source file to define message sets for errors, warnings, and so forth. Programmers would be instructed to add new messages only to the end of each set and to delete obsolete messages with message deletion directives. Message deletion directives remove messages from the catalog without changing the position numbers for subsequent messages in the same set.

To make the task of maintaining message files easier, consider the following guidelines:

- Add new messages at the end of a message set. This helps to maintain backward compatibility with existing message catalogs.
- Do not remove obsolete messages. This allows older programs to continue to work with newer message catalogs. You can, however, add comments to the message file identifying the message as obsolete.
- Resist the temptation to make cosmetic changes to messages. Because changed messages often require retranslation, you must weigh this cost against the need for change. In general, only change messages that contain incorrect parameters (number or type), incorrect information, and egregious spelling or grammatical errors.
- Correct messages without changing the placement of the message in the file. This avoids any mismatch between old and new programs or catalogs. Also, add a comment to the file explaining the correction.

3.5 Displaying Messages and Locale Data

After a message catalog is created, you can display its contents to make sure that the catalog contains the messages you intended and that both messages and message sets are in the proper order. Your application might also include scripts that, like programs, need to determine locale settings, retrieve locale-dependent data, and display messages in a locale-dependent manner at execution time.

The following list describes the `dspcat`, `dspmsg`, and `printf` commands, which display messages in a message catalog, and the `locale` command, which displays information for the current locale:

- `dspcat` command

  The `dspcat` command can display all messages, all messages in a particular set, or a specific message. The following example displays the fourth message in the second set of the `xpg4demo.cat` catalog:

  ```
  % cd /usr/examples/xpg4demo/en_US
  % dspcat xpg4demo.cat 2 4
  Are these the changes you want to make?%
  ```

  The `dspcat` command also includes a `-g` flag, which reformats the output stream for an entire catalog or message set so that it can be piped to the `gencat` command. This option may be useful if you need to add or replace message sets in one catalog by using message sets in another catalog, perhaps as part of an application update procedure at end-user sites. You can also use the `dspcat -g` command to create a source file from an existing message catalog. You can then translate or customize

the source file for end users before building the translated source into a new catalog with the gencat command.

The following example first displays the message source for the message catalog used by the du command for the en_US.ISO8859-1 locale and then redirects that source to a file that can be edited:

```
% dspcat -g \
/usr/lib/nls/msg/en_US.ISO8859-1/du.cat

$delset 1
$set 1
$quote "

1       "usage: du [-a|-s] [-klrx] [name ...]\n"
2       "du: Cannot find the current directory.\n"
3       "du: %s\n\
The specified pathname exceeded 255 bytes.\n"
4       "du: %s\n\
The generated pathname exceeded 255 bytes.\n"
5       "du: Cannot change directory to ../%s \n"
6       "Out of memory"
% dspcat -g \
/usr/lib/nls/msg/en_US.ISO8859-1/du.cat > \
du.msg
```

- dspmsg command

  The dspmsg command displays a particular message from a catalog and optionally allows you to substitute text strings for all %s or %*n* $s specifiers in the message. For example:

```
% dspmsg xpg4demo.cat -s 1 9 'Cannot open %s for output' xpg4demo.dat
Cannot open xpg4demo.dat for output%
```

- printf command

  The printf command writes a formatted string to standard output. Like the printf( ) function, the command supports conversion specifiers that let you format messages in a way that is locale dependent. You can also use this command in scripts, along with the locale command, to interpret "yes/no" responses in the user's native language. For example:

```
if printf "%s\n" "$response" | grep -Eq "'locale yesexpr'"
then
        <processing for an affirmative response goes here>
else
        <processing for a response other than affirmative goes here>
fi
```

- locale command

  The locale command displays information for the current locale setting or tells you what locales are installed on the system. In the following example, the locale command displays the current settings of all

locale variables, then the keywords and values for a specific variable (LC_MESSAGES), and finally the value for a particular item of locale data:

```
% locale
LANG=en_US.ISO8859-1
LC_COLLATE="en_US.ISO8859-1"
LC_CTYPE="en_US.ISO8859-1"
LC_MONETARY="en_US.ISO8859-1"
LC_NUMERIC="en_US.ISO8859-1"
LC_TIME="en_US.ISO8859-1"
LC_MESSAGES="en_US.ISO8859-1"
LC_ALL=
% locale -ck LC_MESSAGES
LC_MESSAGES
yesexpr="^([yY]|[yY][eE][sS])"
noexpr="^([nN]|[nN][oO])"
yesstr="yes:y:Y"
nostr="no:n:N"
% locale yesexpr
^([yY]|[yY][eE][sS])
```

See dspcat(1), dspmsg(1), printf(1), and locale(1) for more information on the preceding commands.

3.6 Accessing Message Catalogs in Programs

Programs call the following functions to work with a message catalog:

- catopen( ) to open message catalogs (Section 3.6.1)
- catclose( ) to close message catalogs (Section 3.6.2)
- catgets( ) to read program messages (Section 3.6.3)

Message catalogs are usually located through the setting of the NLSPATH environment variable. The following sections discuss this variable and the calls in the preceding list.

3.6.1 Opening Message Catalogs

Programs call the catopen( ) function to open a message catalog. For example:

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

nl_catd        MsgCat;
⋮

setlocale(LC_ALL, "");
```

```
⋮

MsgCat = catopen("new_application.cat", NL_CAT_LOCALE);
```

In this example, the `catopen( )` function returns a message catalog descriptor to the `MsgCat` variable. The variable that contains the descriptor is declared as type `nl_catd`. The `catopen( )` function and the `nl_catd` type are defined in the `/usr/include/nl_types.h` header file, which the program must include. A call to `catopen( )` requires the following arguments:

- The name of the catalog

  The catalog name is customarily specified as *`filename`*`.cat` (or a program variable whose value is *`filename`*`.cat`) without the preceding directory path. At run time, the `catopen( )` function determines the full pathname of the catalog by integrating the name argument into pathname formats defined by the `NLSPATH` environment variable. If you specify any slash (/) characters in the catalog name argument, the `catopen( )` function assumes that the specified catalog name represents a full pathname and does not refer to the value of the `NLSPATH` variable at run time.

- An *`oflag`* argument

  This argument is either the `NL_CAT_LOCALE` constant (defined in `/usr/include/nl_types.h`) or zero (`0`). If you specify the `NL_CAT_LOCALE` constant, the `catopen( )` function searches for a message catalog that supports the locale set for the `LC_MESSAGES` environment variable. If you specify `0`, the `catopen( )` function searches for a message catalog that supports the locale set for the `LANG` environment variable.

  A `0` argument is supported for compatibility with XPG3. The `NL_CAT_LOCALE` argument conforms to The Open Group's current UNIX CAE specifications and is recommended.

  Although the `LC_MESSAGES` setting is usually inherited from the `LANG` setting rather than set explicitly, there are circumstances when programs or users set `LC_MESSAGES` to a different locale than set for `LANG`.

The names and locations of message catalogs are not standard from one system to another. The Open Group's UNIX standard therefore specifies the `NLSPATH` environment variable to define the search paths and pathname format for message catalogs on the system where the program runs. The `catopen( )` function refers to the variable setting at run time to find the catalog being opened by the program. If you do not install your application's message catalogs in customary locations on the user's system, your application's startup procedure will need to prepend an appropriate pathname format to the current search path for `NLSPATH`.

The syntax for setting the NLSPATH environment variable is as follows:

NLSPATH= [[[:]] [*/directory*] [[[/]] | [*substitution-field*] | [*literal*]] ... [[:] *alternate_pathname*] ...]

A leading colon (:) or two adjacent colons (::) indicate the current directory; subsequent colons act solely as separators between different pathnames. Each pathname in the search path is assembled from the following components:

- */directory* to indicate the full directory path to the catalog

  You can also specify *./directory* to indicate a relative path.

- *substitution-field,* which can be one of the following directives:
  - %N

    The value of the first argument to catopen( ), for example, xpg4demo.cat in the following call:

    ```
    catopen("xpg4demo.cat", NL_CAT_LOCALE);
    ```

  - %L

    The locale set for one of the following:

    LC_MESSAGES, if the second argument to catopen( ) is the NL_CAT_LOCALE constant

    LANG, if the second argument to catopen( ) is zero (0)

    This substitution field represents an entire locale name, such as fr_FR.ISO8859-1.

  - %l

    The language component of the locale set for either the LC_MESSAGES or LANG variable (as determined by the same conditions specified for %L)

    Given the locale name fr_FR.ISO8859-1, this substitution field represents the component fr.

  - %t

    The territory component of the locale set for either the LC_MESSAGES or LANG variable (as determined by the same conditions specified for %L)

    Given the locale name fr_FR.ISO8859-1, this substitution field represents the component FR.

  - %c

    The codeset component of the locale set for either the LC_MESSAGES or LANG variable (as determined by the same conditions specified for %L)

    Given the locale name fr_FR.ISO8859-1, this substitution field represents the component ISO8859-1.

  - %%

    A single % character

- *literal* to indicate the following:
  - Directory or file names that cannot be specified using substitution fields
  - Field separators, for example, an underscore (_) or period (.) between the language, territory, and codeset substitution fields or a slash (/) between the %L and %N substitution fields

To clarify how the LC_MESSAGES setting, NLSPATH setting, and the catopen( ) function interact, consider the following set of conditions:

- The locale set for LC_MESSAGES is fr_FR.ISO8859-1. (Unless explicitly set by the user or program, the locale set for LC_MESSAGES is derived from the locale set for LANG.)
- The NLSPATH variable is set to the following value:

```
:%l_%t/%N:/usr/kits/xpg4demo/msg/%l_%t/%N:\
/usr/lib/nls/msg/%L/%N
```

- The program initializes the locale with the following call:

```
.
.
.
setlocale(LC_ALL, "");
.
.
.
```

- The program opens a message catalog with the following call:

```
.
.
.
MsgCat = catopen("xpg4demo.cat", NL_CAT_LOCALE);
.
.
.
```

Given the preceding conditions, the catopen( ) function looks for catalogs at run time in the following pathname order:

1. xpg4demo.cat

2. ./fr_FR/xpg4demo.cat
3. /usr/kits/xpg4demo/msg/fr_FR/xpg4demo.cat
4. /usr/lib/nls/msg/fr_FR.ISO8859-1/xpg4demo.cat

When troubleshooting run-time problems, consider how catopen( ) behaves when certain variables are not set.

If LC_MESSAGES is not set (directly or through the LANG variable), the %L and %l fields contain the value C (the default locale for LC_MESSAGES) and the %t and %c substitution fields are omitted from the search path. In this case, catopen( ) searches for the following catalogs:

1. xpg4demo.cat
2. ./C_/xpg4demo.cat
3. /usr/kits/xpg4demo/msg/C/xpg4demo.cat
4. /usr/lib/nls/msg/C/xpg4demo.cat

If LC_MESSAGES is set but the NLSPATH variable is not set, the catopen( ) function searches for the catalog by using a default search path that is vendor defined. On Tru64 UNIX systems, the default search path is /usr/lib/nls/msg/%L/%N:. For the sample set of conditions under discussion now, this default would result in catopen( ) searching for the following:

1. /usr/lib/nls/msg/fr_FR.ISO8859-1/xpg4demo.cat
2. xpg4demo.cat

Finally, if neither LC_MESSAGES nor NLSPATH is set, catopen( ) searches for the following:

1. /usr/lib/nls/msg/xpg4demo.cat
2. ./xpg4demo.cat

If catopen( ) fails to find a message catalog that matches the locale, the function next checks for an appropriate /usr/share/.msg_conv-*locale-name* file. This file, if it exists, specifies another locale for which a message catalog is available and from which messages can be converted. If this file is found, the available message catalog is opened and the appropriate codeset converter is invoked to convert messages to the codeset of the LC_MESSAGES setting. For example, the .msg_conv-fr_FR.UTF-8 file specifies that, if *catalog_name* exists for French in ISO8859–1 format, that catalog can be opened and its messages converted to UTF-8 format.

The catopen( ) function does not return an error status when a message catalog cannot be opened. To improve program performance, the catalog is

not actually opened until execution of the first `catgets( )` call that refers to the catalog. If you need to detect the open file failure at the point in your program where the `catopen( )` call executes, you must include a call to `catgets( )` immediately following `catopen( )`. You can then design your program to exit on an error returned by the `catgets( )` call. Including an early call to `catgets( )` may be important to do in programs that perform a good deal of work before they retrieve any messages from the message catalog. However, informing the user of this particular error is a problem because you cannot retrieve an error message in the user's native language unless the catalog is opened successfully.

For additional information on the `catopen( )` function, see `catopen`(3).

Note

When running in a process whose effective user ID is root, the `catopen( )` function ignores the `NLSPATH` setting and searches for message catalogs by using the `/usr/lib/nls/msg/%L/%N` path. If a program runs with an effective user ID of root, you must do one of the following:

- Install all message catalogs used by the program in locale directories identified as `/usr/lib/nls/msg/%L`.
- Or, install message catalogs used by the program in another directory and create links in the `/usr/lib/nls/msg/%L` directories to those catalog files.

This restriction does not apply to a program when it is run by a user who is logged in as superuser. The restriction applies only to a program that executes the `setuid(\|)` call to spawn a subprocess whose effective user ID is root.

3.6.2 Closing Message Catalogs

The `catclose( )` function closes a message catalog. This function has one argument, which is the catalog descriptor returned by the `catopen( )` function. For example:

```
(void) catclose(MsgCat);
```

The `exit( )` function also closes open message catalogs when a process terminates.

3.6.3 Reading Program Messages

The `catgets( )` function reads messages into the program. This function takes the following arguments:

- The message catalog descriptor returned by the `catopen( )` call
- The symbolic or numeric identifier of the message set

  Use the `NL_SETD` constant when retrieving messages from message catalogs that do not contain user-defined message sets.
- The symbolic or numeric identifier of the message
- The default message string

  The program uses the default message string when the program cannot retrieve the specified message from a catalog, which usually occurs because the catalog was not found or opened. Because programs commonly use default strings, make sure that the default text is meaningful and always available.

You ordinarily use the `catgets( )` function in conjunction with another routine, either directly or as part of a program-defined macro. The following code from the `xpg4demo` program defines a macro to access a specific message set, then uses the macro as an argument to the `printf( )` routine:

```
.
.
#define GetMsg(id, defmsg)\
                        catgets(MsgCat, MSGInfo, id, defmsg)
.
.

printf(GetMsg(I_COM_DISP_LIST_FMT,
               "%6ld  %20S %-30S %3S %10s\n"),
               emp->badge_num,
               emp->first_name,
               emp->surname,
               emp->cost_center,
               buf);
.
.
```

See `catgets`(3) for more information.

Note

The `gettxt( )` function also reads messages from message catalogs. This function is included in the System V Interface Definition (SVID) but is not recognized by the X/Open UNIX standard. For information about this function, see `gettxt`(3).

4

Handling Wide-Character Data with curses Library Routines

The curses Library provides functions for developing user interfaces on character-cell terminals. This chapter discusses enhancements made to the curses Library to support wide-character format, which accommodates multibyte characters.

The recommended functions for handling multibyte characters in wide-character or complex-character format conform to Version 4.2 of the X/Open Curses CAE specification and supersede those specified by the *System V Multi-National Language Supplement* (MNLS).

This chapter summarizes the curses Library functions and macros that process characters and character strings from the screen or keyboard. Tables in each section note if more than one curses interface is available to perform the same operation and recommend the curses interface that is best suited for writing international software. That is, the tables highlight the curses Library functions and macros that handle wide-character or complex-character format and conform to the X/Open Curses CAE specification. Make sure your application uses the curses interface listed in the Recommended Routine column of the table.

The Section 3 reference pages provide syntax and detailed information for each interface. Use this chapter to determine the interface needed for the operation you want to perform; then use the man command to display the reference page for the chosen interface. For an overview of all the functions in the curses Library, see curses(3).

Note

Some curses routines overwrite existing characters on the curses window. Only the routines that use the wchar_t or cchar_t data type ensure that overwriting does not leave partial characters on the screen. When the display width of an overwritten character is greater than one column, as may be the case for multibyte characters, these routines write extra blank characters to remove partial characters. For example, if the English language character a overwrites the first column of a

2-column Chinese language character, the second column of the Chinese character is overwritten with a blank.

Behavior is undefined when you overwrite multibyte characters with `curses` routines that have not been internationalized.

4.1 Writing a Wide Character to a curses Window

The following sections discuss routines that add or insert individual wide characters on a `curses` window. These routines perform one of the following operations if a character already exists at the target position:

- Overwrite the existing character and then advance the cursor
- Insert the new character before the existing one and do not advance the cursor

4.1.1 Add Wide Character (Overwrite) and Advance Cursor

The routines listed in Table 4–1 add a wide character and its attributes to a window on the screen and advance the cursor. If a character already exists at the target position, the character is overwritten by the one being added.

Your choice of routine depends on whether you need to do the following:

- Add the character to the default or a specified window
- Add the character at the current or specified coordinates
- Refresh the screen

Use the `const cchar_t` data type to pass a wide character with its attributes to these routines.

Table 4–1: curses Routines to Add Wide Characters and Advance the Cursor

Recommended Routine	Used in Place of:	Behavior with Respect to:
`add_wch`	`addch`, `addwch`	Window: default Position: current Screen refresh: no
`wadd_wch`	`waddch`, `waddwch`	Window: specified Position: current Screen refresh: no
`mvadd_wch`	`mvaddch`, `mvaddwch`	Window: default Position: specified Screen refresh: no

Table 4–1: curses Routines to Add Wide Characters and Advance the Cursor (cont.)

Recommended Routine	Used in Place of:	Behavior with Respect to:
mvwadd_wch	mvwaddch, mvwaddwch	Window: specified Position: specified Screen refresh: no
echo_wchar	echowchar	Window: default Position: current Screen refresh: yes
wecho_wchar	wechowchar	Window: specified Position: current Screen refresh: yes

4.1.2 Insert Wide Character (No Overwrite) and Do Not Advance Cursor

The routines listed in Table 4–2 insert a wide character in a window at the current or specified coordinates and do not change the position of the cursor after the write operation. The wide character is inserted before an existing character at the target position, so these routines do not overwrite characters that already exist on the line. Existing characters at and to the right of the target position are moved further to the right and the character in the rightmost position is truncated. Your choice of interface in this category depends on whether you want to do the following:

- Write to the default or a specified window
- Write at the current or specified coordinates

Table 4–2: curses Routines to Insert Wide Characters and Not Advance the Cursor

Recommended Routine	Used in Place of:	Behavior with Respect to:
ins_wch	insch, inswch	Window: default Position: current
wins_wch	winsch, winswch	Window: specified Position: current
mvins_wch	mvinsch, mvinswch	Window: default Position: specified
mvwins_wch	mvwinsch, mvwinswch	Window: specified Position: specified

4.2 Writing a Wide-Character String to a curses Window

The following sections discuss routines that add or insert wide-character strings in curses windows.

4.2.1 Add Wide-Character String (Overwrite) and Do Not Advance Cursor

The routines listed in Table 4–3 add a wide-character string and its character attributes to a window. However, these routines also behave as follows:

- Do not advance the position of the cursor
- Do not check the string for special characters (such as newline, tab, and backspace) that usually affect cursor position
- Truncate the string rather than wrapping it to the next line

Characters in the string that these routines add overwrite characters that already exist at the target position. Your choice of interface in this category depends on whether you need to do the following:

- Write all or some of the characters in the string
- Write the characters to the default or a specified window
- Write the characters at the current or specified coordinates

Table 4–3: curses Routines to Add Wide-Character Strings and Not Advance the Cursor

Recommended Routine	Used in Place of:	Behavior with Respect to:
add_wchstr	addwchstr	Number of characters: all Window: default Position: current
add_wchnstr	addwchnstr	Number of characters: specified Window: default Position: current
wadd_wchstr	waddwchstr	Number of characters: all Window: specified Position: current
wadd_wchnstr	waddwchnstr	Number of characters: specified Window: specified Position: current

Table 4–3: curses Routines to Add Wide-Character Strings and Not Advance the Cursor (cont.)

Recommended Routine	Used in Place of:	Behavior with Respect to:
`mvadd_wchstr`	`mvaddwchstr`	Number of characters: all Window: default Position: specified
`mvadd_wchnstr`	`mvaddwchnstr`	Number of characters: specified Window: default Position: specified
`mvwadd_wchstr`	`mvwaddwchstr`	Number of characters: all Window: specified Position: specified
`mvwadd_wchnstr`	`mvwaddwchnstr`	Number of characters: specified Window: specified Position: specified

4.2.2 Add Wide-Character String (Overwrite) and Advance Cursor

As with the routines discussed in Section 4.2.1, the routines listed in Table 4–4 also add a wide-character string (but without video-character attributes) to a window and overwrite existing characters. However, these routines also do the following:

- Advance the position of the cursor
- Check the string for special characters (such as newline, tab, and backspace) that can also affect the position of characters
- Wrap strings to the next line rather than truncating them

Your choice of interface in this category depends on whether you want to do the following:

- Write all or a specified number of characters in the string
- Write the characters to the default or a specified window
- Write the characters at the current or specified coordinates

Table 4–4: curses Routines to Add Wide-Character Strings and Advance the Cursor

Recommended Routine	Used in Place of:	Behavior with Respect to:
`addwstr`	`addstr`	Number of characters: all Window: default Position: current
`addnwstr`	no replacement	Number of characters: specified Window: default Position: current
`waddwstr`	`waddstr`	Number of characters: all Window: specified Position: current
`waddnwstr`	no replacement	Number of characters: specified Window: specified Position: current
`mvaddwstr`	`mvaddstr`	Number of characters: all Window: default Position: specified
`mvaddnwstr`	no replacement	Number of characters: specified Window: default Position: specified
`mvwaddwstr`	`mvwaddstr`	Number of characters: all Window: specified Position: specified
`mvwaddnwstr`	no replacement	Number of characters: specified Window: specified Position: specified

4.2.3 Insert Wide-Character String (no Overwrite) and Do Not Advance Cursor

The routines listed in Table 4–5 insert a wide-character string before a target position in a `curses` window. These routines do the following:

- Move further to the right any existing characters that are at and to the right of the target position

  Existing characters are not overwritten, but rightmost characters may be truncated at the end of the line

- Check the string for special characters (such as newline, tab, and backspace) that can affect character and cursor placement
- Do not advance the cursor after the write operation

Your choice of interface in this category depends on whether you need to do the following:

- Write all or some of the characters in the string
- Write the characters to the default or a specified window
- Write the characters at the current or specified coordinates

Table 4–5: curses Routines to Insert Wide-Character Strings and Not Advance the Cursor

Recommended Routine	Used in Place of:	Behavior with Respect to:
`ins_wstr`	`inswstr`	Number of characters: all Window: default Position: current
`ins_nwstr`	`insnwstr`	Number of characters: specified Window: default Position: current
`wins_wstr`	`winswstr`	Number of characters: all Window: specified Position: current
`wins_nwstr`	`winsnwstr`	Number of characters: specified Window: specified Position: current
`mvins_wstr`	`mvinswstr`	Number of characters: all Window: default Position: specified
`mvins_nwstr`	`mvinsnwstr`	Number of characters: specified Window: default Position: specified
`mvwins_wstr`	`mvwinswstr`	Number of characters: all Window: specified Position: specified
`mvwins_nwstr`	`mvwinsnwstr`	Number of characters: specified Window: specified Position: specified

4.3 Removing a Wide Character from a curses Window

The routines listed in Table 4–6 delete a wide character at the target position in a `curses` window. Characters that follow the deleted character on the line shift one character to the left. These routines existed in the `curses` Library before multibyte characters were supported and have been redefined for correct handling of wide-character format.

Your choice of interface in this category depends on whether you need to do the following:

- Delete a wide character in the default or a specified window
- Delete a wide character at the current or specified coordinates

Table 4–6: curses Routines to Remove a Wide Character

Recommended Routine	Used in Place of:	Behavior with Respect to:
`delch`	no replacement	Window: default Position: current
`wdelch`	no replacement	Window: specified Position: current
`mvdelch`	no replacement	Window: default Position: specified
`mvwdelch`	no replacement	Window: specified Position: specified

4.4 Reading a Wide Character from a curses Window

The routines listed in Table 4–7 read a wide character and its video attributes from a `curses` window. The data returned to the program is of data type `cchar_t`, so that both the wide character and its attributes are stored.

Your choice of interface in this category depends on whether the character being read is one of the following:

- In the default or a specified window
- At the current or specified coordinates

Table 4–7: curses Routines to Read Wide Characters From a Window

Recommended Routine	Used in Place of:	Behavior with Respect to:
`in_wch`	`inch`, `inwch`	Window: default Position: current
`win_wch`	`winch`, `winwch`	Window: specified Position: current
`mvin_wch`	`mvinch`, `mvinwch`	Window: default Position: specified
`mvwin_wch`	`mvwinch`, `mvwinwch`	Window: specified Position: specified

4.5 Reading a Wide-Character String from a curses Window

Two sets of routines allow you to read a wide-character string from a `curses` window. The set of routines described in Section 4.5.1 retrieve strings that include wide characters with their video attributes. The set of routines described in Section 4.5.2 strip attributes from the characters in the string.

4.5.1 Reading Wide-Character Strings with Attributes

The routines listed in Table 4–8 read a wide-character string and its character attributes from a `curses` window. The string returned by the recommended routines is of the data type `cchar_t`.

Your choice of interface in this category depends on whether you want to do the following:

- Read all or up to a specified number of wide characters in the string
- Read characters from the default or a specified window
- Read characters that are at the current or specified coordinates

Table 4–8: curses Routines to Read Wide-Character Strings With Attributes

Recommended Routine	Used in Place of:	Behavior with Respect to:
`in_wchstr`	`inwchstr`	Number of characters: all Window: default Position: current
`in_wchnstr`	`inwchnstr`	Number of characters: specified Window: default Position: current

Table 4–8: curses Routines to Read Wide-Character Strings With Attributes (cont.)

Recommended Routine	Used in Place of:	Behavior with Respect to:
`win_wchstr`	`winwchstr`	Number of characters: all Window: specified Position: current
`win_wchnstr`	`winwchnstr`	Number of characters: specified Window: specified Position: current
`mvin_wchstr`	`mvinwchstr`	Number of characters: all Window: default Position: specified
`mvin_wchnstr`	`mvinwchnstr`	Number of characters: specified Window: default Position: specified
`mvwin_wchstr`	`mvwinwchstr`	Number of characters: all Window: specified Position: specified
`mvwin_wchnstr`	`mvwinwchnstr`	Number of characters: specified Window: specified Position: specified

4.5.2 Reading Wide-Character Strings Without Attributes

The routines listed in Table 4–9 read a wide-character string from a `curses` window and store a string of data type `wchar_t` in a program variable. Video attributes are stripped from the characters included in the string.

Your choice of interface in this category depends on whether you want to do the following:

- Read all or up to a specified number of characters in the string
- Read characters from the default or a specified window
- Read characters that are at the current or specified coordinates of the window

Table 4–9: curses Routines to Read Wide-Character Strings Without Attributes

Recommended Routine	Used in Place of:	Behavior with Respect to:
`inwstr`	no replacement	Number of characters: all Window: default Position: current
`innwstr`	no replacement	Number of characters: specified Window: default Position: current
`winwstr`	no replacement	Number of characters: all Window: specified Position: current
`winnwstr`	no replacement	Number of characters: specified Window: specified Position: current
`mvinwstr`	no replacement	Number of characters: all Window: default Position: specified
`mvinnwstr`	no replacement	Number of characters: specified Window: default Position: specified
`mvwinwstr`	no replacement	Number of characters: all Window: specified Position: specified
`mvwinnwstr`	no replacement	Number of characters: specified Window: specified Position: specified

4.6 Reading a String of Characters from a Terminal

The routines listed in Table 4–10 get strings of characters from the terminal associated with a `curses` window and store the characters in a program buffer.

Your choice of interface in this category depends on whether you want to do the following:

- Read all or up to a specified number of characters in a string
- Read characters for use in the default or a specified window

- Read characters for use at the current or specified coordinates on the window

Table 4–10: curses Routines to Read Wide-Character Strings From a Terminal

Recommended Routine	Used in Place of:	Behavior with Respect to:
`get_wstr`	`getstr`, `getwstr`	Number of characters: all Window: default Position: current
`getn_wstr`	`getnwstr`	Number of characters: specified Window: default Position: current
`wget_wstr`	`wgetstr`, `wgetwstr`	Number of characters: all Window: specified Position: current
`wgetn_wstr`	`wgetnwstr`	Number of characters: specified Window: specified Position: current
`mvget_wstr`	`mvgetstr`, `mvgetwstr`	Number of characters: all Window: default Position: specified
`mvgetn_wstr`	`mvgetnwstr`	Number of characters: specified Window: default Position: specified
`mvwget_wstr`	`mvwgetstr`, `mvwgetwstr`	Number of characters: all Window: specified Position: specified
`mvwgetn_wstr`	`mvwgetnwstr`	Number of characters: specified Window: specified Position: specified

4.7 Reading or Queuing a Wide Character from the Keyboard

The routines listed in Table 4–11 get a single-byte or multibyte character from the terminal keyboard associated with a `curses` window, convert the character to wide-character format, and return the character to the program. Unless `curses` input mode is set to `noecho`, these routines also echo each character back to the screen.

The unget_wch interface places the wide character at the head of the input queue. In this case, the next call to wget_wch returns the character from the input queue to the program.

Your choice of interface in this category depends on the following uses of the character:

- Use with the default or a specified window
- Use at the current or specified position of the window
- Immediate or delayed use

Table 4–11: curses Routines for Reading Wide Characters From the Keyboard

Recommended Routine	Used in Place of:	Behavior with Respect to:
get_wch	getch, getwch	Window: uses default Position: uses current
wget_wch	wgetch, wgetwch	Window: uses specified Position: uses current
mvget_wch	mvgetch, mvgetwch	Window: uses default Position: uses specified
mvwget_wch	mvwgetch, mvwgetwch	Window: uses specified Position: uses specified
unget_wch	ungetch, ungetwch	Window: not applicable Position: not applicable Input queue: queues character

4.8 Converting Formatted Text in a curses Window

The routines listed in Table 4–12 read wide characters from a curses window and convert them. These functions existed in the curses Library before it was internationalized and have been enhanced to handle wide-character data. In all cases, these functions call wgetstr to read a wide-character string from a window and then interpret and convert characters according to scanf( ) function rules. See scanf(3) for more information.

Your choice of interface in this category depends on whether you do the following:

- Convert a string in the default or a specified window
- Convert a string starting at the current or specified coordinates
- Need to include a list of variables as one of the arguments in the call

Table 4–12: curses Routines to Convert Formatted Text in a Window

Recommended Routine	Used in Place of:	Behavior with Respect to:
`scanw`	no replacement	Window: default Position: current Number of arguments: fixed
`wscanw`	no replacement	Window: specified Position: current Number of arguments: fixed
`mvscanw`	no replacement	Window: default Position: specified Number of arguments: fixed
`mvwscanw`	no replacement	Window: specified Position: specified Number of arguments: fixed
`vw_scanw`	`vwscanw`	Window: specified Position: current Number of arguments: variable

4.9 Printing Formatted Text on a curses Window

The routines listed in Table 4–13 format a string and then print it on a `curses` window. The functions existed in the `curses` Library before it was internationalized and have been redefined to process data in wide-character format. These functions are analogous to `printf( )` (or `vprintf( )`) formatting the string and then `addstr( )` (or `waddstr( )`) writing it. See `printf`(3) for formatting information.

Your choice of interface in this category depends on whether you need to do the following:

- Print on the default or a specified window
- Print at the current or a specified position
- Include a list of variables as one of the call arguments

Table 4–13: curses Routines to Print Formatted Text on a Window

Recommended Routine	Used in Place of:	Behavior with Respect to:
`printw`	no replacement	Window: default Position: current Number of arguments: fixed
`wprintw`	no replacement	Window: specified Position: current Number of arguments: fixed
`mvprintw`	no replacement	Window: default Position: specified Number of arguments: fixed
`mvwprintw`	no replacement	Window: specified Position: specified Number of arguments: fixed
`vw_printw`	`vwprintw`	Window: specified Position: current Number of arguments: variable

5

Creating Internationalized X, Xt, and Motif Applications

This chapter discusses some of the internationalization features that are available for creating a graphical user interface. More specifically, this chapter addresses the following components:

- `libXt`, the Toolkit Intrinsics Library available with Release 6 of the X Window System (Section 5.1)
- `libXm` and `libDXm`, the libraries available with Version 1.2 of OSF/Motif and the features provided as DECwindows Extensions to the OSF/Motif Toolkit (Section 5.2)
- `libX11`, the X Library available with Release 6 of the X Window System (Section 5.3)

This chapter assumes that you are already familiar with the X Window System and the OSF/Motif Toolkit. For more complete information on them, see the following manuals:

- *X Window System Environment*, available as part of the operating system online documentation set
- *Programmer's Supplement for Release 6 of the X Window System*, available in printed form, published by O'Reilly and Associates, Inc.

In addition to these manuals, you can see the reference pages for individual functions for more information.

This chapter does not discuss internationalization features specific to the Common Desktop Environment. See the *Common Desktop Environment: Internationalization Programmer's Guide*, available as part of the operating system online documentation set, for information about using these features. (You can view the operating system documentation at the following website: **`http://www.tru64unix.compaq.com/docs/`**.)

For your convenience in localizing an application, the following list specifies files that enable direct support for a locale in the CDE environment:

Message catalogs:

`/usr/lib/nls/msg/`*`locale_name/filename`*`.cat`...

Reference pages:

```
/usr/share/locale_name/man/mann/refpage.n...
```

Resource files:

```
/usr/lib/X11/locale_name/app-defaults/file_name
/usr/dt/app-defaults/locale_name/file_name
```

UID files:

```
/usr/lib/X11/locale_name/uid/file_name
```

CDE action/datatyping files:

```
/usr/dt/appconfig/types/locale_name/file_name
```

CDE help files:

```
/usr/dt/appconfig/help/locale_name/file_name
```

CDE StyleManager backdrop files:

```
/usr/dt/backdrops/desc.locale_name
```

CDE StyleManager palette files:

```
/usr/dt/palettes/desc.locale_name
```

The operating system provides direct support for some locales and indirect support (through automatic codeset conversion) for other locales. For example, the operating system supports the `ja_JP.UTF-8` locale through codeset conversion, so you will not find paths that contain `ja_JP.UTF-8` as a *`locale-name`* directory or a *`locale-name`* extension after installing operating system product software.

However, if you want to enable direct support for the `ja_JP.UTF-8` locale, you can create the appropriately localized and encoded files to do that and install them in locations as specified for the list items. For information on creating an entirely new locale, see Chapter 6.

5.1 Using Internationalization Features in the Xt Intrinsics Library

The X Toolkit (Xt) Intrinsics Library includes internationalization features related to the initialization process and resource management. The following sections describe these features. For complete information on using routines

from the Xt Intrinsics Library (libXt) in your applications, see the reference pages for individual components.

5.1.1 Establishing a Locale with Xt Functions

An internationalized Xt application must parse resources in a locale-dependent manner. Therefore, an application must establish its locale before initializing the resource database. But it is also true that resources can specify the application's locale. To solve this paradox, Release 5 of the X Toolkit introduced the language procedure, which is registered before initializing X Toolkit and then called during initialization at the appropriate time to set locale.

The XtSetLanguageProc( ) function registers the language procedure for setting the locale. By default, this function first calls the Standard C Library function setlocale( ) to set the locale and then calls the X Library functions XSupportsLocale( ) and XSetLocaleModifiers( ) to initialize the locale.

An application that uses the Xt routines must call XtSetLanguageProc( ), even if the application uses the system default language procedure; otherwise, the locale is not set and other Xt routines do not behave in a locale-dependent manner.

One of the most common ways to set a locale is for applications to make the following call before calling XtAppInitialize( ):

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

After calling XtSetLanguageProc( ), your application can then call one of the following Xt initialization functions:

- XtInitialize( )
- XtAppInitialize( )
- XtOpenDisplay( )

These functions call XtDisplayInitialize( ), which obtains the value of the xnlLanguage resource by parsing the command line and the RESOURCE_MANAGER property. The XtDisplayInitialize( ) function then calls the language procedure registered by the call to XtSetLanguageProc( ), passing it the xnlLanguage value as an argument. After that, XtDisplayInitialize( ) parses resources in the locale returned by the language procedure.

5.1.2 Using Font Set Resources with Xt Functions

The Xt routines support the `XFontSet` structure in place of the `XFontStruct` structure in any internationalized widgets that draw native language text. The following resource attributes exist to support `XFontSet`:

- `XtNFontSet` (the resource name)
- `XtCFontSet` (the resource class)
- `XtRFontSet` (the resource representation type)

The X Toolkit includes a converter that changes a preregistered string, such as `-*-*-*-R-*-*-*-120-75-75-*-*-*-*`, to a list of font sets in the structure (`XFontSet`). The converter should establish a default font set list so that, if the string cannot be converted to a valid font set, there is a fallback to another valid font set.

5.1.3 Filtering Events During Text Input with Xt Library Functions

Starting with Release 5 of the X Toolkit Intrinsics Library, the `XtDispatchEvent( )` function was changed to call `XFilterEvent( )`. This change allows an input method to intercept registered X events before being processed by an application that uses Xt routines.

5.1.4 Including the Codeset Component of Locales with Xt Library Functions

Starting with Release 5 of the X Toolkit Intrinsics Library, an integral locale entity supports the codeset component, in addition to the language and territory components supported by earlier releases.

5.2 Using Internationalization Features of the OSF/Motif and DECwindows Motif Toolkits

The *Common Desktop Environment: Internationalization Programmer's Guide* can provide you with guidelines for developing internationalized Motif applications. This manual is part of the operating system documentation set, which you can view at the following Web site: **`http://www.tru64unix.compaq.com/docs/`**.

The following sections supplement the information in the *Common Desktop Environment: Internationalization Programmer's Guide*.

5.2.1 Setting Language in a Motif Application

Most of the internationalization features in the OSF/Motif Toolkit (`libXm`) and the DECwindows Extensions to the OSF/Motif Toolkit (`libDXm`) are

supported through features that were first introduced in Release 5 of the X Library (libX) and the X Toolkit (libXt).

Motif internationalization features are supported the same way regardless of whether Release 6 or Release 6.3 of the X Library and X Toolkit are installed. For example, to establish the locale of your Motif application, you use the same set of functions and guidelines as described for an Xt application. (See Section 5.1.1.) If your application fails to call XtSetLanguageProc( ) before initializing X Toolkit to register the language procedure, the Motif widgets do not support the internationalization features discussed in subsequent sections. In other words, the widgets revert to behavior expected in releases earlier than X Toolkit Release 5 and OSF/Motif Release 1.2.

The language for an application can be specified in one of the following ways. The list is arranged in precedence order; for example, the language setting in the argv argument takes precedence over the language setting in RESOURCE_MANAGER.

1. The value of the argv argument on the call to XtAppInitialize( ), XtOpenDisplay( ), XtDisplayInitialize( ), or XtOpenApplication( )
2. The setting of the language resource in the RESOURCE_MANAGER property of the root window for the specified display
3. The setting of the xnlLanguage resource in the user's .Xdefaults file
4. The setting of the LANG environment variable

Note the following points:

- After an application opens its first display, Motif routines use the established language setting until the application terminates.
- If the RESOURCE_MANAGER property exists in the root window, Motif routines do not use the .Xdefaults file, even if the language resource is not defined in the RESOURCE_MANAGER property.

5.2.2 Using Compound Strings and the XmText and XmTextField Widgets

The OSF/Motif XmText and XmTextField widgets provide internationalization features based on the X and X Toolkit Libraries. The widgets use the codeset of the current locale to encode text information that users enter and display. To display the data in the correct fonts, the widgets use one of the following search patterns to locate the fonts. They

use the search patterns in order from highest to lowest and stop when the font set is determined.

1. Search the font list for an entry that is a font set and has the font list element tag `XmFONTLIST_DEFAULT_TAG`.
2. Search the font list for an entry that specifies a font set and use the first one found.
3. Use the first font in the font list.

The internationalization features available through the text widgets have changed from earlier OSF/Motif releases on the following two dimensions:

- The segments of a compound string can contain data from multiple character sets. This ability is enabled through the font set construct and support for a locale's codeset rather than a single character set for each language. (Except for Latin codesets, codesets usually support multiple character sets.) To take advantage of this change, your application must ensure the following:
  - That the font list structure defines the appropriate font set as the list element used to display segments of the compound string.
  - That the compound string includes a tag that will match the correct font set rather than a single font.
- For Asian input methods, the `XmText` and `XmTextField` widgets support the On-the-Spot, Off-the-Spot, Over-the-Spot, and Root Window interaction, or preediting, styles for Asian input methods.

  You can specify interaction styles as a priority list for the `XmNpreeditType` resource when creating locale-dependent resource files for your application.

Note

When users choose the Off-the-Spot preediting style, an application window is enlarged to make room for the input status and preedit area (usually at the bottom of the window). Therefore, the Off-the-Spot input style requires that auto-resizing be enabled for any application in which that input style is used.

If you are writing an X or Motif application that will be used in Asian countries where the input status and preedit area are used extensively, do not use toolkit functions to disable auto-resizing for your application.

You can use the following functions to create a compound string for codesets that include multiple character sets:

- `XmStringCreate( )`, which creates a compound string composed of text and a font list element tag
- `XmStringCreateLocalized( )`, which creates a compound string that uses the encoding of the current locale

Note

Right-to-left display of language text, which is appropriate for languages such as Hebrew, is supported through the `DXmCSText` widget. The `XmText` and `XmTextField` widgets support only left-to-right displays.

5.2.3 Internationalization Features of Widget Classes

The following widget classes support native language input and display capabilities through the `XmText` and `XmTextField` widgets (see Section 5.2.2):

- Command
- FileSelectionBox
- Label
- List
- MessageBox
- SelectionBox

5.3 Using Internationalization Features in the X Library

Starting with Release 5 of the `libX11` Library, the X Consortium defined new specifications for developing X clients that handle data for different locales. The new specifications are based on the ANSI **C locale** model, which configures the Standard C Library to process data in different native languages. These specifications provide interfaces for the following:

- Requesting user input in different native languages
- Drawing fonts used for native language text
- Obtaining language-specific resource values
- Interclient communication that supports native language text through codeset conversion

The following sections describe different aspects of writing an internationalized application with the X Library:

- Managing locales (Section 5.3.1)

- Displaying text for different locales (Section 5.3.2)
- Handling interclient communication (Section 5.3.3)
- Handling localized resource databases (Section 5.3.4)
- Handling text input (Section 5.3.5)

To illustrate programming techniques, particularly those pertaining to text input, this chapter uses excerpts from an application named `ximdemo`. The complete source file and an `Imakefile` for this application are provided in the `$I18NPATH/usr/examples/ximdemo` directory. You can read the source file and build and run the application to understand more fully how to apply the programming techniques being discussed.

5.3.1 Managing Locales

An internationalized X client uses the same locale announcement mechanism, the `setlocale` function in the Standard C Library, as other kinds of applications use. The X Library includes two additional functions to determine the locale and to configure locale modifiers: `XSupportsLocale( )` and `XSetLocaleModifiers( )`. Table 5–1 briefly describes these functions. They are more fully described in `XSupportsLocale`(3X11).

Table 5–1: Locale Announcement Functions in the X Library

Function	Description
`XSupportsLocale( )`	Determines if the X Library supports the current locale.
`XSetLocaleModifiers( )`	Specifies a list of X modifiers for the current locale setting.

The `XSetLocaleModifiers( )` list is a null-terminated string where list elements use the following format:

@category =value

The only standard *category* currently defined as a locale modifier is `im`, which identifies the input method. However, several `im` entries can appear on a modifier list when a locale supports more than one input method.

To provide default values on the local host system, the value defined for the `XMODIFIERS` environment variable is appended to the list of any modifiers supplied by the `XSetLocaleModifiers( )` function call. For example, on Tru64 UNIX systems, the default value for the input method is `DEC`. The following command explicitly sets the `XMODIFIERS` variable to this value:

```
% setenv XMODIFIERS @im=DEC
```

In this example, the value `@im=DEC` would be appended to the modifier list specified on the call to the `XSetLocaleModifiers( )` function.

X Library functions operate according to current locale and locale-modifier settings or according to locale and locale modifier settings attached to objects that are supplied to the functions. The following types of objects relate to locale settings:

- `XIM` and `XIC`, which are related to text input
- `XFontSet`, which is related to text drawing and measurement
- `XOM` and `XOC`, which are related to text output

  These objects were introduced in the Version 6 implementation of `XrmDatabase`, which is associated with application resource files.

The locale and locale modifiers of these objects depend on the locale setting when the objects were created. Therefore, you can create objects for various languages and use them simultaneously to process data from different locales. This capability lets you develop multilingual X Window applications. Adhere to the following rules when developing your application:

- Identify the locale that applies to data and handle that data with the appropriate locale-specific object.

  Results are unpredictable when the data's locale does not match the object's locale.

- When passing text to WPI interfaces (such as `printf( )`) in the Standard C Library, ensure that the current locale setting for the process matches the locale of the data being passed.

Example 5–1 illustrates how an X application sets or determines locale.

Example 5–1: Setting Locale in an X Window Application

```
#include <stdio.h>
#include <X11/Xlocale.h>
#include <X11/Xlib.h>
.
.
.
#define DEFAULT_LOCALE          "zh_TW.dechanyu"   1
.
.
.
main(argc, argv)
int     argc;
char    *argv[];
{
.
.
.
     immodifier[0]          = '\0';
     for(i=1; i<argc; i++) {
             if(!strcmp(argv[i], "-Root")) {
                     best_style = XIMPreeditNothing;
                    }
.
.
.
```

Example 5–1: Setting Locale in an X Window Application (cont.)

```
            else if (!strcmp(argv[i], "-locale"))  2
                    locale = argv[++i];
            else if (!strcmp(argv[i], "-immodifier")) {
                    strcpy(immodifier, "@im=");
                    strcat(immodifier, argv[++i]);
                 }
            }
.
.
.
     if(locale == NULL)
            locale = DEFAULT_LOCALE;  3
     if(setlocale(LC_CTYPE, locale) == NULL) {
            fprintf(stderr, "Error : setlocale() !\n");
            exit(0);
            }
     if (!XSupportsLocale()) {
        fprintf(stderr, "X does not support this locale");
        exit(1);
     }
     if (XSetLocaleModifiers(immodifier) == NULL) {
        (void) fprintf(stderr, "%s: Warning : cannot set locale \
modifiers. \n", argv[0]);
     }
.
.
.
```

1 Defines a constant to contain the setting for the default locale.

In this example, the constant's value is explicitly set to `zh_TW.dechanyu`.

2 Determines if a locale was specified on the application command line.

The user can override the default locale by using the `-locale` option on the command line that runs this application.

3 Sets the locale to the value of the `DEFAULT_LOCALE` constant if the locale was not specified on the application command line.

If this constant were set to the null string (`""`) rather than to `zh_TW.dechanyu`, the default locale would be determined by the setting of the `LANG` environment variable for the process in which the application is run.

5.3.2 Displaying Text for Different Locales

Codesets for some locales, particularly those for Asian languages, require more than one X Window font to display all the characters defined. To handle these codesets, the X Library supports the concept of a font set, which allows you to use more than one font to draw and measure text. The font set concept is implemented by the `XFontSet` structure, which replaces the

XFontStruct structure that was supported by X Library releases earlier than Release 5.

A font set is bound to the locale with which it was created. The functions that draw and measure text interpret the text according to the locale of the font set and therefore map characters to their font glyphs correctly.

The implementation of functions that draw and measure text allows you to use fonts with different encodings to display native language text.

5.3.2.1 Creating and Manipulating Font Sets

Table 5–2 summarizes the functions that create and use font sets. For complete information on a function, see its reference page.

Table 5–2: X Library Functions That Create and Manipulate Font Sets

Function	Description
XCreateFontSet( )	Creates a font set for a specified display. This function determines the codesets required for the current locale and loads a set of fonts to support those codesets.
XFreeFontSet( )	Frees a specified font set and any associated components, such as the base font name list, the font name list, the XFontStruct list, and XFontSetExtents.
XFontsOfFontSet( )	Returns a list of XFontStruct structures and font names for the given font set.
XBaseFontNameListOfFontSet( )	Returns the original base font name list supplied by the client when the font set was created.
XLocaleOfFontSet( )	Returns the name of the locale bound to the specified font set.

Example 5–2 demonstrates the functions that create and use font sets.

Example 5–2: Creating and Using Font Sets in an X Window Application

```
.
.
.
#define DEFAULT_FONT_NAME     "-*-SCREEN-*-*-R-Normal--*-*, -*" 1
.
.
.
        char                  *base_font_name = NULL;
.
.
.
        XFontSet              font_set;
.
.
.
```

Example 5–2: Creating and Using Font Sets in an X Window Application (cont.)

```
        char                  **missing_list;
        int                   missing_count;
        char                  *def_string;
.
.
.

        if (base_font_name == NULL)
                base_font_name = DEFAULT_FONT_NAME;  2
        font_set = XCreateFontSet(display, base_font_name, &missing_list,
                                  &missing_count, &def_string);
.
.
.

        /*
         * if there are charsets for which no fonts can be found,
         * print a warning message.
         */
        if (missing_count > 0) {
                fprintf(stderr, "The following charsets are \
missing: \n");
                for (i=0; i<missing_count; i++)
                    fprintf(stderr, "%s \n", missing_list[i]);
                XFreeStringList(missing_list);
        }
.
.
.
```

1 Defines the constant `DEFAULT_FONT_NAME` to contain the value of the default base font name list.

In this example, the default base font name list is set to `-*-SCREEN-*-*-R-Normal--*-*, -*`. For a default base font name list, specify a generic name (using wildcard fields as demonstrated in the example) rather than a fully specified list of fonts. A fully specified font list works only for a particular locale, whereas a generic name can be the default for multiple locales.

2 Determines whether the default base font name list was supplied on the command line.

The user can override the default base font name list by using the `-fs` option on the application command line.

5.3.2.2 Obtaining Metrics for Font Sets

Table 5–3 summarizes the X Library functions that can query font set metrics and measure text.

Table 5–3: X Library Functions That Measure Text

Function	Description
`XExtentsOfFontSet( )`	Returns an `XFontSetExtents` structure, which contains information about the bounding box of the fonts in the specified font sets.
`XmbTextEscapement( )`, `XwcTextEscapement( )`	Calculate the escapement (in pixels) required to draw a given string by using the specified font set.
`XmbTextExtents( )`, `XwcTextExtents( )`	Calculate the overall bounding box of the string's image and a logical bounding box for spacing purposes. These functions also return the value returned by `XmbTextEscapement( )` and `XwcTextEscapement( )`, respectively.
`XmbTextPerCharExtents( )`, `XwcTextPerCharExtents( )`	Return the text dimensions of each character of the specified text according to the fonts loaded for the specified font set.

5.3.2.3 Drawing Text with Font Sets

Table 5–4 summarizes functions provided specifically for drawing text in different native languages. Unlike other X Library functions that draw text, the internationalized functions do the following:

- Work with font sets rather than single fonts
- Handle text drawing according to the locale of the font set

Applications use these functions to avoid handling text encoding directly.

Table 5–4: X Library Functions That Draw Text

Function	Description
`XmbDrawText( )`, `XwcDrawText( )`	Draw text, using multiple font sets, and allow complex spacing and font set shifts between text strings. Use these functions in place of their single-font counterparts, `XDrawText( )` and `XDrawText16( )`.

Table 5–4: X Library Functions That Draw Text (cont.)

Function	Description
XmbDrawString( ), XwcDrawString( )	Using one font set, draw only the specified text with the foreground pixel. Use these functions in place of their single-font counterparts, XDrawString( ) and XDrawString16( ).
XmbDrawImageString( ), XwcDrawImageString( )	Fill a destination rectangle with the background pixel. Then draw the specified image text, using one font set, and paint that text with the foreground pixel. Use these functions in place of their single-font counterparts, XDrawImageString( ) and XDrawImageString16( ).

Example 5–3 illustrates how internationalized functions draw text.

Example 5–3: Drawing Text in an X Window Application

```
GC      Jxgc_on, Jxgc_off;
int     Jxcx, Jxcy;
int     Jxcx_offset=2, Jxcy_offset=2;
int     Jxsfont_w, Jxwfont_w, Jxfont_height;
XRectangle      *Jxfont_rect;
int     Jxw_width, Jxw_height;
#define Jxmax_line      10
int     Jxsize[Jxmax_line];
char    Jxbuff[Jxmax_line][128];
int     Jxline_no;
int     Jxline_height;
.
.
.

static int
JxWriteText(display, client, font_set, len, string)
     Display *display;
     Window     client;
     XFontSet   font_set;
     int        len;
     char       *string;
     {
     int   fy;
     XFillRectangle(display, client, Jxgc_off, Jxcx, Jxcy,
                    Jxsfont_w, Jxfont_height);                    1
     if(len == 1 &&
         (string[0] == LF || string[0] == TAB
          || string[0] == CR)) {
         _JxNextLine();
```

Example 5–3: Drawing Text in an X Window Application (cont.)

```
        XFillRectangle(display, client, Jxgc_off, 0, Jxcy,
                       Jxw_width, Jxfont_height);
        }
    else {
        if(Jxcx >= (Jxw_width - Jxwfont_w)
        || (Jxsize[Jxline_no] + len) >=  256) {
            _JxNextLine();
            XFillRectangle(display, client, Jxgc_off, 0, Jxcy,
                           Jxw_width, Jxfont_height);
        }
        strncpy(&Jxbuff[Jxline_no][Jxsize[Jxline_no]], string,
                len);
        Jxsize[Jxline_no] += len;
        fy   = -Jxfont_rect->y + Jxcy;
        XmbDrawImageString(display, client, font_set,
                           Jxgc_on, Jxcx, fy, string, len); 2
        Jxcx += XmbTextEscapement(font_set, string, len);  3
        if(Jxcx >= Jxw_width) {
            _JxNextLine();
            XFillRectangle(display, client, Jxgc_off, 0, Jxcy, \
                           Jxw_width, Jxfont_height);
            }
        }
    XFillRectangle(display, client, Jxgc_on, Jxcx, Jxcy, \
                   Jxsfont_w, Jxfont_height);
    }
```

1 Displays a block-type cursor by using `XFillRectangle( )`.

2 Displays a native language string by using `XmbDrawImageString( )`.

The string may contain both single-byte and multibyte characters.

3 Calculates the position for drawing the next string with `XmbTextEscapement( )`.

5.3.2.4 Handling Text with the X Output Method

The concept of a font set, as described in the preceding sections, was introduced in Version 5 of the X Library. Version 6 of the X Library implements the more generalized concepts of output methods and output contexts. Output methods and output contexts handle multiple fonts and context dependencies to enable bidirectional text and context-sensitive text display.

To draw locale-dependent text, the application requires data about which fonts are required for that text, how the text can be separated into its

components, and which font is required for each of those components. Version 6 of the X Library provides the following objects to address this requirement:

- X Output Method (XOM)

  XOM is an opaque data structure that the application can use to communicate with an output method.

- X Output Context (XOC)

  XOC is compatible with `XFontSet` in terms of its program interface but is a more generalized object.

Table 5–5 summarizes the X Library functions related to XOM and XOC. For more information on these X Library functions, see the respective reference pages.

Table 5–5: X Library Functions for Output Method and Context

Function	Description
`XOpenOM( )`	Opens an output method to match the specification of the current locale and modifiers. The function returns an XOM object to which the current locale and modifiers are bound.
`XCloseOM( )`	Closes the specified output method.
`XSetOMValues( )`	Sets an output method's attributes.
`XGetOMValues( )`	Gets the properties or features of the specified output method.
`XDisplayOfOM( )`	Returns the display associated with the specified output method.
`XLocaleOfOM( )`	Returns the locale associated with the specified output method.
`XCreateOC( )`	Creates an output context within the specified output method.
`XOMOfOC( )`	Returns the output method associated with the specified output context.
`XSetOCValues( )`	Sets the values of the XOC object.
`XGetOCValues( )`	Gets the values of the XOC object.
`XDestroyOC( )`	Destroys the specified output context.

5.3.2.5 Converting Between Different Font Set Encodings

X fonts may be available in different encodings for the following reasons:

- More than one encoding for a character set may be in common use.

For example, character sets for Japanese (JIS X0208), Chinese (GB 2312), and Korean (KS C 5601) are available in GL or GR encoding.

- More than one character set may be supported in a particular country.
- Different vendors have adopted different font encoding schemes in their products.

Font-encoding divergence from one system to another causes problems for applications that you run on different kinds of systems. Therefore, the implementation of the functions for text drawing and measurement incorporates a mechanism to convert between different font encodings. For conversion to take place, you must design your application so that it can determine the base font name list appropriate for the run-time environment. The application can obtain the base font name list from a resource file or through an option the user specifies on the command line. For example, in the command line to run the `ximdemo` application, the user can include the `-fs` option to specify a base font name list.

The conversion mechanism for font encoding is available only when your application uses the internationalized text drawing functions in the X Library. The conversion mechanism is not available with the primitive text drawing functions, such as `XDrawText( )` and `XDrawString( )`.

5.3.3 Handling Interclient Communication

When designing applications for use with different languages and in different countries, you cannot assume that only Latin-1 or ASCII text strings are used for interclient communication. The X Library therefore contains functions that can handle text strings from any language for interclient communication. Table 5–6 summarizes these functions.

Table 5–6: X Library Functions for Interclient Communication

Function	Description
`XmbSetWMProperties( )`	Provides a single programming interface for setting essential window properties. Your application uses these properties to communicate with other clients, particularly window and session managers. For example, the functions have arguments for window and icon names, and these names can contain multibyte characters in some locales.
`XmbTextListToTextProperty( )`, `XwcTextListToTextProperty( )`	Convert text encoded in the current locale to text properties of type `STRING` or `COMPOUND_TEXT`.

Table 5–6: X Library Functions for Interclient Communication (cont.)

Function	Description
XmbTextPropertyToTextList( ), XwcTextPropertyToTextList( )	Convert text properties of type STRING or COMPOUND_TEXT to a list of multibyte-character or wide-character strings.
XwcFreeStringList( )	Frees the memory allocated by XwcTextPropertyToTextList( ).
XDefaultString( )	Queries the default string that is substituted when a character cannot be converted. When conversion routines encounter a string with a character that cannot be converted, they substitute a locale-dependent default string. The XDefaultString( ) function queries that default string.

Example 5–4 is an example of interclient communication in an X application.

Example 5–4: Communicating with Other Clients in an X Window Application

```
⋮
        if (!strcmp(locale,"zh_TW.dechanyu")) {
                strcpy(title, "XIM F|n/");
        } else if (!strcmp(locale, "zh_CN.dechanzi")) {
                strcpy(title, "XIM J>76");
        } else if (!strncmp(locale, "ja_JP", 5)) {
                strcpy(title, "XIM %G%b");
        } else if (!strcmp(locale, "ko_KR.deckorean")) {
                strcpy(title, "XIM 5%8p");
        } else if (!strcmp(locale, "th_TH.TACTIS")) {
                strcpy(title, "XIM !RCJR8T5");
        } else {
            strcpy(title, "XIM Demo")  [1]
        }
        XmbSetWMProperties(display, window, title, title, NULL, \
                           0, NULL, NULL, NULL);  [2]
⋮
```

[1] Inserts native language text in quoted arguments to the strcmp( ) and strcpy( ) functions.

In this example, the text is for a window title. Text strings are explicitly specified in the function calls for the sake of simplicity. In practice, X or Motif applications extract such text strings from locale-specific resource or user interface language (UIL) files.

2 Passes the text to the XmbSetWMProperties( ) function to parse the title, using the locale, and to set the window manager's property accordingly.

5.3.4 Handling Localized Resource Databases

The locale of an X resource file depends on the locale setting when the file was created. Therefore, when a resource file or string is loaded to create a resource database, the file or string is parsed in the current locale. This situation is similar to the situation described for the binding of font sets with locales in Section 5.3.2.

Table 5–7 summarizes the X Library functions that handle localized resource databases.

Table 5–7: X Library Functions That Handle Localized Resource Databases

Function	Description
XrmLocaleOfDatabase( )	Returns the name of the locale bound to the specified database.
XrmGetFileDatabase( )	Opens the specified file, creates a new resource database, and loads it with the specifications read from the file. The file is parsed in the current locale.
XrmGetStringDatabase( )	Creates a new resource database and stores the resources that are specified in a null-terminated string. The string is parsed in the current locale.
XrmPutLineResource( )	Adds a single resource entry to the specified database. The entry string is parsed in the locale of the database.
XrmPutFileDatabase( )	Stores a copy of the specified database in the specified file. The file is written in the locale of the database.
XResourceManagerString( )	Converts the RESOURCE_MANAGER property encoded in type STRING to the multibyte string encoded in the current locale. This function converts encoding in the same way encoding is converted by the XmbTextPropertyToTextList( ) function.

5.3.5 Handling Text Input with the X Input Method

When developing internationalized X applications, you must be able to request data input in different locales from the same keyboard. The X Library incorporates the following objects to address this problem:

- X Input Method (XIM)

  XIM is an opaque data structure that an application can use to communicate with an input method.

- X Input Context (XIC)

  XIC represents the state of a text entry field in the context of a multithreaded approach to user input.

An application can provide multiple text entry fields for users to enter text data to switch between fields. To obtain data input, the application calls `XmbLookupString( )` or `XwcLookupString( )` with an input context. The strings returned are always encoded in the locale associated with the XIM or XIC objects. The following sections provide more information about using input method objects.

5.3.5.1 Opening and Closing an Input Method

To use an input method, an application must first call `XOpenIM( )`. This function establishes a connection to the input method for the current locale and locale modifiers. The function returns an XIM object to which the current locale and locale modifiers are bound. The binding of the locale and modifiers to the XIM object occurs when the call executes. You cannot change the binding dynamically.

When the input method is no longer required, the application calls `XCloseIM( )` to close the XIM object.

The following functions are also available to obtain information about an XIM object:

- `XDisplayOfIM( )`

  Returns the display associated with the specified XIM object.

- `XLocaleOfIM( )`

  Returns the locale associated with the specified XIM object.

The input method opened by the `XOpenIM( )` function is determined by one of the following (in order of highest to lowest priority):

1. The value for the `im` modifier specified in the call to `XSetLocaleModifiers( )`
2. The input method specified for the `XMODIFIERS` environment variable
3. The default input method, whose name is `DEC`

If `XOpenIM( )` fails to obtain the input method from any of the preceding sources, the default is to support only ISO Latin-1 input. The `XOpenIM( )` call can fail under the following conditions:

- The server for the specified input method is not running

- The `im` modifier is specified incorrectly
- The specified input method does not support the current locale

Example 5–5 is an example of how to open and close an input method.

Example 5–5: Opening and Closing an Input Method in an X Window Application

```
main(argc, argv)
int     argc;
char    *argv[];
        {
        Display                 *display;
.
.
.
        XIM                     im;
.
.
.
        char                    *res_file = NULL;
.
.
.
        XrmDatabase             rdb = NULL;
.
.
.
        preedcb_cd.win = client;
        if(res_file) {
                printf("Set Database : file name = %s\n", res_file);
                rdb = XrmGetFileDatabase(res_file);  1
                }
        if((im = XOpenIM(display, rdb, NULL, NULL)) == NULL) {
                printf("Error : XOpenIM() !\n");   2
                exit(0);
                }
.
.
.
        XCloseIM(im);  3
.
.
.
```

1 Passes the resource database `rdb` to `XOpenIM( )` for looking up resources that are private to an input method.

You can specify resource databases created in the application by the internationalized Xt functions.

2 Checks if the input method has been opened successfully.

3 Closes the input method.

5.3.5.2 Querying Input Method Values

Some input method behavior is vendor-defined. For example, different implementations of an input method may support different combinations of user interaction styles.

To help you develop portable applications, the X Library includes the `XGetIMValues( )` function to determine the attributes of an input method. The `XNQueryInputStyle` attribute specifies the user interaction styles supported by an input method.

Example 5–6 demonstrates how to use the `XGetIMValues( )` function with the `XNQueryInputStyle` attribute to obtain information for an input method.

Example 5–6: Obtaining User Interaction Styles for an Input Method

```
main(argc, argv)
int     argc;
char    *argv[];
        {
        Display                 *display;
.
.
.
        int                     i, n;
.
.
.
        XIMStyles               *im_styles;
        XIMStyle                xim_mode=0;
        XIMStyle                best_style = XIMPreeditCallbacks;
        XIM                     im;
.
.
.
        XIMStyle                app_supported_styles;
.
.
.
        for(i=1; i<argc; i++) {
                if(!strcmp(argv[i], "-Root")) {
                        best_style = XIMPreeditNothing;
                        }
                else if (!strcmp(argv[i], "-Cb")) {
                        best_style = XIMPreeditCallbacks;   1
                        }
.
.
.
        /* set flags for the styles our application can support */
        app_supported_styles = XIMPreeditNone | XIMPreeditNothing |
XIMPreeditCallbacks;   2
        app_supported_styles |= XIMStatusNone | XIMStatusNothing;
        XGetIMValues(im, XNQueryInputStyle, &im_styles, NULL);
        n = 1;   3
        if(im_styles != (XIMStyles *)NULL) {
                for(i=0; i<im_styles->count_styles; i++) {
                        xim_mode = im_styles->supported_styles[i];
                        if((xim_mode & app_supported_styles) ==
```

Example 5–6: Obtaining User Interaction Styles for an Input Method (cont.)

```
xim_mode) { /* if we can handle it */
                                n = 0;
                               if (xim_mode & best_style) /* pick user
selected style */
                                   break;  4
                        }
                }
        }
        if(n) {
                printf("warning : Unsupport InputStyle. or No
IMserver.\n");
                exit (0);
                }
.
.
.
```

1 Determines if the user specified a preferred interaction style on the application command line.

In the `ximdemo` application, users can use the `-Root` and `-Cb` options to specify the interaction styles. These options represent the only two styles supported by this particular application. The `-Root` option specifies the style to be Root Window; this style requires minimal interaction between the client and the input server. The `-Cb` option specifies a style where preediting is handled by callbacks. This style enables On-the-Spot preediting.

2 Defines the `app_supported_styles` bitmask to specify the two interaction styles that the application can support.

3 Calls `XGetIMValues( )` to query interaction styles.

The call returns the interaction styles to the `im_styles` parameter.

4 Selects the interaction style that the input method supports and that the application can handle properly.

The interaction style specified by the user takes precedence; otherwise, the application selects the last interaction style in the returned style list.

The interaction styles, or preediting styles, supported for an input method can vary from one locale to another.

To find out what interaction styles are supported for a particular input method, see the following series of country-specific manuals:

- *Technical Reference for Using Chinese Features*
- *Technical Reference for Using Japanese Features*
- *Technical Reference for Using Korean Features*

- *Technical Reference for Using Thai Features*

These manuals are available from the programming bookshelf of the operating system documentation website (**http://www.tru64unix.compaq.com/docs/**).

5.3.5.3 Creating and Using Contexts for an Input Method

Just as the X server can maintain multiple windows for a display, an application can create multiple contexts for an input method. The X Library contains the `XCreateIC( )` function to create an object for input context (XIC). The XIC object maintains a number of attributes that you can set and obtain through other functions. Among these attributes are the following:

- The interaction style for the input context
- The font set with which preediting and status text is drawn
- The callbacks for handling On-the-Spot preediting

To destroy an XIC object, call the `XDestroyIC( )` function.

Example 5–7 demonstrates how to use the `XCreateIC( )` and `XDestroyIC( )` functions.

Example 5–7: Creating and Destroying an Input Method Context in an X Window Application

```
.
.
.
        Display                 *display;
.
.
.
        Window                  root, window, client;
.
.
.
        XIMStyle                xim_mode=0;
.
.
.
        XIM                     im;
        XIC                     ic;
.
.
.
        XVaNestedList           preedit_attr, status_attr;
        XIMCallback             ximapicb[10];
        char                    immodifier[100];
        preedcb_data            preedcb_cd;
.
.
.
        window = XCreateSimpleWindow(display, root, 0, 0,
                                W_WIDTH, W_HEIGHT, 2, bpixel, fpixel);
.
.
.
        client = JxCreateTextWindow(display, window, 0, 0,
```

Example 5–7: Creating and Destroying an Input Method Context in an X Window Application (cont.)

```
                                W_WIDTH-2, W_HEIGHT-2, 1, bpixel, fpixel,
                                font_set, &font_height);
.
.
.
        if (xim_mode & XIMPreeditCallbacks) {
                ximapicb[0].client_data = (XPointer)NULL;
                ximapicb[0].callback = (XIMProc)api_preedit_start_cb;
                ximapicb[1].client_data = (XPointer)(&preedcb_cd);
                ximapicb[1].callback = (XIMProc)api_preedit_done_cb;
                ximapicb[2].client_data = (XPointer)(&preedcb_cd);
                ximapicb[2].callback = (XIMProc)api_preedit_draw_cb;
                ximapicb[3].client_data = (XPointer)NULL;
                ximapicb[3].callback = (XIMProc)api_preedit_caret_cb;
                nestlist = XVaCreateNestedList(10,
                                XNPreeditStartCallback, &ximapicb[0],
                                XNPreeditDoneCallback, &ximapicb[1],
                                XNPreeditDrawCallback, &ximapicb[2],
                                XNPreeditCaretCallback, &ximapicb[3],
                                NULL);   [1]
        }
        if (xim_mode & XIMPreeditCallbacks) {   [2]
                        ic = XCreateIC(im,
                        XNInputStyle, xim_mode,
                        XNClientWindow, window,
                        XNFocusWindow, client,
                        XNPreeditAttributes, nestlist,
                        NULL);   [3]
        } else {        /* preedit nothing */
                        ic = XCreateIC(im,
                                XNInputStyle, xim_mode,
                                XNClientWindow, window,
                                XNFocusWindow, client,
                                NULL );  [4]
        }
        if(ic == NULL) {    [5]
                printf("Error : XCreateIC() !\n");
                XCloseIM(im);
                exit(0);
                }
.
.
.
exit:
        XDestroyIC(ic);   [6]
```

[1] Calls the `XVaCreateNestedList( )` function to create a nested argument list for preediting and status attributes.

The `XNPreeditAttributes` and `XNStatusAttributes` attributes contain a list of subordinate attributes. Your application must create a nested list to contain the subordinate attributes before setting or querying them.

[2] Specifies XIC attributes.

Your application must always specify some XIC attributes when creating an XIC object. The `XNInputStyle` attribute is mandatory; requirements for other attributes depend on the interaction style.

3 Registers callbacks for On-the-Spot interaction style.

When the interaction style is On-the-Spot, your application must register all callbacks when creating the XIC object.

Your application does not have to set the `XNClientWindow` attribute when creating the XIC, but it must set this attribute before using the XIC. If the XIC is used before `XNClientWindow` is set, results are unpredictable.

4 Sets the interaction style, client window, and focus window attributes for the Root Window style.

These are the only attributes your application needs to set at XIC creation time when the interaction style is Root Window.

5 Specifies actions when XIC creation fails.

The call to `XCreateIC( )` fails (that is, returns `NULL`) under the following conditions:

- A required attribute is not set.
- A read-only attribute (for example, `XNFilterEvents`) is set.
- An attribute name is not recognized.

6 Closes the XIC.

Table 5–8 summarizes the functions available for managing an XIC object.

Table 5–8: X Library Functions That Manage Input Context (XIC)

Function	Description
`XSetICFocus( )`	Enables keyboard events to be directed to the input method. You must call this function when the focus window of an XIC receives input focus; otherwise, keyboard events are not directed to the input method.
`XUnsetICFocus( )`	Prevents keyboard events from being directed to the input method. Call this function when the focus window of an XIC loses focus.
`XmbResetIC( )`, `XwcResetIC( )`	Reset the XIC to its initial state. Any input pending on that XIC is deleted. These functions return either the current preedit string or `NULL`, depending on the implementation of the input server.

Table 5–8: X Library Functions That Manage Input Context (XIC) (cont.)

Function	Description
XIMOfIC( )	Returns the XIM associated with the specified XIC.
XSetICValues( )	Sets attributes to a specified XIC.
XGetICValues( )	Queries attributes from a specified XIC.

5.3.5.4 Providing Preediting Callbacks for the On-the-Spot Input Style

If your application supports the On-the-Spot interaction style, you have to provide a set of preediting callbacks. A number of callbacks are associated with XIC. Example 5–8 demonstrates these callbacks.

Example 5–8: Using Preediting Callbacks in an X Window Application

```
.
.
int     Jxsize[Jxmax_line];
char    Jxbuff[Jxmax_line][128];
int     Jxline_no;
int     Jxline_height;
int     sav_cx, sav_cy;
int     sav_w_width, w_height;
int     sav_size[Jxmax_line];
int     sav_line_no;
char    preedit_buffer[12];
void
save_value()
{
    int i;
    sav_cx = Jxcx;
    sav_cy = Jxcy;
    sav_line_no = Jxline_no;
    for (i=0; i< Jxmax_line; i++)
        sav_size[i] = Jxsize[i];
}
void
restore_value()
{
    int i;
    Jxcx = sav_cx;
    Jxcy = sav_cy;
    Jxline_no = sav_line_no;
    for (i=0; i< Jxmax_line; i++)
        Jxsize[i] = sav_size[i];
}
int
api_preedit_start_cb(ic, clientdata, calldata)
XIC ic;
XPointer clientdata;
XPointer calldata;
{
    int len;
    len = 12;
/* save up the values */
    save_value();  [1]
    return(len);   [2]
}
```

Example 5–8: Using Preediting Callbacks in an X Window Application (cont.)

```
void
api_preedit_done_cb(ic, clientdata, calldata)
XIC ic;
XPointer clientdata;
XPointer calldata;
{
    preedcb_data *cd = (preedcb_data *)clientdata;
/* restore up the values */
    restore_value();   3
/* convenient handling */
    JxRedisplayText(cd->dpy, cd->win, cd->fset);
    return;
}
void
api_preedit_draw_cb( ic, clientdata, calldata)
XIC ic;
XPointer clientdata;
XIMPreeditDrawCallbackStruct *calldata;
{
    preedcb_data *cd = (preedcb_data *)clientdata;
    int count;
    char *reset_str;
    if (calldata->text) {
        if (calldata->text->encoding_is_wchar)   4
        {
        } else {
          count =  strlen(calldata->text->string.multi_byte);
          if (count > 12) {
/* preedit string > max preedit buffer */
             reset_str = XmbResetIC(ic);   5
             XFillRectangle(cd->dpy, cd->win, Jxgc_off, Jxcx, Jxcy,
Jxw_width*13, Jxfont_height);       /* clear the preedit area */
             restore_value();
             if (reset_str)
                XFree(reset_str);
             return;
          }
          if (!calldata->chg_length) {  /* insert character */
             if (!calldata->chg_first)  { /* insert in first character
in preedit buffer */
strncpy(&preedit_buffer[0],calldata->text->string.multi_byte, count);
                restore_value();
             } else {
                /* Not Yet Implemented */
             }
          } else {                      /* replace character */
             if (!calldata->chg_first)  { /* replace from first
character in pre-edit buffer */
strncpy(&preedit_buffer[0],calldata->text->string.multi_byte, count);
                restore_value();
             } else {
                /* Not Yet Implemented */
             }
          }
          XFillRectangle(cd->dpy, cd->win, Jxgc_off, Jxcx, Jxcy,
Jxw_width*13, Jxfont_height); /* clear the preedit area */
          JxWriteText(cd->dpy, cd->win, cd->fset, count, preedit_buffer);
        }
    } else {    /* should delete preedit buffer */
```

Example 5–8: Using Preediting Callbacks in an X Window Application (cont.)

```
        /* Not yet implemented */
    }
    return;
}
void
api_preedit_caret_cb(ic, clientdata, calldata)
XIC ic;
XPointer clientdata;
XIMPreeditCaretCallbackStruct *calldata;
{
    /* Not yet implemented */
    return;
}
.
.
.
```

1 Saves the current drawing position.

As part of the operation of drawing preediting strings, this application saves the current drawing position as the value of the `PreeditStartCallback` attribute. After the preediting is complete, the application erases the preediting string and restores the original drawing position.

2 Returns the length of the preediting string.

The value of 12 bytes is an arbitrary number to limit the length of the string. The value should match the size of the preediting buffer. This application declares the preediting buffer (`preedit_buffer`) to be a 12-byte character array.

3 Restores the drawing position and redraws the text buffer.

4 Handles wide-character encoding.

This example assumes that the preediting string is in multibyte encoding. However, your application should handle both multibyte and wide-character encoding. Wide-character encoding is preferable because information, such as character position, is returned in the `XIMPreeditDrawCallbackStruct` structure as the number of characters rather than the number of bytes.

5 Clears the preediting string when its size exceeds 12 bytes.

The size of the string is obtained from the `PreeditDrawCallback` attribute. Without processing the string returned on the call to `XmbResetIC( )`, the application frees the string with a call to `Xfree( )`.

5.3.5.5 Filtering Events for an Input Method

An input method has to receive events before the events are processed by the application. The application has to pass to the input method not only KeyPress/KeyRelease events but other events as well. The X Library contains the `XfilterEvent( )` function to pass events to an input method. Use this function, along with related functions, as follows:

1. Obtain a mask for the events to be passed to the input method by calling the `XGetICValues( )` function with the `XNFilterEvents` argument.
2. Register the event types with the `XSelectInput( )` function.
3. In the main loop of the program (usually right after the call to `XNextEvent( )`) call `XFilterEvent( )` to pass the event to the input method.

   A return status of `True` indicates that the input method has filtered the event and it needs no further processing by the application.

Example 5–9 illustrates the preceding process.

Example 5–9: Filtering Events for an Input Method in an X Window Application

```
.
.       long                    im_event_mask;
.
.

       XGetICValues(ic, XNFilterEvents, &im_event_mask, NULL);
       mask = StructureNotifyMask | FocusChangeMask | ExposureMask;
       XSelectInput(display, window, mask);
       mask = ExposureMask | KeyPressMask | FocusChangeMask |
              im_event_mask;
       XSelectInput(display, client, mask);
.
.

        for(;;) {
                XNextEvent(display, &event);
                if(XFilterEvent(&event, NULL) == True)
                        continue;  [1]
                switch(event.type ) {
                /* dispatch event */
.
.

                }
       }
.
.
```

[1] Filters the event.

Because the XtDispatchEvent( ) function calls XFilterEvent( ), you can replace the for loop as demonstrated in this example with a call to XtAppMainLoop( ).

5.3.5.6 Obtaining Composed Strings from the Keyboard

You use the XmbLookupString( ) or XwcLookupString( ) function in your X application to obtain native language characters and key symbols. Your application has to take into account the complexity of some input methods, which require several keystrokes to compose a single character. Therefore, expect that a composed character or string may not be returned on every call to one of these functions.

Example 5–10 demonstrates how to get keyboard input in an X application.

Example 5–10: Obtaining Keyboard Input in an X Window Application

```
.
.
.
    XEvent                  event;
.
.
.

    int                     len = 128;
    char                    string[128];
    KeySym                  keysym;
    int                     count;
.
.
.

    for(;;) {
            XNextEvent(display, &event);
            if(XFilterEvent(&event, NULL) == True)
                    continue;
            switch(event.type ) {
            case FocusIn :    1
                    if(event.xany.window == window)
                            XSetInputFocus(display, client,
                                RevertToParent, CurrentTime);
                    else if(event.xany.window == client) {
                            XSetICFocus(ic);
                            }
                    break;              case FocusOut :    1
                    if(event.xany.window == client) {
                            XUnsetICFocus(ic);
                            }
                    break;
            case Expose :
                    if(event.xany.window == client)
                            JxRedisplayText(display, client,
                                            font_set);
                    break;
            case KeyPress :    2
                    count = XmbLookupString(ic, (XKeyPressedEvent
                    *)&event, string, len, &keysym, NULL);
                    if( count == 1 && string[0] == (0x1F&'c')) {
                            /* exit */
                            goto exit;
                            }
```

Example 5–10: Obtaining Keyboard Input in an X Window Application (cont.)

```
        if( count > 0 ) {    3
                JxWriteText(display, client,
                            font_set, count, string);
        }
        break;
case MappingNotify :
        XRefreshKeyboardMapping( (XMappingEvent *)&event);
        break;
case DestroyNotify :
        printf("Error : DestroyEvent !\n");
        break;
        }
}
```

1 Handles FocusIn and FocusOut events.

In this example, one XIC is associated with a focus window. Some input servers require focus change information to update the status area. Therefore, each FocusIn event calls `XSetICFocus( )` and each FocusOut event calls `XUnsetICFocus( )`.

Your application can also use one XIC for several focus windows. In this case, you do not need to call `XSetICFocus( )` for every focus change event, but you do have to set the `XNFocusWindow` attribute of the XIC.

2 Handles KeyPress events.

Make sure that your application passes only KeyPress events to `XmbLookupString( )` or `XwcLookupString( )`. Results are undefined if you pass KeyRelease events to these functions.

For simplicity in this example, the status field in the call to `XmbLookupString( )` is `NULL`. Your own application should check for the status return and respond appropriately. For example, if the status return is `XBufferOverflow`, your application might try to allocate more memory for the buffer.

3 Processes the string when one is returned.

`XmbLookupString( )` returns the size of the composed string (in bytes).

5.3.5.7 Handling Failure of the Input Method Server

The `XNDestroyCallback` resources for an input method and an input method context were introduced in X11R6. These resources, which are triggered by failure of the input method server, close the XIM and XIC objects for a client application. If a client application continues to run without detecting server failure and then closes the XIC and XIM objects, results are unpredictable.

Example 5–11 illustrates how to register the XNDestroyCallback resource for the XIM object and how to close the XIM in the event of server failure.

Example 5–11: Handling Failure of the Input Method Server

```
static void     _imDestroyCallback(); [1]
.
.
.
        Bool                    IMS_Connected = False;
        XIMCallback             cb; [2]
.
.
.
        if((im = XOpenIM(display, rdb, NULL, NULL)) == NULL) {
                printf("Error : XOpenIM() !\n");
                exit(0);
                }
        else {
            IMS_Connected = True;
            cb.client_data = (XPointer) &IMS_Connected;
            cb.callback = (XIMProc) _imDestroyCallback;
            XSetIMValues(im, XNDestroyCallback, &cb, NULL); [3]
        }
.
.
.
                case KeyPress :
                        if (IMS_Connected)  count = XmbLookupString(ic,
(XKeyPressedEvent *)&event, string, len, &keysym, NULL);
                        else   count =
XLookupString((XKeyPressedEvent *)&event, string, len, &keysym, NULL); [4]
.
.
.
static void
_imDestroyCallback(im, client_data, call_data)
    XIM im;
    XPointer client_data;
    XPointer call_data;
{
    Bool *Connected = (Bool *)client_data;
    *Connected =3D False; [5]
}
```

[1] Declares the function that closes the XIM if the input method server (IMS) fails for any reason.

[2] Declares the IMS_Connected variable to specify whether the input method server is still connected and the cb structure to contain client information needed for resource registration.

[3] If the call to open the XIM fails, prints an error message and exits.

Otherwise, sets the IMS_Connected variable to True, fills the cb structure with appropriate client data, and calls the XSetIMValues( ) function to register the XNDestroyCallback resource for the XIM.

4 If the input method server is running, uses the `XmbLookupString( )` function to process user input.

   Otherwise, uses the `XLookupString( )` function.

5 Specifies the prototype for the function that closes the XIM if the input method server fails.

The `ximdemo` program is very simple and uses only one input method context. In this case, there is no need to explicitly close the XIC when the input method server fails. The following example describes the prototype for a callback function that would close an XIC:

```
static void icDestroyCallback(ic, client_data, call_data)
XIC ic;
XPointer client_data;
XPointer call_data;
```

5.3.6 Using Xt and X Library Features: A Summary

The following list of steps for processing native language input summarizes the information presented in preceding sections on the X Library. For your convenience, the step description also notes when programming with X Toolkit Intrinsics Library (Xt) functions differs from programming with X Library functions. See Section 5.1 for discussion of internationalization features of the X Toolkit Intrinsics Library.

1. Call `setlocale( )` to bind to the current locale.

   You can accomplish the same result by registering an initialization callback function with `XtSetLanguageProc( )`.

2. Call `XSupportsLocale( )` to verify that X supports the current locale.
3. Either call `XSetLocaleModifiers( )` or set the `XMODIFIERS` environment variable to define the input method being used.
4. Call `XOpenIM( )` to connect to the selected input method.

   If you are writing a widget, you can skip this step and assume that a valid XIM will be passed to the widget as a resource.

5. Call `XGetIMValues( )` to query the interaction styles supported by the input method.

   When writing a widget, do this step in the initialization method.

6. Create a window to associate with an XIC.

   When using Xt functions, create a widget.

7. Call `XCreateFontSet( )` to create a font set for this window. In X11R6, you can use `XOpenOM( )` instead.

   If you are using Xt functions and have created a widget, use the value set for `XtDefaultFontSet`.

8. Choose an interaction style from the supported values obtained by the application and pass this value as an argument to `XCreateIC( )`.

   If you are using `XIMPreeditCallbacks`, you must write the callback routines and register them on the call to `XCreateIC( )`.

9. Call `XGetICValues( )` to query the `XNFilterEvents` attribute and register the event that the input method needs from the focus window.

10. Call `XFilterEvent( )` in the main event loop before dispatching an event.

    If the call returns `True`, you can discard the event.

    If programming with routines from the X Intrinsics (Xt) Library, use `XtDispatchEvent( )`.

11. In the main event loop, set and unset input focus when the focus window receives FocusIn and FocusOut events.

    If programming with routines from the X Intrinsics (Xt) Library, use an event handler or a translation or action table to track focus events.

12. For unfiltered KeyPress events, call `XmbLookupString( )` or `XwcLookupString( )` to obtain key symbols and the composed string.

    You can draw the string with the internationalized functions for text drawing.

6

Creating Locales

This chapter explains how to develop a locale. A locale is the set of data that supports a particular combination of native language, cultural data, and codeset on the operating system. You use the `localedef` command to create locales from the following files:

- `charmap`, a character map source file (Section 6.1)

  See `charmap`(4) for an explanation of the format and rules for this file. This chapter includes a `charmap` example that conforms to binary character encodings specified for the ISO Latin-1 codeset, which defines all characters as single 8-bit bytes. The chapter also includes an example of part of a `charmap` file for the SJIS codeset, which defines both single-byte and multibyte characters.

- A locale source file (Section 6.2)

  See `locale`(4) for an explanation of the rules and format for this file. This chapter includes an example of the development of a locale named `fr_FR.ISO8859-1@example`, which supports the language and customs of France.

- A methods file with associated shareable library (Section 6.3)

  A methods file and shareable library are required when the `charmap` file defines multibyte characters; otherwise, they are optional. The methods file contains an entry for each function used by the locale and defined in the associated shared library. The message file entry includes the library name and path. Method file entries also specify the shared library containing redefinitions of the C Library interfaces that convert data to and from internal process (wide-character) encoding.

For a list of files that must be changed in order for desktop applications to use a new locale, see Chapter 5.

6.1 Creating a Character Map Source File for a Locale

A `charmap` file defines symbols for character binary encodings. The `localedef` command uses this file to map character symbols in a locale source file to the character encodings. Example 6–1 is a fragment of the `ISO8859-1.cmap` source file that is used in the `fr_FR.ISO8859-1@example` locale being developed in this chapter. Section D.1 contains the `ISO8859-1.cmap` file in its entirety.

Example 6–1: The charmap File for a Sample Locale

```
#     1
#     Charmap for ISO 8859-1 codeset    1
#     1

<code_set_name>                "ISO8859-1"    2
<mb_cur_max>                   1      2
<mb_cur_min>                   1      2
<escape_char>                  \      2
<comment_char>                 #      2

CHARMAP      3

#  Portable characters and other standard    1
#  control characters                        1

<NUL>                          \x00     4
<SOH>                          \x01
<STX>                          \x02
<ETX>                          \x03
<EOT>                          \x04
<ENQ>                          \x05
<ACK>                          \x06
<BEL>                          \x07
<alert>                        \x07
<backspace>                    \x08
<tab>                          \x09
<newline>                      \x0a
<vertical-tab>                 \x0b
<form-feed>                    \x0c
<carriage-return>              \x0d
<SO>                           \x0e
⋮

<zero>                         \x30     4
<one>                          \x31
<two>                          \x32
<three>                        \x33
<A>                            \x41
<B>                            \x42
<C>                            \x43
<D>                            \x44
⋮

<underscore>                   \x5f     4
<low-line>                     \x5f
<grave-accent>                 \x60
<a>                            \x61
```

Example 6–1: The charmap File for a Sample Locale (cont.)

```
<b>                             \x62
<c>                             \x63
<d>                             \x64
⋮


# Extended control characters      1
# (names taken from ISO 6429)      1

<PAD>                           \x80    4
<HOP>                           \x81
<BPH>                           \x82
<NBH>                           \x83
<IND>                           \x84
⋮


# Other graphic characters     1

<nobreakspace>                  \xa0    4
<inverted-exclamation-mark>     \xa1
⋮


END CHARMAP     5
```

1 Comment line

By default, the comment character is the number sign (#). You can override this default with a <comment_char> definition (see Example 6–1).

2 Keyword declarations

This example provides entries for all valid declarations and specifies default values for all but <code_set_name>. Usually, you specify a declaration only when you want to override its default value. In this example, the declarations for <escape_char> and <comment_char> specify the default values for the escape character and comment character, respectively. The value for <mb_cur_max>, the maximum length (in bytes) of a character, is 1 for this particular charmap file. The value for <mb_cur_min>, the minimum length (in bytes) of a character, must be 1 in charmap files for all locales. (All locales include characters in the Portable Character Set, which defines single-byte characters.)

The <code_set_name> value is the value returned on the nl_langinfo(CODESET) call made by applications that bind to the locale at run time.

3 Header marking start of character maps

4 Symbol-to-coding maps for characters

Each character map consists of a symbolic name and encoding. The name and encoding are separated by one or more spaces.

A symbolic name begins with the left angle bracket (<) and ends with the right angle bracket (>). The characters between the angle brackets can be any characters from the Portable Character Set, except for control and space characters. If the name includes more than one right angle bracket (>), all but the last one must be preceded by the value of <escape_character>. A symbolic name cannot exceed 128 bytes in length.

An encoding can be one or more decimal, octal, or hexadecimal constants. (Multiple constants apply to multibyte encodings.) The constants have the following formats:

- Decimal

  \d*nnn* or \d*nn*, where *n* is a decimal digit

- Hexadecimal

  \x*nn*, where *n* is a hexadecimal digit

- Octal

  nnn or *nn*, where *n* is an octal digit

You can define multiple character map entries (each with a different symbolic name) for the same encoding value. This example does not define multiple symbolic names for the same encoding value.

5 Trailer marking end of character maps

The source files for codesets with multibyte characters have more complex character maps. Example 6–2 is a subset of character map entries from a source file for the Japanese SJIS codeset. This source file specifies entries from several character sets that must be supported within the same codeset.

Example 6–2: Fragment from a charmap File for a Multibyte Codeset

```
# SJIS charmap
#
<code_set_name> "SJIS"   1
<mb_cur_min>    1    2
<mb_cur_max>    2    3
CHARMAP
#
```

Example 6–2: Fragment from a charmap File for a Multibyte Codeset (cont.)

```
# CS0: ASCII
#
⋮

<commercial-at>         \x40    4
<A>                     \x41    4
<B>                     \x42    4
⋮

#
# CS1:  JIS X0208-1983 for ShiftJIS.
#
<zenkaku-space>         \x81\x40    5
<j0101>...<j0163>       \x81\x40    5
<j0164>...<j0194>       \x81\x80    5
⋮

#
# UDC Area in JIS X0208 plane
#
<u8501>...<u8563>       \xeb\x40    6
<u8564>...<u8594>       \xeb\x80    6
<u8601>...<u8663>       \xeb\x9f    6
⋮

#
# CS2:  JIS X0201 (so-called Hankaku-Kana)
#
<kana-fullstop>         \xa1    7
⋮

<kana-conjunctive>      \xa5    7
<kana-WO>               \xa6    7
<kana-a>                \xa7    7
⋮

END CHARMAP
```

1 Codeset name

2 Minimum number of bytes for each character

This value must be 1.

3 Maximum number of bytes for each character

In SJIS, the largest multibyte character is 2 bytes in length.

4 Symbols and encodings for ASCII characters

5 Symbols and encodings for SJIS characters

Note how character symbols are specified as a range and how two hexadecimal values determine the encoding for a 2-byte character.

When symbols are specified as a range of symbol values, the specified character encoding applies to the first symbol in the range. The `localedef` command automatically increments both the symbol value and the encoding value to create symbols and encodings for all characters in the range.

6 Maps for UDCs within the SJIS codeset

These maps establish ranges of encodings for which users can later define characters.

7 Maps for the single-byte characters of the Hankaku-Kana character set

See `charmap`(4) for a complete list of rules that apply to character map source files.

Note

The symbolic names for characters in character map source files are in the process of becoming standardized. A future revision of the X/Open UNIX standard will likely specify both long and short symbolic names for characters.

The symbolic names for characters in examples are not necessarily the names being proposed for adoption by any standards group.

6.2 Creating Locale Definition Source Files

A locale definition source file defines data that is specific to a particular language and territory. The source file is organized into sections, one for each category of locale data being defined. The locale categories include the following:

- `LC_CTYPE` defines character classes and attributes (Section 6.2.1)
- `LC_COLLATE` defines how characters and strings are collated (Section 6.2.2)
- `LC_MESSAGES` defines the strings used for affirmative and negative responses (Section 6.2.3)
- `LC_MONETARY` defines the rules and symbols for monetary values (Section 6.2.4)

- LC_NUMERIC defines the rules and symbols for numeric data (Section 6.2.5)
- LC_TIME defines date and time (Section 6.2.6)
- LC_ALL references all the categories

Example 6–3 illustrates the structure of a locale definition source file in pseudocode.

Example 6–3: Structure of Locale Source Definition File

```
# comment-line    1

comment_char      <char_symbol1>   2
escape_char       <char_symbol2>   3

CATEGORY_NAME    4

category_definition-statement   5
category_definition-statement   5
⋮

END CATEGORY_NAME    6
⋮

7
```

1 Comment line

The number sign (#) is the default comment character. You can specify comments as entire lines by entering the comment character in the first column of the line. You cannot specify comments on the same lines as definition statements in locale source files. In this respect, locale source files differ from character map source files.

2 Redefinition of comment character

You can override the default comment character with an entry line that begins with the comment_char keyword followed by the symbol for the desired character. The character symbol is defined in the character map (charmap) source file for the locale.

3 Redefinition of escape character

The escape character is the backslash (\) by default. It is used in decimal, hexadecimal, and octal constants to indicate when definition statements are continued to the next line of the source file. You can override the default escape character with an entry line that begins with the escape_char keyword followed by one or more blank characters,

then the symbol for the desired character. The character symbol is defined in the character map source file for the locale.

4 Header for locale category section

Section headers correspond to category names, which are LC_CTYPE, LC_COLLATE, LC_NUMERIC, LC_MONETARY, LC_MESSAGES, and LC_TIME.

5 Definition statement for the category

The format of these statements varies from one category to the next. In general, a statement begins with a keyword, followed by one or more spaces or tabs, then by the definition itself.

In place of any category definition statements, you can include a copy statement to include definition statements in another locale source file. For example:

```
copy en_US.ISO8859-1
```

If you include a copy statement, do not include other statements in the category.

6 Trailer for locale category section

Section trailers begin with the END keyword followed by the category name.

7 You can include sections for all locale categories or only a subset of categories. If you omit a section for a locale category from the source file, the definition for the omitted category is derived from the default locale (POSIX or C).

The following sections describe specific locale categories and illustrate the description with parts of the fr_FR.ISO8859-1@example.src locale source file. Section D.2 contains this source file in its entirety.

6.2.1 Defining the LC_CTYPE Locale Category

The LC_CTYPE section of a locale source file defines character classes and character attributes used in operations such as case conversion. Example 6–4 describes the definition for this section.

Example 6–4: LC_CTYPE Category Definition

```
#############
LC_CTYPE    1
#############

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
        <A-grave>;\
.
.
.
```

Example 6–4: LC_CTYPE Category Definition (cont.)

```
        <U-diaeresis>   2

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
        <a-grave>;\
.
.
.
        <u-diaeresis>   2

space   <tab>;<newline>;<vertical-tab>;<form-feed>;\
        <carriage-return>;<space>   2

cntrl   <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;\
        <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
        <form-feed>;<carriage-return>;\
.
.
.
        <SOS>;<SGCI>;<SCI>;<CSI>;<ST>;<OSC>;<PM>;<APC>   2

graph   <exclamation-mark>;<quotation-mark>;<number-sign>;\
.
.
.
        <u-circumflex>;<u-diaeresis>;<y-acute>;<thorn-icelandic>;<y-diaeresis>  2

# print class includes everything in the graph class above, plus <space>.

print   <exclamation-mark>;<quotation-mark>;<number-sign>;\
.
.
.
        <u-circumflex>;<u-diaeresis>;<y-acute>;<thorn-icelandic>;<y-diaeresis>;\
        <space>   2

punct   <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
        <left-parenthesis>;<right-parenthesis>;<asterisk>;\
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
        <greater-than-sign>;<question-mark>;<commercial-at>;\
        <left-square-bracket>;<backslash>;<right-square-bracket>;\
        <circumflex>;<underscore>;<grave-accent>;<left-brace>;\
        <vertical-line>;<right-brace>;<tilde>   2

digit   <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>  2

xdigit  <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>;\
        <A>;<B>;<C>;<D>;<E>;<F>;\
        <a>;<b>;<c>;<d>;<e>;<f>   2

blank   <space>;<tab>   2

toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);\
```

Example 6–4: LC_CTYPE Category Definition (cont.)

```
        (<z>,<Z>);\
        (<a-grave>,<A-grave>);\
        (<a-circumflex>,<A-circumflex>);\
        (<ae-ligature>,<AE-ligature>);\
        (<c-cedilla>,<C-cedilla>);\
        (<e-grave>,<E-grave>);\
        (<e-acute>,<E-acute>);\
        (<e-circumflex>,<E-circumflex>);\
        (<e-diaeresis>,<E-diaeresis>);\
        (<i-circumflex>,<I-circumflex>);\
        (<i-diaeresis>,<I-diaeresis>);\
        (<o-circumflex>,<O-circumflex>);\
        (<u-grave>,<U-grave>);\
        (<u-circumflex>,<U-circumflex>);\
        (<u-diaeresis>,<U-diaeresis>)   3

# tolower class is the inverse of toupper.

tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);\
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);\
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);\
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);\
        (<Z>,<z>);\
        (<A-grave>,<a-grave>);\
        (<A-circumflex>,<a-circumflex>);\
        (<AE-ligature>,<ae-ligature>);\
        (<C-cedilla>,<c-cedilla>);\
        (<E-grave>,<e-grave>);\
        (<E-acute>,<e-acute>);\
        (<E-circumflex>,<e-circumflex>);\
        (<E-diaeresis>,<e-diaeresis>);\
        (<I-circumflex>,<i-circumflex>);\
        (<I-diaeresis>,<i-diaeresis>);\
        (<O-circumflex>,<o-circumflex>);\
        (<U-grave>,<u-grave>);\
        (<U-circumflex>,<u-circumflex>);\
        (<U-diaeresis>,<u-diaeresis>)    3

END LC_CTYPE    4
```

1 Section header

2 Definition of character class

These definitions start with a keyword that stands for the character class (also referred to as a property) followed by one or more blank characters, then a list of symbols for all characters in that class. You can substitute the character's encoding for its symbol; however, specifying characters by their encodings diminishes the readability of the locale source file and makes it impossible to use the file with more than one codeset.

Although not illustrated in the example, you can specify a horizontal ellipsis (`...`) to represent a range of characters. In the string `<NUL>;...;<tab>`, for example, the ellipsis represents all characters

whose encodings are between the character whose symbol is <NUL> and the character whose symbol is <tab>. The symbols and their encodings are specified in the charmap file for the locale.

Character classes as defined by the X/Open UNIX standard are represented by the following keywords:

- upper (uppercase letter characters)
- lower (lowercase letter characters)
- alpha (all letter characters)

  By default, the alpha class is the combination of characters specified for the upper and lower classes. Because the sample locale does not explicitly define the alpha class, the default definition applies.
- space (white-space characters)
- cntrl (control characters)
- punct (punctuation characters)
- digit (numeric digits)
- xdigit (hexadecimal digits)
- blank (blank characters)
- graph

  By default, this class is the combination of characters in the alpha, digit, and punct classes.
- print

  By default, this class is the combination of characters in the alpha, digit, and punct classes, plus the space character.

From the application standpoint, there is also the class alnum. This class is rarely defined in a locale because it is always a combination of characters in the alpha and digit classes.

Unicode (*.UTF-8) locales include character classes as defined by the Unicode standard. See locale(4) for details about character classification for Unicode.

Certain locales, such as those for Asian languages like Japanese, may define nonstandard character classes.

3 Definitions of case conversion for letter characters

Case conversion definitions, which begin with the keywords toupper( ) and tolower, list symbols in pairs rather than individually. In the toupper( ) definition described here, the first symbol in the pair is the symbol for a lowercase letter and the second symbol is the symbol for that letter's uppercase equivalent. This definition determines

what a letter is converted to when functions, like `towupper( )` and `towlower( )`, perform case conversion on text data.

Locales that define nonstandard character classes may define other property conversion definitions that are used by the `wctrans( )` and `towctrans( )` functions.

4 Section trailer

The preceding example does not completely illustrate all the options you can use when defining the `LC_CTYPE` category. Additional options allow you to perform the following tasks:

- Use a `copy` statement to include the entire category definition from another locale

  When you use a `copy` statement, it must be the only entry between the section trailer and header.

- Omit any of the standard character classes or define different character classes

  The standard character classes are language specific. Therefore, the standard character classes may not apply to all languages. When you define a locale, use only the standard character classes that are appropriate for the locale's language. Depending on the language, it may be necessary to define nonstandardized classes.

  A definition for a nonstandardized character class must be preceded by the `charclass` statement to define a keyword for the class, followed by the class definition. For example:

```
charclass vowel
vowel        <a>;<e>;<i>;<o>;<u>;<y>
```

Applications can use the `wctype( )` and `iswctype( )` functions to determine and test all character classes (including user-defined ones). Applications can use class-specific functions, such as `iswalpha( )` and `iswpunct( )` to test the standard character classes.

Note

The `LC_CTYPE` category of the `fr_FR.ISO8859-1@example` locale is limited to letter characters in the French language. Some locale developers would define character classes to include characters in all the languages supported by the ISO 8859–1 character set. This practice allows locales for multiple Western European languages to use the same `LC_CTYPE` source definitions through a `copy` statement.

See locale(4) for additional rules and restrictions that apply to the LC_CTYPE category definition.

6.2.2 Defining the LC_COLLATE Locale Category

The LC_COLLATE section of a locale source file specifies how characters and strings are collated. Example 6–5 is part of an LC_COLLATE section.

Example 6–5: LC_COLLATE Category Definition

```
LC_COLLATE      1
order_start             forward;backward;forward  2
<NUL>    3
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
.
.
.

<APC>    3
<space>                 <space>;<space>;<space>
<exclamation-mark>      <exclamation-mark>;<exclamation-mark>;<exclamation-mark>
<quotation-mark>        <quotation-mark>;<quotation-mark>;<quotation-mark>
.
.
.

<a>                     <a>;<a>;<a>   3
<A>                     <a>;<a>;<A>
<feminine>              <a>;<feminine>;<feminine>
<a-acute>               <a>;<a-acute>;<a-acute>
<A-acute>               <a>;<a-acute>;<A-acute>
<a-grave>               <a>;<a-grave>;<a-grave>
<A-grave>               <a>;<a-grave>;<A-grave>
<a-circumflex>          <a>;<a-circumflex>;<a-circumflex>
<A-circumflex>          <a>;<a-circumflex>;<A-circumflex>
<a-ring>                <a>;<a-ring>;<a-ring>
<A-ring>                <a>;<a-ring>;<A-ring>
<a-diaeresis>           <a>;<a-diaeresis>;<a-diaeresis>
<A-diaeresis>           <a>;<a-diaeresis>;<A-diaeresis>
<a-tilde>               <a>;<a-tilde>;<a-tilde>
<A-tilde>               <a>;<a-tilde>;<A-tilde>
<ae-ligature>           <a>;<a><e>;<a><e>
<AE-ligature>           <a>;<a><e>;<A><E>
<b>                     <b>;<b>;<b>
<B>                     <b>;<b>;<B>
<c>                     <c>;<c>;<c>
<C>                     <c>;<c>;<C>
<c-cedilla>             <c>;<c-cedilla>;<c-cedilla>
<C-cedilla>             <c>;<c-cedilla>;<C-cedilla>
.
.
.

<z>                     <z>;<z>;<z>   3
<Z>                     <z>;<z>;<Z>
UNDEFINED   4
order_end   5
```

Example 6–5: LC_COLLATE Category Definition (cont.)

```
END LC_COLLATE      6
```

1 Section header

2 An `order_start` keyword that marks the beginning of a section with statements that assign collating weights to elements

Following the `order_start` keyword on the same line are sort directives, separated by semicolons (;) that apply to each sorting pass. Sort directives can include the following keywords.

- `forward`, which specifies that the comparison operation proceeds from the start of the string towards the end of the string.
- `backward`, which specifies that the comparison operation proceeds from the end of the string towards the start of the string.
- `position`, which specifies that the comparison operation considers the relative position of characters in the string that are not subject to the collating weight `IGNORE`. In other words, the first characters collated are those that do not have a collation weight of `IGNORE` and are the shortest distance from the start (`forward,position`) or end (`backward,position`) of the string.

  When a sort directive includes two keywords, the `position` keyword combined with either `forward` or `backward`, the two keywords are separated by a comma (,). The `position` keyword by itself is equivalent to the directive `forward,position`.

The number of sort directives corresponds to the number of weights each collating element is assigned in subsequent statements.

Each sort directive and its associated set of weights specify information for one pass, or level, of string comparison. The first directive applies when the string comparison operation applies the primary weight, the second when the string comparison operation applies the secondary weight, and so on. The number of levels required to collate strings correctly depends on language and cultural requirements and therefore varies from one locale to another. There is also a level number maximum, associated with the `COLL_WEIGHTS_MAX` setting in the `limits.h` and `sys/localedef.h` files. On Tru64 UNIX systems, you are limited to six collation levels (sort directives).

The `backward` directive is used for many languages to ensure that accented characters sort after unaccented characters only if the compared strings are otherwise equivalent.

The position directive is frequently used to handle characters, such as the hyphen (-) in Western European languages, whose significance can be relative to word position. For example, assume you wanted the word "o-ring" to collate in a word list before the word "or-ing", but do not want the hyphen to be considered until after strings are sorted by letters alone. You would need two sort directives and associated sets of weight specifiers to implement this order. For the first comparison operation, you specify forward as the sort directive, letters as the first weights for all letter characters, and IGNORE as the weight for the hyphen character. For the second, or a later, comparison operation, you specify forward position as the sort directive, IGNORE as the weight for all letter characters, and the hyphen as the weight for the hyphen character.

If you do not specify a sort directive, the default is forward.

3 Collation order statements for elements

These statements specify a character symbol, optionally followed by one or more blank characters (spaces or tabs), then the symbols for characters that have the same weight at each stage of the sort.

In the example, the sort order is control characters, followed by punctuation and digits, and then letters. Letters are sorted on multiple passes, with diacritics and case ignored on the first pass, diacritical marks being significant on the second pass, and case being significant on the third pass.

4 Collation order statement for characters not specified in other collation order statements

The UNDEFINED keyword begins a collation order statement to be applied to all characters that are defined in the locale's charmap file but not specified in other collation order statements. Characters that fall into the UNDEFINED category are considered in regular expressions to belong to the same equivalence class.

Always include the UNDEFINED collation order statement. If this statement is absent, the localedef command includes undefined characters at the end of the collating order and issues a warning.

Furthermore, if you place an UNDEFINED statement as the last collation order statement, the localedef command can sometimes compress all undefined characters into one entry. This action can reduce the size of the locale.

This locale specifies that any characters specified in the locale's charmap file, but not handled by other collation order statements, be ordered last.

An UNDEFINED statement can have an operand. For example, the IGNORE keyword causes any characters unspecified by other collation order statements to be ignored for the sort pass in which IGNORE

appears. If the following UNDEFINED statement had been included in the example, characters not specified in other collation order statements would be ignored in all sort passes defined by those statements:

```
UNDEFINED     IGNORE;IGNORE;IGNORE
```

5 Trailer to indicate the end of collation order statements

6 Trailer to indicate the end of the LC_COLLATE section

Example 6–5 contains only a few of the options that you can specify when defining the LC_COLLATE category. Additional options allow you to use the following:

- A copy statement to include the entire category definition from another locale

  A copy statement can be the only entry between the section trailer and header.

- Collating order statements that specify a string of characters, rather than single characters, as the collating elements

  In such cases, you first specify collating-element statements before the order_start statement to define symbols for the strings. You can then specify those symbols in collating order statements.

  For example:

```
collating-element <ch> from "<c><h>"
 ⋮
order_start forward;forward;backward
 ⋮
        <ch>    <Ch>;<ch>;<ch>
 ⋮
```

- Symbolic names, such as <UPPERCASE>, to use as weight specifiers in collation order statements

  You must define each symbolic name by using the collating-symbol statement in the source file before the order_start statement. You then include the symbol in the appropriate position in the list of collation order statements for collating elements. For example, if you wanted the symbol <LOW> to represent the lowest position in the collating order, <LOW> would be the line entry immediately following the order_start statement. A symbol such as <UPPERCASE> would be positioned on the line immediately preceding the section of collating order statements for uppercase letters.

  A symbol must occur before the first collation order statement in which it is used. Therefore, you cannot define a symbol for the highest position in the collating order.

After symbols are defined and positioned, you can use them as weights in collating order statements. For example:

```
collating-symbol <LOWERCASE>
collating-symbol <UNACCENTED>
 .
 .
 .
order_start forward;backward;forward;forward
 .
 .
 .
<UNACCENTED>
 .
 .
 .
<LOWERCASE>
<a>       <a>;<UNACCENTED>;<LOWERCASE>;IGNORE
 .
 .
 .
```

Remember that, because Unicode and dense code locales are equivalent, you can use the same charmaps and locale source for Unicode and dense code locales. However, Unicode and dense code characters that are defined in the charmap but not defined in the LC_COLLATE section may be sorted differently.

See locale(4) for detailed information on the LC_COLLATE category definition.

6.2.3 Defining the LC_MESSAGES Locale Category

The LC_MESSAGES section of a locale source file defines strings that are valid for affirmative and negative responses from users. Example 6–6 is an LC_MESSAGES section.

Example 6–6: LC_MESSAGES Category Definition

```
LC_MESSAGES    1

# yes expression. The following designates:
# "^([oO]|[oO][uU][iI])"

yesexpr       "<circumflex><left-parenthesis>\
<left-square-bracket><o><O><right-square-bracket>\
<vertical-line><left-square-bracket><o><O>\
<right-square-bracket><left-square-bracket><u><U>\
<right-square-bracket><left-square-bracket><i><I>\
<right-square-bracket><right-parenthesis>"    2

# no expression. The following designates:
# "^([nN]|[nN][oO][nN])"

noexpr        "<circumflex><left-parenthesis>\
<left-square-bracket><n><N><right-square-bracket>\
```

Example 6–6: LC_MESSAGES Category Definition (cont.)

```
<vertical-line><left-square-bracket><n><N>\
<right-square-bracket><left-square-bracket><o><O>\
<right-square-bracket><left-square-bracket><n><N>\
<right-square-bracket><right-parenthesis>"   3

# yes string. The following designates: "oui:o:O"

yesstr        "<o><u><i><colon><o><colon><O>"   4

# no string. The following designates: "non:n:N"

nostr         "<n><o><n><colon><n><colon><N>"   5
END LC_MESSAGES   6
```

1 Section header

2 Definition of an expression for a valid “yes” response

This entry consists of the `yesexpr` keyword followed by one or more spaces or tabs, and an extended regular expression that is delimited by double quotation marks.

This expression specifies that “oui” or “o” (case is ignored) is a valid affirmative response in this locale. The regular expression for `yesexpr` specifies individual characters by their symbols as defined in the locale’s `charmap` file.

3 Definition of an expression for a valid “no” response

This entry consists of the `noexpr` keyword followed by one or more spaces or tabs, and an extended regular expression that is delimited by double quotation marks.

This expression specifies that “non” or “n” (case is ignored) is a valid negative response in this locale.

4 Definition of a string for a valid “yes” response

This entry consists of the `yesstr` keyword followed one or more spaces or tabs, and a fixed string that is delimited by double quotation marks.

The `yesstr` entry is marked as LEGACY in the X/Open UNIX standard and is not included in the POSIX standard; however, some applications and systems software still might use `yesstr` rather than `yesexpr`. To ensure that your locale works correctly with such software, you should define `yesstr` in your locale. The X/Open UNIX standard defines a single fixed string for `yesstr`. The colon (:) separator, which allows multiple fixed strings to be specified, is an extension to the standard definition.

5 Definition of a string for a valid “no” response

This entry consists of the `nostr` keyword followed one or more spaces or tabs, and a fixed string that is delimited by double quotation marks.

The `nostr` entry is marked as LEGACY in the X/Open UNIX standard and is not included in the POSIX standard; however, some applications and systems software still might use `nostr` rather than `noexpr`. To ensure that your locale works correctly with such software, you should define `nostr` in your locale. The X/Open UNIX standard defines a single fixed string for `nostr`. The colon (:) separator, which allows multiple fixed strings to be specified, is an extension to the standard definition.

6 Section trailer

As an alternative to specifying symbol definitions, you can use the `copy` statement between the section header and trailer to duplicate an existing locale's definition of the `LC_MESSAGES` category. The `copy` statement represents a complete definition of the category and cannot be used when explicit symbol definitions are used.

6.2.4 Defining the LC_MONETARY Locale Category

The `LC_MONETARY` section of the locale source file defines the rules and symbols used to format monetary values. Application developers use the `localeconv( )` and `nl_langinfo( )` functions to determine the information defined in this section and apply formatting rules through the `strfmon( )` function. Example 6–7 is an `LC_MONETARY` section.

Example 6–7: LC_MONETARY Category Definition

```
LC_MONETARY   1

int_curr_symbol   "<F><R><F><space>"   2
currency_symbol   "<F>"    2
mon_decimal_point "<comma>"    2
mon_thousands_sep ""    2
mon_grouping      3;0    2
positive_sign     ""    2
negative_sign     "<hyphen>"    2
⋮

END LC_MONETARY   3
```

1 Section header

2 Symbol definitions

The entries in the example specify the following:

- The international currency symbol is FRF (French Franc) and the local currency symbol is F (Franc).
- The decimal point is the comma (,).
- No character is defined to group digits to the left of the decimal point.
- The digits in each grouping to the left of the decimal point in this locale are in groups of three. Because this locale does not define a default monetary thousands separator, the monetary grouping defined in this locale is significant only if the application uses a function to specify a thousands separator.
- The positive sign is null.
- The negative sign is the minus (–) character.

3 Section trailer

The following list describes the symbol names you can define in the LC_MONETARY section.

- int_curr_symbol

  The international currency symbol

- currency_symbol

  The local currency symbol

- mon_decimal_point

  The radix character, or decimal point, used in monetary formats

- mon_thousands_sep

  The character used to separate groups of digits to the left of the radix character

- mon_grouping

  The size of each group of digits to the left of the radix character. The character defined by mon_thousands_sep, if any, is inserted between the groups defined by mon_grouping. You can vary the size of groups by specifying multiple digits separated by a semicolon (;). For example, 3;2 specifies that the first group to the left of the radix character contains three digits and all subsequent groups contain 2 digits. On Tru64 UNIX systems, 3;0 and 3 are equivalent; that is, all digits to the left of the decimal point are grouped by three.

- positive_sign

  The string indicating that a monetary value is not negative

- negative_sign

  The string indicating that a monetary value is negative

- `int_frac_digits`

  The number of digits to be written to the right of the radix character when `int_curr_symbol` appears in the format

- `frac_digits`

  The number of digits to be written to the right of the radix character when `currency_symbol` appears in the format

- `p_cs_precedes`

  An integer that determines if the international or local currency symbol precedes a nonnegative value

- `p_sep_by_space`

  An integer that determines whether a space separates the international or local currency symbol from other parts of a formatted, nonnegative value

- `n_cs_precedes`

  An integer that determines if the international or local currency symbol precedes a negative value

- `n_sep_by_space`

  An integer that determines whether a space separates the international or local currency symbol from other parts of a formatted, negative value

- `p_sign_posn`

  An integer that indicates if or how the positive sign string is positioned in a nonnegative, formatted value

- `n_sign_posn`

  An integer that indicates how the negative sign string is positioned in a negative, formatted value

As an alternative to specifying symbol definitions, you can use the `copy` statement between the section header and trailer to duplicate an existing locale's definition of `LC_MONETARY`. The `copy` statement represents a complete definition of the category and cannot be used when explicit symbol definitions are used.

The `LC_MONETARY` definition is set to the euro character for the UTF-8 and ISO8859–15 locales of the languages that have fully adopted the euro. Because the euro character is not in the Latin-1 repertoire, the ISO8859–1 locales of the languages that have adopted the euro continue to use the pre-euro currency. For example, the Italian locale `it_IT.ISO8859-15` supports the euro; the Italian locale `it_IT.ISO8859-1` supports the lira.

See `locale`(4) for complete information about specifying `LC_MONETARY` symbol definitions.

6.2.5 Defining the LC_NUMERIC Locale Category

The LC_NUMERIC section of the locale source file defines the rules and symbols used to format numeric data. You can use the localeconv( ) and nl_langinfo( ) functions to access this formatting information. Example 6–8 is an LC_NUMERIC section.

Example 6–8: LC_NUMERIC Category Definition

```
LC_NUMERIC    1
decimal_point      "<comma>"    2
thousands_sep      ""    3
grouping           3;0    4

END LC_NUMERIC    5
```

1 Category header.

2 Definition of radix character (decimal point).

3 Definition of character used to separate groups of digits to the left of the radix character. In this locale, no default character is defined. Therefore, applications must supply this character, if needed.

4 The size of each group of digits to the left of the radix character. The character defined by thousands_sep, if any, is inserted between the groups defined by grouping.

You can vary the size of groups by specifying multiple digits separated by a semicolon (;). For example, 3;2 specifies that the first group to the left of the radix character contains three digits and all subsequent groups contain 2 digits. On Tru64 UNIX systems, 3;0 and 3 are equivalent; that is, all digits to the left of the radix character are grouped by three.

5 Category trailer.

Example 6–8 contains all of the symbols you can define in the LC_NUMERIC section. In place of any symbol definitions, you can specify a copy statement between the section header and trailer to include this section from another locale.

See locale(4) for detailed rules about LC_NUMERIC symbol definitions.

6.2.6 Defining the LC_TIME Locale Category

The LC_TIME section of a locale source file defines the interpretation of field descriptors supported by the date command. This section also affects the behavior of the strftime( ), wcsftime( ), strptime( ), and nl_langinfo( ) functions. Example 6–9 contains some of the symbols defined for the sample French locale.

Example 6–9: LC_TIME Category Definition

```
LC_TIME    1

abday    "<d><i><m>";\
         "<l><u><n>";\
         "<m><a><r>";\
         "<m><e><r>";\
         "<j><e><u>";\
         "<v><e><n>";\
         "<s><a><m>"    2

day      "<d><i><m><a><n><c><h><e>";\
         "<l><u><n><d><i>";\
         "<m><a><r><d><i>";\
         "<m><e><r><c><r><e><d><i>";\
         "<j><e><u><d><i>";\
         "<v><e><n><d><r><e><d><i>";\
         "<s><a><m><e><d><i>"    3

abmon    "<j><a><n>";\
         "<f><e-acute><v>";\
         "<m><a><r>";\
         "<a><v><r>";\
         "<m><a><i>";\
         "<j><u><n>";\
         "<j><u><l>";\
         "<a><o><u-circumflex>";\
         "<s><e><p>";\
         "<o><c><t>";\
         "<n><o><v>";\
         "<d><e-acute><c>"    4

mon      "<j><a><n><v><i><e><r>";\
         "<f><e-acute><v><r><i><e><r>";\
         "<m><a><r><s>";\
         "<a><v><r><i><l>";\
         "<m><a><i>";\
         "<j><u><i><n>";\
         "<j><u><i><l><l><e><t>";\
         "<a><o><u-circumflex><t>";\
         "<s><e><p><t><e><m><b><r><e>";\
         "<o><c><t><o><b><r><e>";\
         "<n><o><v><e><m><b><r><e>";\
         "<d><e-acute><c><e><m><b><r><e>"    5

# date/time format. The following designates this
# format: "%a %e %b %H:%M:%S %Z %Y"

d_t_fmt "<percent-sign><a><space><percent-sign><e>\
```

Example 6–9: LC_TIME Category Definition (cont.)

```
<space><percent-sign><b><space><percent-sign><H>\
<colon><percent-sign><M><colon><percent-sign><S>\
<space><percent-sign><Z><space><percent-sign><Y>"    6
⋮

END LC_TIME    7
```

1 Section header

2 Abbreviated names for days of the week

Use the %a conversion specifier to include these strings in formats.

3 Full names for days of the week

Use the %A conversion specifier to include these strings in formats.

4 Abbreviated names for months of the year

Use the %b conversion specifier to include these strings in formats.

5 Full names for months of the year

Use the %B conversion specifier to include these strings in formats.

6 Format for combined date and time information

The format combines field descriptors as defined for the `strftime( )` function. See `strftime`(3) for a complete list of field descriptors.

The specified format includes the field descriptors for the abbreviated day of the week (%a), the day of the month (%e), the number of hours in a 24-hour period (%H), the number of minutes (%M), and the number of seconds (%S), the time zone (%Z), and the full representation of the year (%Y). If the date were April 23, 1999, on the East coast of the United States, the format specified in this example would cause the `date` command to display `ven 23 avr 13:43:05 EDT 1999`.

7 Section trailer

Example 6–9 includes only some of the symbol definitions that are standard for the LC_TIME category. LC_TIMEalso allows you to specify the following standard definitions:

- `d_fmt`

  Format for the date alone; corresponds to the %x field descriptor

- `t_fmt`

  Format for the time alone; corresponds to the %X field descriptor

- `am_pm`

Format for the ante meridiem and post meridiem time strings; corresponds to the %p field descriptor

For example, the definition for the English language would be as follows:

```
am_pm        "<A><M>";"<P><M>"
```

- t_fmt_ampm

  Format for the time according to the 12-hour clock; corresponds to the %r field descriptor

- era

  Definition of how years are counted and displayed for each era in the locale. This format is for countries that use a year-counting system other than the Gregorian calendar. Such countries often use both the Gregorian calendar and a local era system.

- era_d_fmt

  Format of the date alone in era notation; corresponds to the %Ex field descriptor

- era_t_fmt

  Format of the time alone in era notation; corresponds to the %EX field descriptor

- era_d_t_fmt

  Format of both date and time in era notation; corresponds to the %Ec field descriptor

- alt_digits

  Definition of alternative symbols for digits; corresponds to the %O field descriptor

  This format is for countries that include alternative symbols in date strings.

As is true for other category sections, you can specify a copy statement to include all LC_TIME definitions from another locale. The operating system supports symbols and field descriptors in addition to those described here. See locale(4) for complete information on LC_TIME definitions.

6.3 Building Libraries to Convert Multibyte and Wide-Character Encodings

C Library routines rely on a set of special interfaces to convert characters to and from data file encoding and wide-character encoding (internal process code). By default, the C Library routines use interfaces that handle only single-byte characters. However, many are defined with entry points that

permit use of alternative interfaces for handling multibyte characters. The interfaces that can be tailored to a locale's codeset are called methods.

Locales with multibyte codesets must use methods. Also, some situations require a locale with single-byte codesets to supply methods. For example, a locale must supply a method when the corresponding interface is converting characters between data formats and the interface requires codeset-specific logic to do that operation correctly. However, a method is optional when the corresponding interface is working with data that has already been converted to wide-character format and the interface can apply logic that is valid for both single-byte and multibyte characters.

When a locale supplies a method, it must include a set of required methods as described in Section 6.3.1. See Section 6.3.2 for a description of optional methods.

Methods must be available on the system in a shareable library. This library and the functions that implement each method in the library are made known to the `localedef` command through a `methods` file. When the `localedef` command processes the `methods` file along with the `charmap` and `locale` source files, the resulting locale includes pointers to all methods that are supplied with the locale, and pointers to default implementations for optional methods that are not supplied with the locale. When you set the `LANG` variable to the newly built locale and run a command or application, methods are used wherever they have been enabled in the system software.

6.3.1 Required Methods

If your locale uses methods, it must supply the following:

- `__mbstopcs` (Section 6.3.1.1)
- `__mbtopc` (Section 6.3.1.2)
- `__pcstombs` (Section 6.3.1.3)
- `__pctomb` (Section 6.3.1.4)
- `mblen` (Section 6.3.1.5)
- `mbstowcs` (Section 6.3.1.6)
- `mbtowc` (Section 6.3.1.7)
- `wcstombs` (Section 6.3.1.8)
- `wctomb` (Section 6.3.1.9)
- `wcswidth` (Section 6.3.1.10)
- `wcwidth` (Section 6.3.1.11)

These methods make it possible for C Library functions to convert data between multibyte and wide-character formats.

6.3.1.1 Writing the __mbstopcs Method for the fgetws Function

The `fgetws( )` function uses the `__mbstopcs` method to convert the bytes in the standard I/O (`stdio`) buffer to a wide-character string. The function that implements this method must return the number of wide characters converted by the call.

This method is similar to the one for `mbstowcs( )` (see Section 6.3.1.6) but contains additional parameters to meet the needs of `fgetws( )`. By convention, a C source file for this method has the file name `__mbstopcs_`*`codeset`*`.c`, where *codeset* identifies the codeset for which the method is tailored. Example 6–10 is the file `__mbstopcs_sdeckanji.c`, which defines the `__mbstopcs` method used with the `ja_JP.sdeckanji` locale.

Example 6–10: The __mbstopcs_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h> [1]
#include <wchar.h>  [1]
#include <sys/localedef.h>  [1]

int __mbstopcs_sdeckanji(
        wchar_t *pwcs,  [2]
        size_t pwcs_len,  [3]
        const char *s,  [4]
        size_t s_len,  [5]
        int stopchr,  [6]
        char **endptr,  [7]
        int *err,  [8]
        _LC_charmap_t *handle )  [9]
{
    int cnt = 0;  [10]
    int pwcs_cnt = 0;  [10]
    int s_cnt = 0;  [10]

    *err = 0;  [11]

    while (1) {  [12]
        if (pwcs_cnt >= pwcs_len || s_cnt >= s_len) {
            *endptr = (char *)&(s[s_cnt]);
            break;
        }  [13]
        if ((cnt = __mbtopc_sdeckanji(&(pwcs[pwcs_cnt]),
            &(s[s_cnt]), (s_len - s_cnt), err)) == 0) {
            *endptr = (char *)&(s[s_cnt]);
            break;
        }  [14]
        pwcs_cnt++;  [15]
        if ( s[s_cnt] == (char) stopchr) {
            *endptr = (char *)&(s[s_cnt+1]);
            break;
        }  [16]
        s_cnt += cnt;  [17]
```

Example 6–10: The __mbstopcs_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
    }  18
    return (pwcs_cnt);  19
}
```

1 Include header files that contain constants and structures required for this method.

2 Points, through `pwcs`, to a buffer that stores the wide-character string.

3 Defines a variable, `pwcs_len`, to store the size of the `pwcs` buffer.

4 Points, through `s`, to a buffer that stores the multibyte character string being converted.

5 Defines a variable, `s_len`, to store the number of bytes of data in the `s` buffer.

This parameter is needed because the `fgetws( )` function reads from the standard I/O buffer, which does not contain null-terminated strings.

6 Defines a variable, `stopchr`, to contain a byte value that would force conversion to stop.

This value, typically `\n`, is passed to the method on the call from the `fgetws( )` function, which handles only one line of input for each call.

7 Defines a variable, `endptr`, that points to the byte following the last byte converted.

This pointer is needed to specify the starting character in the standard I/O buffer for the next call to `fgetws( )`.

8 Points, through `err`, to a variable that stores execution status for the call made by this method to the `mbtopc` method.

9 Points, through `hdl`, to a structure that points to the methods that parse character maps for this locale.

The `localedef` command creates and stores values in the `_LC_charmap_t` structure.

10 Initializes variables that indicate the number of bytes that a character uses in multibyte format (supplied by the `mbtopc` method) and the byte or character position in buffers that the `fgetws( )` function uses.

11 Sets `err` to zero (0) to indicate success.

12 Starts the `while` loop that converts the multibyte string.

13 Sets `endptr` and breaks out of the loop when there is either no more space in the buffer that stores wide-character data or no more data in the buffer that stores multibyte data.

14 Calls the `mbtopc` method to convert a character from multibyte format to wide-character format.

If the `mbtopc` method fails to convert a character and returns an error, the program breaks out of the loop and sets `endptr` to the first byte of the character that could not be converted.

The `err` variable contains one of the following status returns of the call to the `mbtopc` method:

- 0 indicates success.
- –1 indicates an invalid character.
- A value greater than 0 indicates that too few bytes remain in the multibyte character buffer to form a valid character. In this case, the return is the number of bytes required to form a valid character. The `fgetws( )` function can then refill the buffer and try again.

15 Increments the character position in the buffer that stores the wide-character data.

16 Sets `endptr` to the character following the character stored in `stopchr` if the `stopchr` character is encountered in the multibyte data.

17 Increments the byte position in the buffer that contains multibyte data.

18 Ends the `while` loop.

19 Returns the number of characters in the buffer that contains wide-character data.

6.3.1.2 Writing the __mbtopc Method for the getwc( ) Function

The `getwc( )` or `fgetwc( )` function calls the `__mbtopc` method to convert a multibyte character to a wide character. The method returns the number of bytes in the multibyte character that is converted. This method is similar to the one for `mbtowc` (see Section 6.3.1.7) but contains an additional parameter that `getwc( )` needs. By convention, a C source file for this method has the file name `__mbtopc_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–11 is the `__mbtopc_sdeckanji.c` file, which defines the `__mbtopc` method used with the `ja_JP.sdeckanji` locale.

Example 6–11: The __mbtopc_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h> 1
#include <wchar.h>
#include <sys/localedef.h>

/*
The algorithm for this conversion is:
s[0] < 0x9f:  PC = s[0]
s[0] = 0x8e:  PC = s[1] + 0x5f;
s[0] = 0x8f   PC = (((s[1] - 0xa1) << 7) | (s[2] - 0xa1)) + 0x303c
s[0] > 0xa1:0xa1 < s[1] < 0xfe
              PC = (((s[0] - 0xa1) << 7) | (s[1] - 0xa1)) + 0x15e
            0x21 < s[1] < 0x7e
              PC = (((s[0] - 0xa1) << 7) | (s[1] - 0x21)) + 0x5f1a
+-----------------+-----------+-----------+-----------+
|  process code   |   s[0]    |   s[1]    |   s[2]    |
+-----------------+-----------+-----------+-----------+
| 0x0000 - 0x009f | 0x00-0x9f |    --     |    --     |
| 0x00a0 - 0x00ff |   --      |    --     |    --     |
| 0x0100 - 0x015d | 0x8e      | 0xa1-0xfe |    --     | JIS X0201 RH
| 0x015e - 0x303b | 0xa1-0xfe | 0xa1-0xfe |    --     | JIS X0208
| 0x303c - 0x5f19 | 0x8f      | 0xa1-0xfe | 0xa1-0xfe | JIS X0212
| 0x5f1a - 0x8df7 | 0xa1-0xfe | 0x21-0xfe |    --     | UDC
+-----------------+-----------+-----------+-----------+
*/  2
int  __mbtopc_sdeckanji(
        wchar_t *pwc,  3
        char *ts,  4
        size_t maxlen,  5
        int *err,  6
        _LC_charmap_t *handle )  7
{
    wchar_t dummy;  8
    unsigned char *s = (unsigned char *)ts;  9
    if (s == NULL)
        return(0);  10
    if (pwc == (wchar_t *)NULL)
        pwc = &dummy;  11
    *err = 0;  12
    if (s[0] <= 0x8d) {
        if (maxlen < 1) {
            *err = 1;
            return(0);
        }
        else {
            *pwc = (wchar_t) s[0];
            return(1);
        }
    }  13
    else if (s[0] == 0x8e) {
        if (maxlen >= 2) {
            if (s[1] >=0xa1 && s[1] <=0xfe) {
                *pwc = (wchar_t) (s[1] + 0x5f);
                return(2);
            }
        }
        else {
            *err = 2;
            return(0);
        }
    }  14
```

Example 6–11: The __mbtopc_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
    else if (s[0] == 0x8f) {
        if (maxlen >= 3) {
            if ((s[1] >=0xa1 && s[1] <=0xfe) &&
                (s[2] >=0xa1 && s[2] <= 0xfe)) {
                *pwc = (wchar_t) (((s[1] - 0xa1) << 7) |
                       (wchar_t) (s[2] - 0xa1)) + 0x303c;
                return(3);
            }
        }
        else {
            *err = 3;
            return(0);
        }
    }  15

    else if (s[0] <= 0x9f) {
        if (maxlen < 1) {
            *err = 1;
            return(0);
        }
        else {
            *pwc = (wchar_t) s[0];
            return(1);
        }

    }  16
    else if (s[0] >= 0xa1 && s[0] <= 0xfe) {
        if (maxlen >= 2) {
            if  (s[1] >=0xa1 && s[1] <= 0xfe) {
                *pwc = (wchar_t) (((s[0] - 0xa1) << 7) |
                       (wchar_t) (s[1] - 0xa1)) + 0x15e;
                return(2);
            } else if  (s[1] >=0x21 && s[1] <= 0x7e) {
                *pwc = (wchar_t) (((s[0] - 0xa1) << 7) |
                       (wchar_t) (s[1] - 0x21)) + 0x5f1a;
                return(2);
            }
        }
        else {
            *err = 2;
            return(0);
        }

    }  17
    *err = -1;
    return(0);  18
}
```

1 Include header files that contain constants and structures required for this method.

2 Describes the algorithm used to determine the number of bytes and valid byte combinations for the different character sets that the codeset supports.

The codeset supports several character sets and each set contains characters of only one length. The value in the first byte indicates the character set and therefore the character length. For character sets with multibyte characters, one or more additional bytes must be examined to determine whether the value sequence identifies a character or is invalid.

3 Points, through `pwc`, to a buffer that stores the wide character.

4 Points, through `ts`, to a buffer that stores the bytes that are passed to the method from the calling function.

5 Declares a variable, `maxlen`, that stores the maximum number of bytes in the multibyte data.

This value is passed by the calling function.

6 Points, through `err`, to a buffer that stores execution status.

7 Points, through `handle`, to a structure that contains pointers to the methods that parse the character maps for this locale.

8 Declares a variable, `dummy`, to which `pwc` can be set to ensure a valid address.

9 Casts `ts` (an array of signed characters) to `s` (an array of unsigned characters).

This operation prevents problems when integer values are stored in the array and then referenced by index. Compilers apply **sign extension** to values when comparing a small signed data type, such as `char`, to a large signed data type, such as `int`. In this case, a condition such as the following is evaluated as true when you expect it to be false:

```
if (s[0] <= 0x8d
```

10 Returns zero (0) if the `s` buffer contains or points to `NULL`.

11 Stores the contents of `dummy` in the wide-character buffer if the `ts` buffer contains or points to `NULL`.

This operation ensures that `*pwc` always points to a valid address. If this were not the case, and a wide character is not stored in `pwc`, an application produces a segmentation fault by referring to this pointer.

12 Initializes `err` to zero (0) to indicate success.

13 Determines if the character is one of the single-byte characters that the codeset defines for values equal to or less than 0x8d.

If `s` contains no characters, returns zero (0) to indicate that no bytes were converted and sets `err` to 1 to indicate that 1 byte is needed to form a valid character.

If the byte value is in the range being tested, moves the associated process code value to pwc and returns 1 to indicate the number of bytes converted.

14 Determines if the character is one of the double-byte characters that the codeset defines for the value 0x8e (first byte) and the value range 0xa1 to 0xfe (second byte).

If yes, moves the associated process code value to the pwc buffer and returns 2 to indicate the number of bytes converted; otherwise, returns 0 to indicate that no conversion took place and sets err to 2 to specify that at least 2 bytes are needed to form a valid character.

15 Determines if the character is one of the triple-byte characters that the codeset defines for the value 0x8f (first byte), the range 0xa1 to 0xfe (second byte), and the range 0xa1 to 0xfe (third byte).

If yes, moves the associated process code value to pwc and returns 3 to indicate the number of bytes converted; otherwise, sets err to 3 to indicate that at least 3 bytes are needed and returns zero (0) to indicate that no character was converted.

16 Determines if the character is one of the single-byte characters that the codeset defines for the range 0x90 to 0x9f.

If there are no bytes in the standard I/O buffer, returns zero (0) to indicate that no bytes were converted and sets err to 1 to indicate that at least 1 byte is needed to form a valid character.

If the byte value is in the defined range, moves the associated process code value to pwc and returns 1 to indicate the number of bytes converted.

17 Determines if the character is one of the double-byte characters that the codeset defines for the range 0xa1 to 0xfe (first byte) and 0x21 to 0x7e (second byte).

If yes, moves the associated process code value to pwc buffer and returns 2 to indicate the number of bytes converted; otherwise, sets err to 2 to indicate that at least 2 bytes are needed to form a valid character and returns zero (0) to indicate that no bytes were converted.

18 Sets err to –1 to indicate that an invalid multibyte sequence was encountered and returns zero (0) to indicate that no bytes were converted.

These statements execute if the multibyte data in s satisfies none of the preceding if conditions.

6.3.1.3 Writing the __pcstombs Method for the fputws() Function

The fputws( ) function first calls the __pcstombs method to convert a string of characters from process (wide-character) code to multibyte

code. If this method returns –1 to indicate no support by the locale, `fputws( )` then calls `putwc( )` for each wide character in the string being converted. By convention, a C source file for this method has the file name `__pcstombs_`*`codeset`*`.c`, where *codeset* identifies the codeset for which this method is tailored. Example 6–12 is the file `__pcstombs_sdeckanji.c`, which defines the `__pcstombs` method used with the `ja_JP.sdeckanji` locale.

Example 6–12: The __pcstombs_sdeckanji Method for the ja_JP.sdeckanji Locale

```
int __pcstombs_sdeckanji()
{
       return -1;  [1]
}
```

[1] Returns –1 to indicate that the locale does not support the method.

This return causes the `fputws( )` function to use multiple calls to `putwc( )` to convert wide characters in the string.

If you choose to implement this method fully rather than writing it to return –1, your function implementation returns the number of wide characters converted and must include header files and parameters as illustrated in the following example:

```
#include <stdlib.h>
#include <wchar.h>
#include <sys/localedef.h>

int __pcstombs_newcodeset(
        wchar_t *pcsbuf,   [1]
        size_t pcsbuf_len,   [2]
        char *mbsbuf, [3]
        size_t mbsbuf_len, [4]
        char **endptr, [5]
        int *err,  [6]
        _LC_charmap_t *handle ) [7]
```

[1] Specifies a pointer to a buffer that contains the wide-character string.

[2] Specifies a variable with the length of the wide-character buffer.

This value is passed to the method on the call from `fputws( )`.

[3] Specifies a pointer to a buffer that contains the multibyte character string.

[4] Specifies a variable with the length of the multibyte character buffer.

This value is passed to the method on the call from `fputws( )`.

5 Points, through `endptr`, to a pointer to the byte position in the multibyte character buffer where the next character would begin if multiple calls to `fputws( )` are required to convert all the wide-character data.

6 Specifies a pointer to the execution status return.

If this method calls the `wctomb` method to perform the character conversion, the `wctomb` method sets this status. Otherwise, this method must incorporate the logic to perform wide-character to multibyte character conversion and set the status directly.

In any event, the `fputws( )` function expects the following values:

- 0 for success
- –1 to indicate that the wide-character value is invalid and therefore cannot be converted
- A positive value to indicate that the multibyte character buffer contains too few bytes after the last character to store the next character

  In this case, the value is the number of bytes required to store the next character. The `fputws( )` function can then empty the multibyte character buffer and try again.

7 Specifies a pointer to the `_LC_charmap_t` structure that stores pointers to the methods used with this locale.

The `__pcstombs` method performs the reverse of the operation that the `__mbstopcs` method performs (as described in Section 6.3.1.1). Because of the direction of the data conversion, the `__pcstombs` method behaves as follows:

- Does not require a variable for a stop conversion character, such as `\n`.
- Calls (or implements the operation performed by) the `wctomb` method rather than calling the `mbtowc` method to convert each character and determine the number of bytes it needs in the multibyte character buffer.

6.3.1.4 Writing a __pctomb Method

C Library functions currently do not use the `__pctomb` interface. The `putwc( )` function, for example, calls the `wctomb` method to convert a character from wide-character to multibyte character format. Nonetheless, the `localedef` command requires a method for this function when your locale supplies methods. By convention, a C source file for this method has the file name `__pctomb_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–13 is the `__pctomb_sdeckanji.c` file, which defines the `__pctomb` method used with the `ja_JP.sdeckanji` locale.

Example 6–13: The __pctomb_sdeckanji Method for the ja_JP.sdeckanji Locale

```
int __pctomb_sdeckanji()
{
        return -1;  [1]
}
```

[1] Returns –1 to indicate that the locale does not support this method.

6.3.1.5 Writing a Method for the mblen() Function

The `mblen( )` function uses the `mblen` method to return the number of bytes in a multibyte character. By convention, a C source file for this method has the file name `__mblen_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–14 is the `__mblen_sdeckanji.c` file, which defines the `mblen` method used with the `ja_JP.sdeckanji` locale.

Example 6–14: The __mblen_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h>  [1]
#include <wchar.h>
#include <sys/errno.h>
#include <sys/localedef.h>

/*
The algorithm for this conversion is:

s[0] < 0x9f:  1 byte
s[0] = 0x8e:  2 bytes
s[0] = 0x8f   3 bytes
s[0] > 0xa1   2 bytes
+-----------------+-----------+-----------+-----------+
|  process code   |   s[0]    |   s[1]    |   s[2]    |
+-----------------+-----------+-----------+-----------+
| 0x0000 - 0x009f | 0x00-0x9f |    --     |    --     |
| 0x00a0 - 0x00ff |   --      |    --     |    --     |
| 0x0100 - 0x015d | 0x8e      | 0xa1-0xfe |    --     | JIS X0201 RH
| 0x015e - 0x303b | 0xa1-0xfe | 0xa1-0xfe |    --     | JIS X0208
| 0x303c - 0x5f19 | 0x8f      | 0xa1-0xfe | 0xa1-0xfe | JIS X0212
| 0x5f1a - 0x8df7 | 0xa1-0xfe | 0x21-0xfe |    --     | UDC
+-----------------+-----------+-----------+-----------+
*/  [2]

int __mblen_sdeckanji(
        char *fs,  [3]
        size_t maxlen,  [4]
        _LC_charmap_t *handle )  [5]
{
    const unsigned char *s = (void *) fs;  [6]   if (s == NULL || *s == '\0')
        return(0);  [7]

    if (maxlen < 1) {
        _Seterrno(EILSEQ);
        return((size_t)-1);
```

Example 6–14: The __mblen_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
    }  8    if (s[0] <= 0x8d)
        return(1);  9

    else if (s[0] == 0x8e) {
        if (maxlen >= 2 && s[1] >=0xa1 && s[1] <=0xfe)
            return(2);
    }  10

    else if (s[0] == 0x8f) {
        if(maxlen >=3 && (s[1] >=0xa1 && s[1] <=0xfe) &&
            (s[2] >=0xa1 && s[2] <= 0xfe))
            return(3);
    }  11

    else if (s[0] <= 0x9f)
        return(1);  12

    else if (s[0] >= 0xa1) {
            if (maxlen >=2 && (s[0] <= 0xfe) )
                    if ( (s[1] >=0xa1 && s[1] <= 0xfe) ||
                       (s[1] >=0x21 && s[1] <= 0x7e) )
                        return(2);
    }  13

    _Seterrno(EILSEQ);
    return((size_t)-1);  14
}
```

1 Includes header files that contain constants and structures required by this method.

2 Describes the algorithm used to determine the number of bytes in the character and whether it is a valid byte sequence.

The codeset supports several character sets and each set contains characters of only one length. The value in the first byte indicates the character set and therefore the character length. For character sets with multibyte characters, one or more additional bytes must be examined to determine whether the value sequence identifies a character or is invalid.

3 Points, through `fs`, to a buffer that stores the byte string to be examined.

4 Defines a variable, `maxlen`, that stores the maximum length of a multibyte character.

This value is passed to the method by the `mblen( )` function.

5 Points, through `handle`, to a structure that stores pointers to the methods that parse character maps for this locale.

6 Casts `fs` (an array of signed characters) to `s` (an array of unsigned characters).

This operation prevents problems when integer values are stored in the array and then referenced by index. Compilers apply sign extension to values when comparing a small signed data type, such as `char`, to a large signed data type, such as `int`. In this case, a condition such as the following is evaluated as true when you expect it to be false:

```
if (s[0] <= 0x8d
```

7 Returns zero (0) to indicate that the character length is zero (0) bytes if `s` contains or points to `NULL`.

8 Returns –1 and sets `errno` to [EILSEQ] (invalid character sequence) if `maxlen` (the maximum number of bytes to consider) is 0 or a negative number.

To set `errno` in a way that works correctly with multithreaded applications, use `_Seterrno` rather than an assignment statement.

9 Determines if the first byte identifies a single-byte character whose value is equal to or less than 0x8d.

If yes, returns 1 to indicate that the character length is 1 byte.

10 Determines if the first byte identifies a double-byte character whose first byte contains the value 0x8e and second byte contains a value in the range 0xa1 to 0xfe.

If yes, returns 2 to indicate that the character length is 2 bytes.

11 Determines if the first byte identifies a triple-byte character whose first byte contains the value 0x8f and whose second and third bytes contain a value in the range 0xa1 to 0xfe.

If yes, returns 3 to indicate that the character length is 3 bytes.

12 Determines if the first byte identifies a single-byte character whose value is equal to or less than 0x9f.

If yes, returns 1 to indicate that the character length is 1 byte.

13 Determines if the first byte identifies a double-byte character whose first byte contains a value in the range 0xa1 to 0xfe and whose second byte contains a value in the range 0x21 to 0x7e.

If yes, returns 2 to indicate that the character length is 2 bytes.

14 Returns –1 and sets `errno` to [EILSEQ] to indicate an invalid multibyte sequence.

These statements execute if the multibyte data in the standard I/O buffer satisfies none of the preceding `if` conditions.

6.3.1.6 Writing a Method for the mbstowcs( ) Function

The `mbstowcs( )` function uses the `mbstowcs` method to convert a multibyte character string to process wide-character code and to return the number of resultant wide characters. By convention, a C source file for this method has the file name `__mbstowcs_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–15 is the `__mbstowcs_sdeckanji.c` file, which defines the `mbstowcs` method used with the `ja_JP.sdeckanji` locale.

Example 6–15: The __mbstowcs_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h>  1
#include <wchar.h>
#include <sys/localedef.h>

size_t __mbstowcs_sdeckanji(
        wchar_t *pwcs,  2
        const char *s,  3
        size_t n,  4
        _LC_charmap_t *handle )  5
{
    int len = n;  6
    int rc;  7
    int cnt;  8
    wchar_t *pwcs0 = pwcs;  9
    int mb_cur_max;  10

    if (s == NULL)
        return (0);  11

    mb_cur_max = MB_CUR_MAX;  12

    if (pwcs == (wchar_t *)NULL) {
        cnt = 0;
        while (*s != '\0') {
             if ((rc = __mblen_sdeckanji(s, mb_cur_max, handle)) == -1)
                return(-1);
             cnt++  ;
             s += rc;
        }
        return(cnt);
    }  13

    while (len-- > 0) {
        if ( *s == '\0') {
            *pwcs = (wchar_t) '\0';
            return (pwcs - pwcs0);
        }
        if ((cnt = __mbtowc_sdeckanji(pwcs, s, mb_cur_max, handle)) < 0)
            return(-1);
        s += cnt;
        ++pwcs;
    }  14
    return (n);  15
```

Example 6–15: The __mbstowcs_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
}
```

1 Includes header files that contain constants and structures required for this method.

2 Points, through `pwcs`, to a buffer that contains the wide-character string.

3 Points, through `s`, to a buffer that contains the multibyte character string.

4 Defines a variable, `n`, that contains the number of wide characters in `pwcs`.

5 Points, through `handle`, to a structure that stores pointers to the methods that parse character maps for this locale.

6 Assigns the number of wide characters in the `pwcs` buffer (the `n` value supplied by the calling function) to `len`.

7 Defines a variable, `rc`, that stores the return count from a call this method makes to the `mblen` function.

8 Defines a variable, `cnt`, that counts the bytes used by characters in the `s` buffer.

9 Saves the start of the wide-character string passed by the calling function in the `pwcs0` variable.

10 Defines a variable, `mb_cur_max`, that is later set to `MB_CUR_MAX` and used in a call to the `mblen` method.

11 Returns zero (0) if `s` is `NULL`.

A method should return zero (0) if the locale's character encoding is stateless and a nonzero value if the locales's character encoding is stateful.

12 Assigns the value defined for `MB_CUR_MAX` to `mb_cur_max` for use on the following call to the `mblen` method.

13 Checks to see if a `NULL` pointer was passed from the calling function and, if yes, calls the `mblen` method to calculate the size of the wide-character string.

You can request the size of the `pwcs` buffer (for memory allocation purposes) by passing a null wide character as the `pwcs` parameter in the call to `mbstowcs( )`. You can then use the return value to efficiently

allocate memory space for the application's wide-character buffer before calling `mbstowcs( )` again to actually convert the multibyte string.

14 Converts bytes in the multibyte character buffer by calling the `__mbtowc` method until a null character (end-of-string) is encountered.

Stops processing and returns the number of wide characters in the `pwcs` buffer if a null character is encountered; increments the byte position in the multibyte character buffer by an appropriate number each time a character is successfully converted.

This `while` loop uses the condition `len-- > 0` to ensure that processing stops when the `pwcs` buffer is full. The first `if` condition in the loop makes sure that, if the multibyte string in the `s` buffer is null terminated, the associated null terminator in the `pwcs` buffer is not included in the wide-character count that the `mbtowcs( )` function returns to the application.

15 Returns the value in `n` to indicate the resultant number of wide characters in the `pwcs` buffer.

This statement executes if the `pwcs` buffer runs out of space before a null is encountered in the `s` buffer.

6.3.1.7 Writing a Method for the mbtowc() Function

The `mbtowc( )` function uses the `mbtowc` method to convert a multibyte character to a wide character and to return the number of bytes in the multibyte character that was converted. By convention, a C source file for this method has the file name `__mbtowc_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–16 is the `__mbtowc_sdeckanji.c` file, which defines the `mbtowc` method used with the `ja_JP.sdeckanji` locale.

Example 6–16: The __mbtowc_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h>  1
#include <wchar.h>
#include <sys/errno.h>
#include <sys/localedef.h>

/*
The algorithm for this conversion is:

s[0] < 0x9f:  PC = s[0]
s[0] = 0x8e:  PC = s[1] + 0x5f;
s[0] = 0x8f   PC = (((s[1] - 0xa1) << 7) | (s[2] - 0xa1)) + 0x303c
s[0] > 0xa1:0xa1 < s[1] < 0xfe
              PC = (((s[0] - 0xa1) << 7) | (s[1] - 0xa1)) + 0x15e
0x21 < s[1] < 0x7e
              PC = (((s[0] - 0xa1) << 7) | (s[1] - 0x21)) + 0x5f1a

+-----------------+-----------+-----------+-----------+
|  process code   |   s[0]    |   s[1]    |   s[2]    |
```

Example 6–16: The __mbtowc_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
+-----------------+-----------+-----------+-----------+
| 0x0000 - 0x009f | 0x00-0x9f |    --     |    --     |
| 0x00a0 - 0x00ff |   --      |    --     |    --     |
| 0x0100 - 0x015d | 0x8e      | 0xa1-0xfe |    --     | JIS X0201 RH
| 0x015e - 0x303b | 0xa1-0xfe | 0xa1-0xfe |    --     | JIS X0208
| 0x303c - 0x5f19 | 0x8f      | 0xa1-0xfe | 0xa1-0xfe | JIS X0212
| 0x5f1a - 0x8df7 | 0xa1-0xfe | 0x21-0xfe |    --     | UDC
+-----------------+-----------+-----------+-----------+
*/  2
int __mbtowc_sdeckanji(
        wchar_t *pwc,  3
        const char *ts,  4
        size_t maxlen,  5
        _LC_charmap_t *handle )  6
{
    unsigned char *s = (unsigned char *)ts;  7
    wchar_t dummy;  8

    if (s == NULL)
        return(0);  9

    if (maxlen < 1) {
        _Seterrno(EILSEQ);
        return((size_t)-1);
    }  10

    if (pwc == (wchar_t *)NULL)
        pwc = &dummy;  11

    if (s[0] <= 0x8d) {
        *pwc = (wchar_t) s[0];
        if (s[0] != '\0')
            return(1);
        else
            return(0);
    }  12

    else if (s[0] == 0x8e) {
        if ( (maxlen >= 2) && ((s[1] >=0xa1) && (s[1] <=0xfe))) {
            *pwc = (wchar_t) (s[1] + 0x5f); /* 0x100 - 0xa1 */
            return(2);
        }
    }  13

    else if (s[0] == 0x8f) {
        if((maxlen >= 3) && (((s[1] >=0xa1) && (s[1] <=0xfe))
           && ((s[2] >=0xa1) && (s[2] <= 0xfe)))) {
                *pwc = (wchar_t) (((s[1] - 0xa1) << 7) |
                   (wchar_t) (s[2] - 0xa1)) + 0x303c;
           return(3);
        }
    }  14

    else if (s[0] <= 0x9f) {
        *pwc = (wchar_t) s[0];
        if (s[0] != '\0')
            return(1);
        else
            return(0);
```

Example 6–16: The __mbtowc_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
    }  15

    else if (((s[0] >= 0xa1) && (s[0] <= 0xfe)) && (maxlen >= 2)){
            if (((s[1] >=0xa1) && (s[1] <= 0xfe))){
                    *pwc = (wchar_t) (((s[0] - 0xa1) << 7) |
                                (wchar_t)(s[1] - 0xa1)) + 0x15e;
                    return(2);
            } else if (((s[1] >=0x21) && (s[1] <= 0x7e))){
                    *pwc = (wchar_t) (((s[0] - 0xa1) << 7) |
                                (wchar_t)(s[1] - 0x21)) + 0x5f1a;
                    return(2);
            }
    }  16
    _Seterrno(EILSEQ);
    return(-1);  17
}
```

1 Includes header files that contain constants and structures required for this method.

2 Describes the algorithm used to determine the number of bytes in the character and whether it is a valid byte sequence.

The codeset supports several character sets and each set contains characters of only one length. The value in the first byte indicates the character set and therefore the character length. For character sets with multibyte characters, one or more additional bytes must be examined to determine whether the value sequence identifies a character or is invalid.

3 Points, through `pwc`, to a buffer that contains the wide character.

4 Points, through `ts`, to a buffer that contains values in multibyte character format.

5 Defines a variable, `maxlen`, that stores the maximum length of a multibyte character.

This value is passed from the calling function; the value will have been set to `MB_CUR_MAX` on the original call made by the application programmer.

6 Points, through `handle`, to a structure that stores pointers to the methods that parse character maps for this locale.

7 Casts `ts` (an array of signed characters) to `s` (an array of unsigned characters).

This operation prevents problems when integer values are stored in the array and then referenced by index. Compilers apply sign extension to values when comparing a small signed data type, such as `char`, to

a large signed data type, such as `int`. In this case, a condition such as the following would be evaluated as true when you would expect it to be false:

```
if (s[0] <= 0x8d
```

8 Defines a variable, `dummy`, that can be assigned to `pwc` to ensure `pwc` points to a valid address.

9 Returns zero (0) to indicate that the locale's character encoding is stateless if `s` contains or points to `NULL`.

If passed a `NULL` pointer, this method should return a value to indicate whether the locale's character encoding is stateful or stateless. Return a nonzero value if your locale's character encoding is stateful.

10 Returns –1 cast to `size_t` and sets `errno` to [EILSEQ] (invalid byte sequence) if the multibyte data buffer is less than 1 byte in length.

11 Stores the contents of `dummy` in the wide-character buffer if the `ts` buffer contains or points to `NULL`.

This operation ensures that `pwc` always points to a valid address; otherwise, an application could produce a segmentation fault by referring to this pointer when a wide character has not been stored in `pwc`.

12 Determines if the first byte identifies a single-byte character whose value is equal to or less than 0x8d.

If yes, stores the associated process code value in the `pwc` buffer and returns 1 to indicate that the character length is 1 byte.

13 Determines if the first byte identifies a double-byte character whose first byte contains the value 0x8e and second byte contains a value in the range 0xa1 to 0xfe.

If yes, stores the associated process code value in the `pwc` buffer and returns 2 to indicate that the character length is 2 bytes.

14 Determines if the first byte identifies a triple-byte character whose first byte contains the value 0x8f and whose second and third bytes contain a value in the range 0xa1 to 0xfe.

If yes, stores the associated process code value in the `pwc` buffer and returns 3 to indicate that the character length is 3 bytes.

15 Determines if the first byte identifies a single-byte character whose value is equal to or less than 0x9f.

If yes, stores the associated process code value in the `pwc` buffer and returns 1 to indicate that the character length is 1 byte.

16 Determines if the first byte identifies a double-byte character whose first byte contains a value in the range x0a1 to x0fe and whose second byte contains a value in the range 0x21 to 0x7e.

If yes, stores the associated process code value in the `pwc` buffer and returns 2 to indicate that the character length is 2 bytes.

17 Returns –1 and sets `errno` to [EILSEQ] to indicate that an invalid multibyte sequence was encountered.

These statements execute if the multibyte data in the `s` buffer satisfies none of the preceding `if` conditions.

6.3.1.8 Writing a Method for the wcstombs( ) Function

The `wcstombs( )` function calls the `wcstombs` method to convert a wide-character string to a multibyte character string and to return the number of bytes in the resultant multibyte character string. By convention, a C source file for this method has the file name `__wcstombs_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–17 is the `__wcstombs_sdeckanji.c` file, which defines the `wcstombs` method used with the `ja_JP.sdeckanji` locale.

Example 6–17: The __wcstombs_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h>  1
#include <wchar.h>
#include <limits.h>
#include <sys/localedef.h>

size_t __wcstombs_sdeckanji(
        char *s,  2
        const wchar_t *pwcs,  3
        size_t n,  4
        _LC_charmap_t *handle )  5
{
    int cnt=0;  6
    int len=0;  7
    int i=0;  8
    char tmps[MB_LEN_MAX+1];  9

    if ( s == (char *)NULL) {
        cnt = 0;
        while (*pwcs != (wchar_t)'\0') {
            if ((len = __wctomb_sdeckanji(tmps, *pwcs)) == -1)
                    return(-1);
            cnt += len;
            pwcs++;
        }
        return(cnt);
    }  10

    if (*pwcs == (wchar_t)'\0') {
        *s = '\0';
        return(0);
    }  11
```

Example 6–17: The __wcstombs_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
    while (1) {  12

        if ((len = __wctomb_sdeckanji(tmps, *pwcs)) == -1)
            return(-1);  13

        else if (cnt+len > n) {
            *s = '\0';
            break;
        }  14

        if (tmps[0] == '\0') {
            *s = '\0';
            break;
        }  15

        for (i=0; i<len; i++) {
            *s = tmps[i];
            s++;
        }  16

        cnt += len;  17

        if (cnt == n)
            break;  18

        pwcs++;  19
    }  20

    if (cnt == 0)
        cnt = len;  21
    return (cnt);  22
}
```

1 Includes header files that contain constants and structures required for this method.

2 Points, through `s`, to a buffer that stores the multibyte character string that this method passes to the calling function.

3 Points, through `pwcs`, to a buffer that stores the wide-character string that is being converted.

4 Defines a variable, `n`, that stores the maximum number of bytes in the multibyte character string buffer.

This value is supplied by the calling function.

5 Points, through `handle,` to a structure that points to the methods that parse character maps for this locale.

6 Initializes a variable, `cnt`, that is incremented by the number of bytes (`len`) of each converted character.

7 Initializes a variable, `len`, that stores the length of each converted character.

8 Initializes a variable, `i`, that is used to index the bytes in each multibyte character when moving a converted character from temporary storage to `s`.

9 Defines a temporary buffer, `tmps`, that stores the multibyte character returned to this method from a call to the `wctomb` method.

10 Checks to see if a `NULL` was passed from the calling function in the `s` buffer.

If yes, calls the `wctomb` method to calculate the number of bytes required for converted characters (excluding the null terminator) in the multibyte character buffer.

You can request the size of the `s` buffer (for memory allocation purposes) by passing a null byte as the data in the `s` parameter on the call to `wcstombs( )`. You can then use the return value to efficiently allocate memory space for the application's wide-character buffer before calling `wcstombs( )` again to actually convert the wide-character string.

11 Returns zero (0) to indicate that no multibyte characters resulted and sets `s` to `NULL` if `pwcs` points to `NULL`.

12 Starts a `while` loop to process characters in the wide-character string.

13 Converts characters in the wide-character buffer by calling the `wctomb` method; returns –1 to indicate an invalid character if `wctomb` returns –1.

14 Terminates `s` with `NULL` and breaks out of the `while` loop if there is no room in `s` for the character just converted by `wctomb`.

15 Moves a null terminator to `s` and breaks out of the `while` loop when a `NULL` is encountered in `s`.

16 Appends each byte in `tmps` to `s` if the current wide character is not a `NULL`.

17 Increments `cnt` by the number of bytes (`len`) occupied by this character in multibyte format.

18 Breaks out of the `while` loop without adding a null terminator if the number of bytes processed equals `n` (the maximum number of bytes in `s`).

19 Increments `pwcs` to point to the next wide character to be converted.

20 Ends the `while` loop that converts each wide character.

21 Ensures that zero (0) is returned if `s` does not contain enough space for even one character.

22 Returns the number of bytes in the resultant multibyte character string.

6.3.1.9 Writing a Method for the wctomb() Function

The `wctomb( )` function calls the `wctomb` method to convert a wide character to a multibyte character and to return the number of bytes in the resultant multibyte character. By convention, a C source file for this method has the file name `__wctomb_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–18 is the `__wctomb_sdeckanji.c` file, which defines the `wctomb` method for the `ja_JP.sdeckanji` locale.

Example 6–18: The __wctomb_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h>  1
#include <wchar.h>
#include <sys/errno.h>
#include <sys/localedef.h>

/*
  The algorithm for this conversion is:

PC <= 0x009f:                 s[0] = PC
PC >= 0x0100 and PC <=0x015d: s[0] = 0x8e
                              s[1] = PC - 0x005f
PC >= 0x015e and PC <=0x303b: s[0] = ((PC - 0x015e) >> 7) + 0x00a1
                              s[1] = ((PC - 0x015e) & 0x007f) + 0x00a1
PC >= 0x303c and PC <=0x5f19: s[0] = 0x8f
                              s[1] = ((PC - 0x303c) >> 7) + 0x00a1
                              s[2] = ((PC - 0x303c) & 0x007f) + 0x00a1
PC >= 0x5f1a and PC <=0x8df7  s[0] = ((PC - 0x5f1a) >> 7) + 0x00a1
                              s[1] = ((PC - 0x5f1a) & 0x007f) + 0x0021

+-----------------+-----------+-----------+-----------+
|  process code   |   s[0]    |   s[1]    |   s[2]    |
+-----------------+-----------+-----------+-----------+
| 0x0000 - 0x009f | 0x00-0x9f |    --     |    --     |
| 0x00a0 - 0x00ff |   --      |    --     |    --     |
| 0x0100 - 0x015d | 0x8e      | 0xa1-0xfe |    --     | JIS X0201 RH
| 0x015e - 0x303b | 0xa1-0xfe | 0xa1-0xfe |    --     | JIS X0208
| 0x303c - 0x5f19 | 0x8f      | 0xa1-0xfe | 0xa1-0xfe | JIS X0212
| 0x5f1a - 0x8df7 | 0xa1-0xfe | 0x21-0xfe |    --     | UDC
+-----------------+-----------+-----------+-----------+
*/  2

int __wctomb_sdeckanji(
        char *s,    3
        wchar_t wc,    4
        _LC_charmap_t *handle )    5
{
    if (s == (char *)NULL)
        return(0);    6

    if (wc <= 0x9f) {
        s[0] = (char) wc;
        return(1);
    }    7

    else if ((wc >= 0x0100) && (wc <= 0x015d)) {
        s[0] = 0x8e;
        s[1] = wc - 0x5f;
```

Example 6–18: The __wctomb_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
        return(2);
    }   8

    else if ((wc >=0x015e) && (wc <= 0x303b)) {
        s[0] = (char) (((wc - 0x015e) >> 7) + 0x00a1);
        s[1] = (char) (((wc - 0x015e) & 0x007f) + 0x00a1);
        return(2);
    }   9

    else if ((wc >=0x303c) && (wc <= 0x5f19)) {
        s[0] = 0x8f;
        s[1] = (char) (((wc - 0x303c) >> 7) + 0x00a1);
        s[2] = (char) (((wc - 0x303c) & 0x007f) + 0x00a1);
        return(3);
    }   10

    else if ((wc >=0x5f1a) && (wc <= 0x8df7)) {
        s[0] = (char) (((wc - 0x5f1a) >> 7) + 0x00a1);
        s[1] = (char) (((wc - 0x5f1a) & 0x007f) + 0x0021);
        return(2);
    }   11

    _Seterrno(EILSEQ);
    return(-1);   12
}
```

1 Includes header files that contain constants and structures required for this method.

2 Describes the conversion algorithm that this method uses.

Each character set supported by the codeset corresponds to a unique range of wide-character (process code) values. Within each character set, multibyte characters are of uniform length (1, 2, or 3 bytes). Therefore, the range in which each wide-character value falls indicates the number of bytes required for the character in multibyte format. The wide-character value itself determines the specific byte value or values for the character in multibyte format.

3 Points, through `s`, to a buffer that stores the multibyte character.

4 Defines the `wc` variable that stores the wide character.

5 Points, through `handle`, to a structure that stores pointers to the methods that parse the character maps for this locale.

6 Returns zero (0) to indicate that no characters were converted if `s` points to `NULL`.

7 If the wide-character value is equal to or less than 0x9f, moves that value into the first byte of the `s` array and returns 1 to indicate that the converted character is 1 byte in length.

8 If the wide-character value is in the range 0x0100 to 0x015d, moves the value 0x8e to the first byte and a calculated value to the second byte of the `s` array; returns 2 to indicate that the converted character is 2 bytes in length.

9 If the wide-character value is in the range 0x015e to 0x303b, moves calculated values to the first and second bytes of the `s` array and returns 2 to indicate that the converted character is 2 bytes in length.

10 If the wide-character value is in the range 0x303c to 0x5f19, moves 0x8f to the first byte and calculated values to the second and third bytes of the `s` array; returns 3 to indicate that the converted character is 3 bytes in length.

11 If the wide-character value is in the range 0x5f1a to 0x8df7, moves calculated values to the first and second bytes of the `s` array, and returns 2 to indicate that the converted character is 2 bytes in length.

12 Sets `errno` to [EILSEQ] and returns –1 to indicate that the wide-character value is invalid.

These statements execute if the wide-character values satisfy none of the preceding conditions.

6.3.1.10 Writing a Method for the wcswidth( ) Function

The `wcswidth( )` function uses the `wcswidth` method to determine the number of columns required to display a wide-character string. By convention, a C source file for this method has the file name `__wcswidth_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored. Example 6–19 is the `__wcswidth_sdeckanji.c` file, which defines the `wcswidth` method used for the `ja_JP.sdeckanji` locale.

Example 6–19: The __wcswidth_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h>  1
#include <wchar.h>
#include <sys/localedef.h>

/*
The algorithm for this conversion is:

PC <= 0x009f:                 s[0] = PC
PC >= 0x0100 and PC <=0x015d: s[0] = 0x8e
                              s[1] = PC - 0x005f
PC >= 0x015e and PC <=0x303b: s[0] = ((PC - 0x015e) >> 7) + 0x00a1
                              s[1] = ((PC - 0x015e) & 0x007f) + 0x00a1
PC >= 0x303c and PC <=0x5f19: s[0] = 0x8f
                              s[1] = ((PC - 0x303c) >> 7) + 0x00a1
                              s[2] = ((PC - 0x303c) & 0x007f) + 0x00a1
PC >= 0x5f1a and PC <=0x8df7  s[0] = ((PC - 0x5f1a) >> 7) + 0x00a1
                              s[1] = ((PC - 0x5f1a) & 0x007f) + 0x0021
```

Example 6–19: The __wcswidth_sdeckanji Method for the ja_JP.sdeckanji Locale (cont.)

```
+-----------------+-----------+-----------+-----------+
|  process code   |   s[0]    |   s[1]    |   s[2]    |
+-----------------+-----------+-----------+-----------+
| 0x0000 - 0x009f | 0x00-0x9f |    --     |    --     |
| 0x00a0 - 0x00ff |   --      |    --     |    --     |
| 0x0100 - 0x015d | 0x8e      | 0xa1-0xfe |    --     | JIS X0201 RH
| 0x015e - 0x303b | 0xa1-0xfe | 0xa1-0xfe |    --     | JIS X0208
| 0x303c - 0x5f19 | 0x8f      | 0xa1-0xfe | 0xa1-0xfe | JIS X0212
| 0x5f1a - 0x8df7 | 0xa1-0xfe | 0x21-0xfe |    --     | UDC
+-----------------+-----------+-----------+-----------+
*/  [2]

int __wcswidth_sdeckanji(
        const wchar_t *wcs,  [3]
        size_t n,  [4]
        _LC_charmap_t *hdl )  [5]
{
    int len;  [6]
    int i;  [7]

    if (wcs == (wchar_t *)NULL || *wcs == (wchar_t)NULL)
        return(0);  [8]

    len = 0;  [9]
    for (i=0; wcs[i] != (wchar_t)NULL && i<n; i++) {  [10]

        if (wcs[i] <= 0x9f)
             len += 1;  [11]

        else if ((wcs[i] >= 0x0100) && (wcs[i] <= 0x015d))
             len += 1;  [12]

        else if ((wcs[i] >=0x015e) && (wcs[i] <= 0x303b))
             len += 2;  [13]

        else if ((wcs[i] >=0x303c) && (wcs[i] <= 0x5f19))
            len += 2;  [14]

        else if ((wcs[i] >=0x5f1a) && (wcs[i] <= 0x8df7))
            len += 2;  [15]

        else
            return(-1);  [16]
    }  [17]

    return(len);  [18]
}
```

[1] Includes header files that contain constants and structures required for this method.

[2] Describes the algorithm used to determine the required display width.

Each character's display width is either 1 or 2 columns, depending on the character set to which a character belongs. Display width is different from the size of the character in multibyte format; for example,

triple-byte characters require 2 display columns and double-byte characters can require either 1 or 2 display columns.

3 Points, through `wcs`, to a buffer that stores the wide-character string for which display width information is requested.

4 Defines a variable, `n`, that stores the maximum size of the `wcs` buffer.

5 Points, through `hdl`, to a structure that stores pointers to the methods that parse character maps for this locale.

6 Defines a variable, `len`, that stores the display width in bytes/columns.

7 Defines a variable, `i`, that functions as a loop counter.

8 Returns zero (0) if `wcs` contains or points to `NULL`.

9 Initializes `len` to zero (0).

10 Begins a `for` loop that processes each wide character in the `wcs` buffer and increments the wide-character pointer.

11 Increments `len` by 1 if the value of the current wide character is less than or equal to 0x9f.

12 Increments `len` by 1 if the value of the current wide character is in the range 0x0100 to 0x015d.

13 Increments `len` by 2 if the value of the current wide character is in the range 0x015e to 0x303b.

14 Increments `len` by 2 if the value of the current wide character is in the range 0x303c to 0x5f19.

15 Increments `len` by 2 if the value of the current wide character is in the range 0x5f1a to 0x8df7.

16 Returns –1 to indicate that the string contains an invalid wide character.

This statement executes if a value that satisfies none of the preceding conditions is encountered in the string. The calling function, `wcswidth( )`, also returns –1 if the wide character is nonprintable; however, this condition is evaluated at the level of the calling function and does not need to be evaluated by the method.

17 Ends the `for` loop that processes wide characters in the `wcs` buffer.

18 Returns `len` to indicate the number of columns required to display the wide-character string.

6.3.1.11 Writing a Method for the wcwidth() Function

The `wcwidth( )` function uses the `wcwidth` method to determine the number of columns required to display a wide character. By convention, a C source file for this method has the file name `__wcwidth_`*`codeset`*`.c`, where *`codeset`* identifies the codeset for which this method is tailored.

Example 6–20 is the __wcwidth_sdeckanji.c file, which defines the wcwidth method used with the ja_JP.sdeckanji locale.

Example 6–20: The __wcwidth_sdeckanji Method for the ja_JP.sdeckanji Locale

```
#include <stdlib.h>  [1]
#include <wchar.h>
#include <sys/localedef.h>

/*
The algorithm for this conversion is:

PC <= 0x009f:                 s[0] = PC
PC >= 0x0100 and PC <=0x015d: s[0] = 0x8e
                              s[1] = PC - 0x005f
PC >= 0x015e and PC <=0x303b: s[0] = ((PC - 0x015e) >> 7) + 0x00a1
                              s[1] = ((PC - 0x015e) & 0x007f) + 0x00a1
PC >= 0x303c and PC <=0x5f19: s[0] = 0x8f
                              s[1] = ((PC - 0x303c) >> 7) + 0x00a1
                              s[2] = ((PC - 0x303c) & 0x007f) + 0x00a1
PC >= 0x5f1a and PC <=0x8df7  s[0] = ((PC - 0x5f1a) >> 7) + 0x00a1
                              s[1] = ((PC - 0x5f1a) & 0x007f) + 0x0021
+-----------------+-----------+-----------+-----------+
|  process code   |   s[0]    |   s[1]    |   s[2]    |
+-----------------+-----------+-----------+-----------+
| 0x0000 - 0x009f | 0x00-0x9f |    --     |    --     |
| 0x00a0 - 0x00ff |   --      |    --     |    --     |
| 0x0100 - 0x015d | 0x8e      | 0xa1-0xfe |    --     | JIS X0201 RH
| 0x015e - 0x303b | 0xa1-0xfe | 0xa1-0xfe |    --     | JIS X0208
| 0x303c - 0x5f19 | 0x8f      | 0xa1-0xfe | 0xa1-0xfe | JIS X0212
| 0x5f1a - 0x8df7 | 0xa1-0xfe | 0x21-0xfe |    --     | UDC
+-----------------+-----------+-----------+-----------+
*/  [2]

int __wcwidth_sdeckanji(
         wint_t wc,  [3]
         _LC_charmap_t *hdl )  [4]
 {
      if (wc == 0)
         return(0);  [5] if (wc <= 0x9f)
         return(1);  [6]
      else if ((wc >= 0x0100) && (wc <= 0x015d))
         return(1);  [7]
      else if ((wc >=0x015e) && (wc <= 0x303b))
         return(2);  [8]
      else if ((wc >=0x303c) && (wc <= 0x5f19))
         return(2);  [9]
      else if ((wc >=0x5f1a) && (wc <= 0x8df7))
         return(2);  [10]
         return(-1);  [11]
 }
```

[1] Includes header files that contain constants and structures required for this method.

[2] Describes the algorithm used to determine the required display width.

A character's display width is either 1 or 2 columns, depending on the character set to which a character belongs. Display width is different

from the size of the character in multibyte format; for example, triple-byte characters require 2 display columns and double-byte characters can require either 1 or 2 display columns.

3 Defines the `wc` variable that stores the wide character for which display width information is requested.

4 Points, through `hdl`, to a structure that stores pointers to the methods that parse character maps for this locale.

5 Returns zero (0) if the wide-character buffer is empty.

6 Returns 1 if the wide-character value is less than or equal to 0x009f.

7 Returns 1 if the wide-character value is in the range 0x0100 to 0x015d.

8 Returns 2 if the wide-character value is in the range 0x015e to 0x303b.

9 Returns 2 if the wide-character value is in the range 0x303c to 0x5f19.

10 Returns 2 if the wide-character value is in the range 0x5f1a to 0x8df7.

11 Returns –1 if the wide-character value is invalid.

The calling function, `wcwidth( )`, also returns –1 if the wide character is nonprintable; however, this condition is evaluated at the level of the calling function and does not need to be evaluated by the method.

6.3.2 Optional Methods

A locale can include optional methods in addition to the required methods discussed in Section 6.3.1. A method is considered optional if a default method is applied in the absence of a method specification. That is, if your locale uses methods but does not supply any methods for the functions associated with particular locale categories or some other locale-related functions, the `localedef` command applies default methods that handle process code for both single-byte and multibyte characters.

Writing optional methods requires detailed information about the internal interfaces to C Library routines. This information is vendor proprietary and may be subject to change. Thus, optional method descriptions in this section are less complete than the descriptions for required methods.

In the rare cases in which your locale must include an optional method, contact your technical support representative to request information.

The following list names the optional methods:

- `LC_CTYPE` category
  - `towupper`
  - `towlower`
  - `wctype`

  - iswctype
- LC_COLLATE category
  - fnmatch
  - strcoll
  - strxfrm
  - wcscoll
  - wcsxfrm
  - regcomp
  - regexec
  - regfree
  - regerror
- LC_MONETARY, LC_NUMERIC, or both categories
  - localeconv
  - strfmon
- LC_TIME category
  - strftime
  - strptime
  - wcsftime
- LC_MESSAGES category
  - rpmatch
- Miscellaneous use
  - nl_langinfo( )

6.3.3 Building a Shareable Library to Use with a Locale

Example 6–21 contains the compiler and linker command lines that are required to build the method source files into a shareable library that is used with the ja_JP.sdeckanji locale.

Example 6–21: Building a Library of Methods Used with the ja_JP.sdeckanji Locale

```
cc -std0 -c \
   __mblen_sdeckanji.c __mbstopcs_sdeckanji.c \
   __mbstowcs_sdeckanji.c __mbtopc_sdeckanji.c \
   __mbtowc_sdeckanji.c __pcstombs_sdeckanji.c \
   __pctomb_sdeckanji.c __wcstombs_sdeckanji.c \
   __wcswidth_sdeckanji.c __wctomb_sdeckanji.c \
   __wcwidth_sdeckanji.c
```

Example 6–21: Building a Library of Methods Used with the ja_JP.sdeckanji Locale (cont.)

```
ld -shared -set_version osf.1 -soname libsdeckanji.so -shared \
   -no_archive -o libsdeckanji.so \
   __mblen_sdeckanji.o __mbstopcs_sdeckanji.o \
   __mbstowcs_sdeckanji.o __mbtopc_sdeckanji.o \
   __mbtowc_sdeckanji.o __pcstombs_sdeckanji.o __pctomb_sdeckanji.o \
   __wcstombs_sdeckanji.o __wcswidth_sdeckanji.o __wctomb_sdeckanji.o \
   __wcwidth_sdeckanji.o \
   -lc
```

See `cc`(1) and `ld`(1) for more information about shared libraries.

6.3.4 Creating a methods File for a Locale

The `methods` file contains an entry for each function that is defined in the methods shared library for use with the locale. The operation performed by the function is identified by a method keyword, followed by quoted strings with the name of the function and the path to the shared library that contains the function.

Example 6–22 illustrates the section of a `methods` file for the methods used with the `ja_JP.sdeckanji` locale. Because you must define a list of required methods if you want to override any C Library interfaces, your `methods` file must always specify an entry for each required method as shown in this example. The `ja_JP.sdeckanji` locale relies on default implementations for all optional methods, and so the example does not contain entries for any of the optional methods.

Example 6–22: The methods File for the ja_JP.sdeckanji Locale

```
# sdeckanji.m  1
# <method_keyword> "<entry>" "<package>" "<library_path>"  1

METHODS   2

__mbstopcs "__mbstopcs_sdeckanji" "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
__mbtopc   "__mbtopc_sdeckanji"   "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
__pcstombs "__pcstombs_sdeckanji" "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
__pctomb   "__pctomb_sdeckanji"   "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
mblen      "__mblen_sdeckanji"    "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
mbstowcs   "__mbstowcs_sdeckanji" "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
mbtowc     "__mbtowc_sdeckanji"   "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
wcstombs   "__wcstombs_sdeckanji" "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
```

Example 6–22: The methods File for the ja_JP.sdeckanji Locale (cont.)

```
wcswidth   "__wcswidth_sdeckanji" "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
wctomb     "__wctomb_sdeckanji"   "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3
wcwidth    "__wcwidth_sdeckanji"  "libsdeckanji.so" \
"/usr/shlib/libsdeckanji.so" 3

END METHODS   4
```

1 Comment lines

These lines specify the name of the `methods` file and the format of method entries. The field identified in the format as `<package>` is ignored, but you must specify some string for this field in order to specify a library path.

2 Header to mark start of method entries

3 Entries for required methods

4 Trailer to mark end of method entries

See `localedef`(1) for detailed information about `methods` file entries.

6.4 Building and Testing the Locale

Use the `localedef` command to build a locale from its source files. Example 6–23 is the command line needed to build the French locale used in most examples in this chapter. Assume for this example that all source files reside in the user's default directory and that the resulting locale is also created in that directory.

Example 6–23: Building the fr_FR.ISO8859-1@example Locale

```
% localedef -f ISO8859-1.cmap \   1
-i fr_FR.ISO8859-1.src \  2
fr_FR.ISO8859-1@example  3
```

1 The `-f` option specifies the character map source file.

2 The `-i` option specifies the locale definition source file.

3 The final argument to the command is the name of the locale.

When you are testing locales, particularly ones that are similar to standard locales installed on the system, add an extension to the locale name. Varying names with the at (@) extension allows you to specify the standard strings for language, territory, and codeset and still be sure that the test locale is

uniquely identified. This is important if you later decide to move the locale to the /usr/lib/nls/loc directory, where other locales reside.

Example 6–23 contains only one form and a few options for the localedef command. See localedef(1) for a complete description of the command.

The following is a summary of some important rules and options:

- If you defined methods for your locale, you must specify the methods file with the -m option. For example, the command line that builds the ja_JP.sdeckanji locale would include –m sdeckanji.m to identify the file shown in Example 6–22.
- You can use the -v option to run the command in verbose mode for debugging purposes. This option, when used with the -c option, creates a .c file that contains useful information about the locale.
- Use the -w option if you want the command to display warnings when it encounters duplicate definitions.

By default, locales must reside in the /usr/lib/nls/loc directory to be found. If you want to test your locale before moving it to the /usr/lib/nls/loc directory, you can define the LOCPATH variable to specify the directory where your locale is located. You can then define the LANG environment variable to be your new locale and interactively test the locale with commands and applications.

Example 6–24 uses the date command to test the date/time format.

Example 6–24: Setting the LOCPATH Variable and Testing a Locale

```
% setenv LOCPATH ~harry/locales
% setenv LANG fr_FR.ISO8859-1@example
% date
ven 23 avr 13:43:05 EDT 1999
```

Note

The LOCPATH variable is an extension to specifications in the X/Open UNIX standard and therefore may not be recognized on all systems that conform to this standard.

Some programs have support files that are installed in system directories with names that exactly match the names of standard locales. In such cases, application software, system software, or both might use the value of the LANG environment variable to determine the locale-specific directory in which the support files reside. If assigned directly to the LANG or LC_ALL

environment variable, locale file names with an at (@) suffix may result in invalid search paths for some applications.

The following example illustrates how you can work around this problem by assigning the standard locale name to the LANG variable and the name of your variant locale to the locale category variables. You need to make assignments only to those category variables that represent areas where your locale differs from the locale on which it is based.

```
% setenv LANG fr_FR.ISO8859-1
% setenv LC_CTYPE fr_FR.ISO8859-1@example
% setenv LC_COLLATE fr_FR.ISO8859-1@example
⋮

% setenv LC_TIME fr_FR.ISO8859-1@example
```

7

Programming Considerations for International Applications

This chapter describes a set of miscellaneous tasks you should consider as you develop international applications. These tasks include the following:

- Choosing an input method and input styles (Section 7.1)
- Managing user-defined character databases (Section 7.2)
- Assigning a sort order with a locale specification (Section 7.3)
- Processing non-English language reference pages (Section 7.4)
- Converting data files from one codeset to another (Section 7.5)
- Using font renderers in Chinese and Korean language PostScript Support (Section 7.6)

This chapter provides information on the tools needed to create international applications. The information in this chapter is also closely related to how international applications are used on the operating system. As you use the information in this chapter, you may also find it helpful to refer to the companion manual, *Using International Software*.

The following manuals provide language-specific information about customization and software use provided for Asian languages on the operating system:

- *Technical Reference for Using Chinese Features*
- *Technical Reference for Using Japanese Features*
- *Technical Reference for Using Korean Features*
- *Technical Reference for Using Thai Features*

These manuals are available from the programming bookshelf of the operating system documentation Web site (**http://www.tru64unix.compaq.com/docs/**). Non-English language characters are embedded in the text of the Chinese, Japanese, and Korean Technical References. To view these characters with your Web browser, you must install the appropriate language support subsets on your system and set your locale to one that includes the local language characters used in the technical reference.

The operating system documentation also provides introductory reference pages on the topics of internationalization (`i18n_intro`(5)) and localization (`l10n_intro`(5)) as well as reference pages for all supported languages and codesets.

7.1 Choosing an Input Method

For some languages, such as Japanese, Chinese, and Korean, you use an input method to enter characters and phrases. An input method lets you enter a character by taking multiple editing actions on entry data. The data entered at intermediate stages of character entry is called the preediting string.

The X Input Method specification defines the following user input, or preediting, styles:

- On-the-Spot

  Data being edited is displayed directly in the application window. Application data is moved to allow the preediting string to display at the point of character insertion.

- Over-the-Spot

  The preediting string is displayed in a window that is positioned over the point of insertion.

- Off-the-Spot

  The preediting string is displayed in a window that is within the application window but not over the point of insertion. Often, the window for the preediting string appears at the bottom of the application window. In this case, the preediting window may block the last line of text from view in the application window. You can resize the application window to make this last line visible.

- Root Window

  The preediting string is displayed in a child window of the application root window.

Input methods for different locales typically support more than one user input style but not all of them. If you work in languages that are supported by an input method, you can specify styles in priority order through the VendorShell resource `XmNpreeditType`. By default, this resource is defined to be the following:

```
OnTheSpot,OverTheSpot,OffTheSpot,Root
```

The priority order of these values means that On-the-Spot input style is used if the input method supports it, else the Over-the-Spot is used if the input method supports it, and so forth.

Use one of the following methods to supply the XmNpreeditType resource value to an application:

- In CDE, use the Input Methods application. See the *CDE Companion* manual for information on using this application.
- In an application-specific resource file.
- On the command line that invokes an application.

  For example:

  ```
  % app-name -xrm '*preeditType: offthespot,onthespot' &
  ```

Input styles are supported by specialized input method servers. An input method server runs as an independent process and communicates with an application to handle input operations.

An input method server does not have to be running on the same system as the application but, with one exception, it must be running and made accessible to the application before the application starts.

If a Motif application that has been internationalized to support simplified Chinese contains an XmText or XmTextField widget with the Reconnectable resource set to True, the application is able to establish a connection with the input server when the application starts first or when the server stops and restarts. See XmText(3X) and XmTextField(3X) for more information.

See the *Using International Software* manual for information on the input method servers available on the operating system and the input styles that each server supports.

7.2 Managing User-Defined Characters and Phrase Input

The national character sets for Japan, Taiwan, and China do not include some of the characters that can appear in Asian place names and personal names. Such characters are defined by users and reside in site-specific databases. These databases are called user-defined character (UDC) or character-attribute databases. When users define ideographic characters, they must also define font glyphs, collating files, and other support files for the characters.

Appendix B provides details on how you set up and use UDC databases.

In Korea, Taiwan, and China, users can enter a complete phrase by typing a keyword, abbreviation, or acronym. This capability is supported by a phrase database and an input mechanism. The *Using International Software* manual provides details on how the user sets up and uses a phrase database.

The /var/i18n/conf/cp_dirs configuration file allows software services or hardware to locate the databases that support UDC and phrase input.

Example 7–1 contains the default entries in the `cp_dirs` file. You can edit these entries to change the default locations.

Example 7–1: Default cp_dirs File

```
#
# Attribute directory configuration file
#
#                       System location         User location
#                       ===============         =============
udc     -               /var/i18n/udc           ~/.udc
odl     -               /var/i18n/odl           ~/.odl
sim     -               /var/i18n/sim           ~/.sim
cdb     /usr/i18n/.cdb  /var/i18n/cdb           ~/.cdb
iks     -               /var/i18n/iks           ~/.iks
pre     -               /var/i18n/fonts         ~/.fonts
bdf     -               /var/i18n/fonts         ~/.fonts
pcf     -               /var/i18n/fonts         ~/.fonts
```

Each line in the `cp_dirs` file represents one entry and has the following format:

[*service_name standard_path system_path user_path*]

The *service_name* can be one of the following:

- `bdf` (for font files in BDF format)
- `cdb` (for collating value databases used with the `asort` command)
- `iks` (for input key sequence files)
- `odl` (for databases of fonts and input key sequences that the SoftODL service uses)
- `pcf` (for font files in Printer Customization File format)

  These files, depending on their font resolution, reside in either the `75dpi` or `100dpi` subdirectory.
- `pre` (for font files in preload format created by the `cgen` utility)

  These are raw font files used to preload multibyte character terminals.
- `sim` (for phrase databases)
- `udc` (for UDC databases)

The `cp_dirs` file can contain only one entry for each service named. Remaining fields in the entry line consist of the following:

- *standard_path* specifies the location of the collating values database for the standard character sets (applies only to the `cdb` entry)
- *system_path* specifies the location of systemwide databases
- *user_path* specifies the location of users' private databases

The preceding locations are specified as one of the following:

- An absolute pathname, starting with a slash (/)
- A pathname, starting with tilde slash (~/), that is relative to a user's home directory
- A minus sign or hyphen (–) to indicate that the entry is not used

  For example, you can specify - to be *user_path* for all services related to user-defined characters if you want these characters supported only through systemwide databases.

Comment lines in the cp_dirs file begin with the number sign (#).

7.3 Assigning a Sort Order with a Locale Specification

The sort command sorts characters according to the collation sequence defined for the current locale. A particular locale can apply one set of collation rules to the associated character set. Multiple locale names do exist, however, for the same combination of language, territory, and character set. These variations offer users the choice of more than one collating sequence.

When more than one locale is available for a given combination of language, territory, and codeset, some of the locale names include a suffix with the format @*variant*. To avoid problems with pathnames constructed using the %L specifier, you should assign a locale name with a suffix that is category specific only to the appropriate locale category variable (or variables). In the following example, the locale assigned to LC_COLLATE differs from the locale assigned to LANG only with respect to collating sequence:

```
% setenv LANG zh_TW.eucTW
% setenv LC_COLLATE zh_TW.eucTW@radical
```

Supporting different collation orders through one or more locales is adequate for most languages. However, collation orders for Asian languages require additional support for the following reasons:

- Asian languages include UDCs, which are not specified in a locale. These characters can be defined with a collation weight. In this case, the collation weight needs to be applied when the UDCs are encountered in the strings being sorted.
- Ideographic characters can be sorted on more than one dimension (radical, stroke, phonetic, and internal code). Some users need to combine these dimensions during sort operations. In one operation the user may need to sort characters first by radical and then according to the number of strokes. For another operation, the user may need to put characters first in phonetic order, then according to the number of strokes, and so on. Sorting by combinations of dimensions requires breadth-first sorting, rather than the depth-first sorting implemented through locales.

For the preceding reasons, the asort command was developed and is available when you install language variant subsets that support Asian languages. The asort command uses, by default, the collating order defined for the LC_COLLATE variable and supports all the options supported by the sort command. In addition, the asort command includes the following options:

- -C

  This option indicates that the sort operation should use special system sort tables, along with sort tables produced by the cgen utility, to support UDCs. This option overrides the sort sequence defined in the locale specified by the LC_COLLATE variable.

- -v

  This option, which you can use only with the -C option, implements breadth-first sorting.

See asort(1) for more information about using this command.

7.4 Processing Non-English Language Reference Pages

Programmers who supply software applications for UNIX systems frequently supply online reference pages (manpages) to document the application and its components. UNIX text-processing commands and utilities must be able to process translated versions of these reference pages for applications sold to the international market. The operating system includes enhanced versions of the nroff, tbl, and man commands to support this requirement.

7.4.1 The nroff Command

The nroff command includes the following functions to support locales:

- Formats reference page source files written in any language whose locale is installed on the system.
- Supports characters of any supported languages in the string arguments of macros and requests.
- Supports character mapping of characters for any supported language through the .tr request in reference page source files.
- Allows you to set the escape character (\), command control character (.), and nobreak control character (') to local language, as well as ASCII, characters.
- Maps each 2-byte space character, which is defined in most codesets for Asian languages, to two ASCII spaces in output.

When formatting reference pages that contain ideographic characters, the nroff command treats each character as a single word. A string of

ideographic characters, including 2-byte letters and punctuation characters, can be wrapped to the next line subject to the following constraints:

- The last character on the text line cannot be defined as a no-last character by either the standard or private list of no-last characters.
- The first character on the text line cannot be defined as a no-first character by either the standard or private list of no-first characters.

The standard no-first, no-last character lists are defined in `nroff` catalog files. For lists of these characters, see the following language-specific manuals:

- *Technical Reference for Using Chinese Features*
- *Technical Reference for Using Japanese Features*
- *Technical Reference for Using Korean Features*
- *Technical Reference for Using Thai Features*

These manuals are available from the programming bookshelf of the operating system documentation Web site (**`http://www.tru64unix.compaq.com/docs/`**).

The no-first and no-last constraints exist to prevent `nroff` from placing a punctuation mark or right parenthesis at the beginning of a text line or placing a left parenthesis at the end of a text line. You can turn the standard constraints on and off in source files with the `.ki` and `.ko` commands, respectively.

You can also define a private set of no-first and no-last characters with the following command:

`.kl '`*`no-first-list`*`'`*`no-last-list`*`'`

The parameters *`no-first-list`* and *`no-last-list`* are strings of characters that you include in the no-first and no-last categories. You cancel a private no-first and no-last list by entering a `.kl` command with null strings as the parameters. For example:

```
.kl '''
```

Note

The characters specified in the `.kl` command override, rather than supplement, the characters in the standard set of no-first and no-last characters. Therefore, you cannot use the standard set of no-first and no-last characters together with a private set.

Using the command `.kl '''` restores use of the standard set of no-first and no-last characters for the current locale.

The nroff command can format text so that it is justified or not justified to the right margin. When text is justified to the right margin, nroff inserts spaces between words in the line. Ideographic characters, although treated as words in most stages of the formatting process, differ in terms of whether they can be delimited by spaces. The characters that can be preceded by a space, followed by a space, or both are listed in the language-specific user manuals that are available on line when you install language variant subsets of the operating system. When right-justifying text, the nroff command inserts spaces only at the following places:

- Where 1-byte or 2-byte spaces already occur
- Between English language characters and ideographic characters
- Before characters defined as can-space-before
- After characters defined as can-space-after

In other cases, no space is inserted between consecutive ideographic characters. Therefore, if a text line contains only ideographic characters, it may not be justified to the right margin.

7.4.2 The tbl Command

The tbl command preprocesses table formatting commands within blocks delimited by the .TS and .TE macros. The tbl command handles multibyte characters that can occur in text of languages other than English.

The tbl command is frequently used with the neqn equation formatting preprocessor to filter input passed to the nroff command. In such cases, specify tbl first to minimize the volume of data passed through the pipes. For example:

```
% cd /usr/usr/share/ja_JP.deckanji/man/man1
% tbl od.1 | neqn | nroff -Tlpr -man -h | \
lpr -Pmyprinter
```

When printing Asian language text, you must use printer hardware that supports the language.

7.4.3 The man Command

The man command can handle multibyte characters in reference page files. By default, the man command automatically searches for reference pages in the/usr/share/*locale_name*/man directory before searching the /usr/share/man and /usr/local/man directories. Therefore, if the LANG environment variable is set to an installed locale and if reference page translations are available for that locale, the man command automatically displays reference pages in the appropriate language.

In addition, the man command automatically applies codeset conversion (assuming the availability of appropriate converters) when reference page translations for a particular language are encoded in a codeset that does not match the codeset of the user's locale. See man(1) for information about redefining the man command search path and for more details about codeset conversion.

7.5 Converting Data Files from One Codeset to Another

Each locale is based on a specific codeset. Therefore, when an application uses a file whose data is coded in one codeset and runs in a locale based on another codeset, character interpretation may be meaningless. For example, assume that a fictional language includes a character named "quo," which is encoded as \031 in one codeset and \042 in another codeset. If the "quo" character is stored in a data file as \031, the application that reads data from that file should be running in the locale based on the same codeset. Otherwise, \031 identifies a character other than "quo."

Users, the applications they run, or both may need to set the process environment to a particular locale and use a data file created with a codeset different from the one on which the locale is based. The data file in question might be appropriate for a given language and in a codeset different from the user's locale for one of the following reasons:

- The data file might have been created on another vendor's system by using a locale based on a vendor-specific codeset. For example, the integration of PCs into the enterprise computing environment increases the likelihood that UNIX users need to process files for which the data encoding is in MS-DOS code page format.
- The locale could be one of several UNIX locales that support the same Asian language, such as Japanese. Asian languages are typically supported by a variety of locales, each based on a different codeset.
- The data file could be in Unicode, UCS-4, UTF-8, UTF-16, or UTF-32 format. If characters in this file are to be printed or displayed on the screen, they might need to be converted to encodings for which fonts are available.

You can convert a data file from one codeset to another by using the iconv command or the iconv_open( ), iconv( ), and iconv_close( ) functions. For example, the following command reads data in the accounts_local file, which is encoded in the SJIS codeset; converts the data to the eucJP codeset; and appends the results to the accounts_central file:

```
% iconv -f SJIS -t eucJP accounts_local \
>> accounts_central
```

Many commands and utilities, such as the man command and internationalized print filters, use the iconv( ) functions and associated converters to perform codeset conversion on the user's behalf.

The iconv command and associated functions can use either an algorithmic converter or a table converter to convert data. Algorithmic converters, if installed on your system, reside in the /usr/lib/nls/loc/iconv directory; this directory is the one searched first for a converter. This directory also contains an alias file (iconv.alias) that maps different name strings for the same converter to the converter as named on the system. Table converters, if installed on your system, reside in the /usr/lib/nls/loc/iconvTable directory. The value of the LOCPATH variable, if defined, overrides the command's default search path.

The iconv command assumes that a converter name uses the following format:

from-codeset_ to-codeset

For the preceding example, the iconv command would search for and use the /usr/lib/nls/loc/iconv/SJIS_eucJP converter.

Also consider operating system support for codeset conversion of the Hong Kong Supplementary Character Set (HKSCS). HKSCS is not a locale or character set name, but is used to provide a common language interface for electronic communication and data exchange conducted in Chinese. The characters in HKSCS are only for computer use. On Tru64 UNIX, HKSCS is used as the name for extended Big-5 encoding that contains HKSCS characters, and support is limited to code conversion between HKSCS and Unicode. Using the iconv command, codeset conversion with HKSCS would be specified as one of the following:

- UTF-16_HKSCS or HKSCS_UTF-16
- UCS-4_HKSCS or HKSCS_UCS-4
- UTF-8_HKSCS or HKSCS_UTF-8

See HKSCS(5) for more information on the Hong Kong Supplementary Character Set.

Table 7–1 specifies the codeset conversions that the operating system supports for English language data. Tables with codeset conversions supported for their respective Asian languages are described in the following manuals:

- *Technical Reference for Using Chinese Features*
- *Technical Reference for Using Japanese Features*
- *Technical Reference for Using Korean Features*

- *Technical Reference for Using Thai Features*

For detailed information about the `iconv` command, see `iconv`(3) and `iconv_intro`(5). For information on functions that programs can use to perform codeset conversion, see `iconv_open`(3), `iconv`(1), and `iconv_close`(3). You can find a list of all the codeset converters available for a particular language in the reference page for that language.

Table 7–1: Supported Codeset Conversions for English

Codeset	ASCII-GR	ISO8859-1	ISO8859-1-GL	ISO8859-1-GR
ASCII-GR	–	Yes	No	No
ISO8859-1	Yes	–	Yes	Yes
ISO8859-1-GL	No	Yes	–	No
ISO8859-1-GR	No	Yes	No	–

7.6 Using Font Renderers in Chinese and Korean PostScript Support

This section describes the use of font renderers in the creation of Motif applications that support PostScript fonts in Chinese and Korean. See the *Using International Software* manual for information on tuning cache size for ideographic characters and customizing windows for local languages.

7.6.1 Using Font Renderers for Multibyte PostScript Fonts

The operating sytem includes font renderers that allow any X application to use the PostScript fonts available for the Chinese and Korean languages. The system administrator can set up font renderers for the following kinds of fonts for use through the X Server or the font server:

- Double-Byte PostScript outline fonts
- UDC fonts

By installing the `IOSWWXFR**` subset, you automatically enable font rendering for the PostScript outline fonts.

7.6.1.1 Setting Up the Font Renderer for Double-Byte PostScript Fonts

You can set up the font renderer for Chinese and Korean PostScript fonts for use either through the X server or the font server by editing the appropriate configuration file.

- For the X server, the font renderer is automatically added at installation time to the `font_renderers` list in the X server's configuration file.

- For a font server, you must manually add the following entry to the `renderers` list in the font server's configuration file:

  ```
  renderers = other_renderer, other_renderer,...
       libfr_DECpscf.so;DECpscfRegisterFontFileFunctions
  ```

  In addition, you must specify the paths for the PostScript font files in the `catalogue` list in the same configuration file. Double-Byte PostScript fonts for the Asian languages are available in the following directories:

  ```
  /usr/i18n/lib/X11/fonts/KoreanPS
  /usr/i18n/lib/X11/fonts/SChinesePS
  /usr/i18n/lib/X11/fonts/TChinesePS
  ```

  Each font in these directories has the following components:

  – A Type1 font header with the `.pfa2` file name extension

    This header file is the only file that must be listed in the `fonts.dir` file in the font directory.

  – A data file with the `.csdata` file name extension

  – A binary metrics file with the `.xafm` file name extension

The renderer for Asian Double-Byte PostScript fonts uses its own configuration file that specifies the following information:

- Cache size (number of cache units)
- Cache unit size
- File handler (names associated with font-rendering software)
- Default character (character that is printed in place of any character for which there is no glyph)

The default pathname for this configuration file is `/var/X11/renderer/DECpscf_config`; however, you can change this path by setting the `DECPSCF_CONFIG_PATH` environment variable.

7.6.1.2 Setting Up the Font Renderer for UDC Fonts

The UDC font renderer accesses the UDC database directly to obtain font glyphs. Therefore, X applications that use this renderer do not need to use `.pcf` files generated by the `cgen` utility.

You can set up the UDC font renderer for use either through the X server or the font server as follows:

- For the X server, the font renderer is automatically added at installation time to the `font_renderers` list in the X server's configuration file.
- For a font server, you must manually add the following entry to the `renderers` list in the font server's configuration file:

```
renderers = other_renderer, other_renderer,...
     libfr_UDC.so;UDCRegisterFontFileFunctions
```

In addition, you must specify the path to the UDC database in the `catalogue` list of the same configuration file. This path should be set to the top directory for the UDC database. For example, `/var/i18n/udc` is the correct path for a systemwide UDC database if the database was set up in the default directory.

To process UDC characters in a particular language, the font renderer also requires entries in the `fonts.dir` file in the appropriate PostScript font directory from the following list:

```
/usr/i18n/lib/X11/fonts/SChinesePS
/usr/i18n/lib/X11/fonts/TChinesePS
```

Edit the `fonts.dir` file to specify virtual file names in the format *`locale_name`*`.udc` followed by the corresponding XLFD names registered for the codesets. Table 7–2 describes the XLFD entry that corresponds to different Asian codesets.

Table 7–2: XLFD Registry Names for UDC Characters

Codeset	XLFD Registry Name
`dechanyu`, `eucTW`	`DEC.CNS11643.1986-UDC`
`big5`	`BIG5-UDC`
`dechanzi`	`GB2312.1980-UDC`
`deckanji`, `sdeckanji`, `eucJP`	`JISX.UDC-1`

The following example entry is appropriate for the `fonts.dir` file in the `/usr/i18n/lib/X11/fonts/TChinesePS` directory:

```
2
zh_TW.dechanyu.udc -system-decwin-normal-r--24-240-75-75-m-24-DEC.CNS11643.1986-UDC
zh_TW.big5.udc -system-decwin-normal-r--24-240-75-75-m-24-BIG5-UDC
```

7.6.1.3 Using the Font Renderer for TrueType Fonts

The operating system includes a font renderer (`/usr/shlib/X11/libfr_TrueType.so`) that enables the use of TrueType fonts. Currently, the operating system includes TrueType fonts only for simplified Chinese. However, you can configure the font renderer to use third-party TrueType fonts for additional languages if these are required by applications used at your site. See `TrueType`(5X) for more information.

A

Summary Tables of Worldwide Portability Interfaces

This appendix lists and summarizes worldwide portability interfaces (WPI) that are defined by Version 5 of the X/Open CAE specification for system interfaces and headers (XSH). All these interfaces support the wide-character data type. Tables in this appendix also list older ISO C functions that use the `char` data type and therefore cannot perform character-by-character processing in all languages. The reference pages (manpages) provide detailed information for each interface. See `standards`(5) for information about compiling a program in the appropriate definition environment for XSH Version 5.

A.1 Locale Announcement

Programs call the following function to use the appropriate locale (language, territory, and codeset) at run time:

WPI Function	Description
`setlocale( )`	Establishes localization data at run time.

A.2 Character Classification

The following character classification functions classify values according to the codeset defined in the locale category `LC_CTYPE`.

WPI Function	Older ISO C Function	Description
`iswalnum( )`	`isalnum( )`	Tests if a character is alphanumeric.
`iswalpha( )`	`isalpha( )`	Tests if a character is alphabetic.
`iswcntrl( )`	`iscntrl( )`	Tests if a character is a control character.
`iswdigit( )`	`isdigit( )`	Tests if a character is a decimal digit in the portable character set.
`iswgraph( )`	`isgraph( )`	Tests if a character is a graphic character.
`iswlower( )`	`islower( )`	Tests if a character is lowercase.
`iswprint( )`	`isprint( )`	Tests if a character is a printing character.

WPI Function	Older ISO C Function	Description
`iswpunct( )`	`ispunct( )`	Tests if a character is a punctuation mark.
`iswspace( )`	`isspace( )`	Tests if a character determines white space in displayed text.
`iswupper( )`	`isupper( )`	Tests if a character is uppercase.
`iswxdigit( )`	`isxdigit( )`	Tests if a character is a hexadecimal digit in the portable character set.

In addition to the functions for each character classification, the WPI includes the following functions that provide a common interface to all classification categories:

- `wctype( )`

  This function returns a value that corresponds to a character classification.

- `iswctype( )`

  This function tests if a character has a certain property.

The WPI functions listed in the preceding table can be replaced by calls to the `wctype( )` and `iswctype( )` functions described in the following table:

Call Using Classification Function	Equivalent Call Using wctype() and iswctype()
`iswalnum(`*wc* `)`	`iswctype(`*wc* `, wctype("alnum"))`
`iswalpha(`*wc* `)`	`iswctype(`*wc* `, wctype("alpha"))`
`iswcntrl(`*wc* `)`	`iswctype(`*wc* `, wctype("cntrl"))`
`iswdigit(`*wc* `)`	`iswctype(`*wc* `, wctype("digit"))`
`iswgraph(`*wc* `)`	`iswctype(`*wc* `, wctype("graph"))`
`iswlower(`*wc* `)`	`iswctype(`*wc* `, wctype("lower"))`
`iswprint(`*wc* `)`	`iswctype(`*wc* `, wctype("print"))`
`iswpunct(`*wc* `)`	`iswctype(`*wc* `, wctype("punct"))`
`iswspace(`*wc* `)`	`iswctype(`*wc* `, wctype("space"))`
`iswupper(`*wc* `)`	`iswctype(`*wc* `, wctype("upper"))`
`iswxdigit(`*wc* `)`	`iswctype(`*wc* `, wctype("xdigit"))`

In this table, the quoted literals in the call to `wctype( )` are the character classes defined in the X/Open UNIX standard for Western European and many Eastern European languages; however, a locale can define other character classes. The Unicode standard defines character classes that do not have class-specific functions, and a locale for an Asian language might

define additional character classes to distinguish ideographic from phonetic characters. You must use the `wctype( )` and `iswctype( )` functions to test if a character belongs to a class when no class-specific function exists for the test. See `locale`(4) for details about character classes and testing equivalence between classes defined in the XSH and the Unicode standards.

Also, the input value for `isw*( )` functions must be in the range of wide characters defined for the current locale. If the input value is outside that range, the result is undefined. For more information, see `iswctype`(3).

Note

The `wctype( )` calls in the second column of the preceding table illustrate only functional equivalence to the `iswctype( )` calls in the first column of the table. In most programming applications, `iswctype( )` needs to execute multiple times for each execution of `wctype( )`. In such cases, you would code calls in the second column of the table as follows to achieve performance equivalence to corresponding calls in the first column:

```
wctype_t    property_handle;
wint_t      wc;
int         yes_or_no;
 .
 .
 .
    property_handle=wctype("alnum");
 .
 .
 .
    while (...) {
      .
      .
      .
    yes_or_no=iswctype(wc, property_handle);
      .
      .
      .
    }
```

A.3 Case and Generic Property Conversion

Use the following case conversion functions to switch the case of a character according to the codeset defined in the locale category `LC_CTYPE`:

WPI Function	Older ISO C Function	Description
`towlower( )`	`tolower( )`	Converts a character to lowercase.
`towupper( )`	`toupper( )`	Converts a character to uppercase.

The WPI also includes the following functions to map and convert a character according to properties defined in the current locale:

- `wctrans( )`

  This function maps a character to a property defined in the current locale.

- `towctrans( )`

  This function converts a character according to a property defined in the current locale.

Currently, the only properties defined in operating system locales are `toupper` and `tolower`. The following example of using `wctrans( )` and `towctrans( )` performs the same conversion as `towupper( )`:

```
wint_t     from_wc, to_wc;
wctrans_t  conv_handle;
.
.
.
       conv_handle=wctrans("toupper");
.
.
.
       while (...) {
         .
         .
         .
       to_wc=towctrans(from_wc,conv_handle);
         .
         .
         .
       }
```

A.4 Character Collation

The function in the following table sort strings according to rules specified in the locale defined for the `LC_COLLATE` category:

WPI Function	Older ISO C Function	Description
`wcscoll( )`	`strcoll( )`	Collates character strings.

You can also use the `wcsxfrm( )` and `wcscmp( )` functions, summarized in Section A.11, to transform and then compare wide-character strings.

A.5 Access to Data That Varies According to Language and Custom

The functions in the following table allow programs to retrieve data that is language specific or country specific, as specified by the locale setting:

WPI Function	Description
`nl_langinfo( )`	A general-purpose function that retrieves language and cultural data according to the locale setting.
`strfmon( )`	Formats a monetary value according to the locale setting.
`localeconv( )`	Returns information used to format numeric values according to the locale setting.

A.6 Conversion and Format of Date/Time Values

The `ctime( )` and `asctime( )` functions do not have the flexibility needed for language independence. The WPI therefore includes the following interfaces to format date and time strings according to information provided by the locale:

WPI Function	Description
`strftime( )`	Formats a date and time string based on the specified format string and according to the locale setting.
`wcsftime( )`	Formats a date and time string based on a specified format string and according to the locale setting, then returns the result in a wide-character array.
`strptime( )`	Converts a character string to a time value according to a specified format string; reverses the operation performed by `strftime( )`.

A.7 Printing and Scanning Text

The WPI extends definitions of the following ISO C functions to support internationalization requirements. The WPI extensions are described after the table that lists the functions.

WPI/ISO C Function	Description
`fprintf( )`	Prints formatted output to a file by using a `vararg` parameter list.
`fwprintf( )`	Prints formatted wide characters to the specified output stream by using a `vararg` parameter list.
`printf( )`	Prints formatted output to the standard output stream by using a `vararg` parameter list.
`sprintf( )`	Formats one or more values and writes the output to a character string by using a `vararg` parameter list.
`swprintf( )`	Prints formatted wide characters to the specified address by using a `vararg` parameter list.
`vfprintf( )`	Prints formatted output to a file by using a `stdarg` parameter list.
`vfwprintf( )`	Prints formatted wide characters to the specified output stream by using a `stdarg` parameter list.
`vprintf( )`	Prints formatted output to the standard output stream by using a `stdarg` parameter list.
`vsprintf( )`	Formats a `stdarg` parameter list and writes the output to a character string.
`vswprintf( )`	Prints formatted output to the specified address by using a `stdarg` parameter list.
`vwprintf( )`	Prints formatted wide characters to the standard output by using a `stdarg` parameter list.
`wprintf( )`	Prints formatted wide characters to the standard output by using a `vararg` parameter list.
`fscanf( )`	Converts formatted input from a file.
`fwscanf( )`	Converts formatted wide characters from the specified output stream.
`scanf( )`	Converts formatted input from the standard input stream.
`sscanf( )`	Converts formatted data from a character string.
`swscanf( )`	Converts formatted wide characters from the specified address.
`wscanf( )`	Converts formatted wide characters from the standard input.

The WPI extensions to the preceding functions include the following:

- `%`*`digit`*`$` conversion specifier, which allows variation in the ordinal position of the argument being printed. Such variation is frequently necessary when text is translated into different languages.

- Use of the decimal-point character as specified by the locale. This feature affects `e`, `E`, `f`, `g`, and `G` conversions.
- Use of the thousands-grouping character specified by the locale.
- The `C` and `S` conversion characters, which let you convert wide characters and wide-character strings, respectively.

A.8 Number Conversion

Functions in the following table convert strings to various numeric formats:

WPI Function	Older ISO C Function	Description
`wcstod( )`	`strtod( )`	Converts the initial portion of a string to a double-precision floating-point number.
`wcstol( )`	`strtol( )`	Converts the initial portion of a string to a long integer number.
`wcstoul( )`	`strtoul( )`	Converts the initial portion of a string to an unsigned long integer number.

A.9 Conversion of Multibyte and Wide-Character Values

To allow an application to get data from or write data to external files (as multibyte data) and process it internally (as wide-character data), the WPI defines the functions listed in the following table to convert between multibyte data and wide-character data:

WPI Function	Description
`btowc( )`	Converts a single byte from multibyte character format to wide-character format.

`mblen( )`	Determines the number of bytes in a character according to the locale setting. You should modify all string manipulation statements, which assume the size of a character is always 1 byte, to call this function. The following statement updates a pointer to the next character, `cp`:
	`cp++;`
	The following example incorporates the `mblen( )` function to ensure language-independent operation at run time; the `MB_CUR_MAX` variable is defined by the locale to be the maximum number of bytes that any character can occupy:
	`cp += mblen(cp, MB_CUR_MAX);`

`mbrlen( )`	Performs the same operation as `mblen( )` but can be restarted for use with locales that include shift-state encoding. [a]

WPI Function	Description
`mbrtowc( )`	Performs the same operation as `mbtowc( )` but can be restarted for use with locales that include shift-state encoding. [a]
`mbsrtowcs( )`	Performs the same operation as `mbstowcs( )` but can be restarted for use with locales that include shift-state encoding. [a]
`mbstowcs( )`	Converts a multibyte character string to a wide-character string.
`mbtowc( )`	Converts a multibyte character to a wide character.
`wcstombs( )`	Converts a wide-character string to a multibyte character string.
`wcrtomb( )`	Performs the same operation as `wctomb( )` but can be restarted for use with locales that include shift-state encoding. [a]
`wcsrtombs( )`	Performs the same operation as `wcstombs( )` but can be restarted for use with locales that include shift-state encoding. [a]
`wctob( )`	Converts a wide character to a single byte in multibyte character format, if possible.
`wctomb( )`	Converts a wide character to a multibyte character.

[a] The operating system does not currently provide locales that use shift-state encoding.

Note

You do not always need to explicitly handle the conversion to and from file code (multibyte data). Functions for printing and scanning text (discussed in Section A.7) include the `%S` and `%C` format specifiers that automatically handle multibyte to wide-character conversion. The WPI alternatives for older ISO C input/output functions (see Section A.10) also perform multibyte/wide-character conversions automatically.

A.10 Input and Output

The WPI functions listed in the following table automatically convert between file code (usually multibyte encoding) and process code (wide-character encoding) for text input and output operations:

WPI Function	Older ISO C Function	Description
`fgetwc( )`	`fgetc( )`	Gets a character from an input stream and advances the file position pointer.
`fgetws( )`	`fgets( )`	Gets a string from an input stream.
`fputwc( )`	`fputc( )`	Writes a character to an output stream.
`fputws( )`	`fputs( )`	Writes a string to an output stream.

WPI Function	Older ISO C Function	Description
`fwide( )`	None	Sets stream orientation to byte or wide character. This function is not useful within current locale environments. [a]
`getwc( )`	`getc( )`	Gets a character from an input stream.
`getwchar( )`	`getchar( )`	Gets a character from the standard input stream.
None	`gets( )`	Use `fgetws( )`.
`mbsinit( )`	None	Determines, for locales that use shift-state encoding, whether a multibyte string is in the initial conversion state. [a]
`putwc( )`	`putc( )`	Writes a character to an output stream.
`putwchar( )`	`getchar( )`	Writes a character to the standard output stream.
None	`puts( )`	Use `fputws( )`.
`ungetwc( )`	`ungetc( )`	Pushes a character back onto an input stream.

[a] The operating system does not currently provide locales that use shift-state encoding.

A.11 String Handling

The WPI defines alternatives and additions to ISO C byte-oriented functions to support manipulation of character strings. The WPI functions support both single-byte and multibyte characters.

String Concatenation:

WPI Function	Older ISO C Function	Description
`wcscat( )`	`strcat( )`	Appends a copy of a string to the end of another string.
`wcsncat( )`	`strncat( )`	Similar to the functions of `wcscat( )`, except that the number of characters to be appended is limited by the *n* parameter.

String Searching:

WPI Function	Older ISO C Function	Description
`wcschr( )`	`strchr( )`	Locates the first occurrence of a character in a string.

WPI Function	Older ISO C Function	Description
wcsrchr( )	strrchr( )	Locates the last occurrence of a character in a string.
wcspbrk( )	strpbrk( )	Locates the first occurrence of any characters from one string in another string.
wcsstr( )	strstr( )	Finds a substring. The wcsstr( ) function also supercedes the wcswcs( ) function included in versions of the XSH specification earlier than Issue 5.
wcscspn( )	strcspn( )	Returns the number of initial elements of one string that are all characters not included in a second string.
wcsspn( )	strspn( )	Returns the number of initial elements of one string that are all characters included in a second string.

String Copying:

WPI Function	Older ISO C Function	Description
wcscpy( )	strcpy( )	Copies a string.
wcsncpy( )	strncpy( )	Similar to the strcpy( ) function, except that the number of characters to be copied is limited by the *n* parameter.

String Comparison:

WPI Function	Older ISO C Function	Description
wcscmp( )	strcmp( )	Compares two strings.
wcsncmp( )	strncmp( )	Similar to the strcmp( ) function, except that the number of characters to be compared is limited by the *n* parameter.

String Length Determination:

WPI Function	Older ISO C Function	Description
wcslen( )	strlen( )	Determines the number of characters in a string.

String Decomposition:

WPI Function	Older ISO C Function	Description
`wcstok( )`	`strtok( )`	Decomposes a string into a series of tokens, each delimited by a character from another string.

Printing Position Determination:

WPI Function	Older ISO C Function	Description
`wcswidth( )`	None	Determines the number of printing positions required for a number of characters in a string.
`wcwidth( )`	None	Determines the number of printing positions required for a character.

Performing Memory Operations on Strings:

WPI Function	Older ISO C Function	Description
`wmemcpy( )`	`memcpy( )`	Copies characters from one buffer to another.
`wmemchr( )`	`memchr( )`	Searches a buffer for the specified character.
`wmemcmp( )`	`memcmp( )`	Compares the specified number of characters in two buffers.
`wmemmove( )`	`memmove( )`	Copies characters from one buffer to another in a nondestructive manner.
`wmemset( )`	`memset( )`	Copies the specified character into the specified number of locations in a destination buffer.

A.12 Codeset Conversion

The WPI provides codeset conversion capabilities through a set of functions for program use or the `iconv` command for interactive use. In the program or at the command level, specify the source and target codesets and the name of a language text file to be converted. The codesets define a conversion stream through which the language text is passed.

The following table summarizes the three functions you use for codeset conversion. These functions reside in the `libiconv.a` library.

WPI Function	Older ISO C Function	Description
`iconv_open( )`	None	Initializes a conversion stream by identifying the source and the target codesets.
`iconv_close( )`	None	Closes the conversion stream.
`iconv( )`	None	Converts an input string encoded in the source codeset to an output string encoded in the target codeset.

See Section 7.5 for a description of the `iconv` command and the types of conversions that are supported. See `iconv`(3) for information on the `iconv` library and program use.

B

Setting Up and Using User-Defined Character Databases

Japanese, Chinese, and Korean can include user-defined characters (UDCs) that supplement the characters defined in the standard character sets for Asian languages. This appendix explains how to create UDCs and the files that support UDC input and display.

You create UDCs with the `cedit` editor, discussed in Section B.1. You use the `cgen` command, discussed in Section B.2, to create font, collation, and other support files for UDCs. X applications can also obtain fonts for UDCs directly from a UDC database by using font renderers. See Section 7.6 for information about font renderers.

Note

The system default `sort` command does not access the collation files created for UDCs. Use the `asort` command to access these files. See `asort`(1) and the *Using International Software* manual for information on sorting strings that may contain these characters.

There are setup operations that you need to complete before terminals or workstation monitors can display UDCs.

The `atty` driver includes a mechanism to allow on-demand loading of files associated with UDCs. You enable this mechanism and change some of its default parameter values with the `stty` command. Table B–1 describes the `stty` command options that you use with on-demand loading.

Table B–1: The stty Options for On-Demand Loading of UDC Support Files

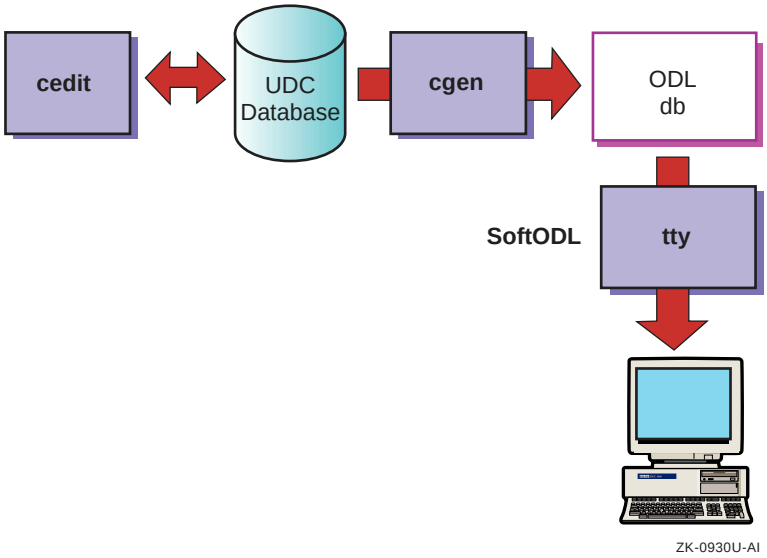
stty Option	Description
`odl`	Enables the Software On-Demand Loading (SoftODL) service.
`–odl`	Disables the Software On-Demand Loading (SoftODL) service.

Table B–1: The stty Options for On-Demand Loading of UDC Support Files (cont.)

stty Option	Description
`odlsize` *size*	Sets the maximum size of the ODL buffer. This size should be the same as a terminal's font-cache size. By default, *size* is 256 characters.
`odltype` *type*	Sets the ODL buffer replacement strategy. Valid values for *type* are `fifo` (first-in-first-out) and `lru` (least recently used)
`odldb` *path*	Sets the path to the database and other files that support UDCs. If this path is not specified, either the system default files are used or, if users are allowed to create personal UDC databases, the process default files are used. Default pathnames for various databases are specified in the `/var/i18n/conf/cp_dirs` file, which is described in Section 7.2. The `cp_dirs` file specifies, for example, that the systemwide defaults are `/var/i18n/udc` and `/var/i18n/odl`, and that the process defaults are `$HOME/.udc` and `$HOME/.odl`. Use the `odldb` option when you want to change the default `odl` file.
`odlreset`	Resets the ODL service and clears the internal ODL buffers.
`odlall`	Displays the current settings for the ODL service.

Figure B–1 demonstrates the relationship among components mentioned in Table B–1 and the SoftODL service.

Figure B–1: Components That Support User-Defined Characters



B.1 Creating User-Defined Characters

The UDC editor (invoked with the `cedit` command) is a `curses` application for managing attributes of user-defined characters. The character attributes that you usually manipulate with the `cedit` application include the following:

- Symbolic names
- Styles and sizes (16x18, 24x24, 32x32, and 40x40) for bitmap fonts
- Codeset values
- Collating values
- Input key sequences for supported input methods
- Character classes

Each user-defined character has a character attribute record, which is stored in a character attribute UDC database. A UDC database can be systemwide or private. There can be only one systemwide database that all users share. However, any user can have a private database in addition to a systemwide database.

The following command invokes the UDC editor:

```
% cedit
```

With no options, the `cedit` command uses the default database. If you are superuser, the default database is `/var/i18n/udc`. If you are not a

privileged user, the default database is `$HOME/.udc`. You can encounter a number of problems when using UDCs that are maintained in private databases. For example, users who exchange data with characters that rely on attributes defined in private databases must maintain those databases in common. To prevent these problems, make sure that a privileged user maintains all UDCs in a systemwide database. The `cedit` command has a number of options and an argument, which are described in Table B–2.

Table B–2: The cedit Command Options

cedit Options and Arguments	Description
`-c` *old_db*	Converts a Japanese ULTRIX `fedit` font file or an Asian ULTRIX character attribute database file to the format used by `cedit`.
cur_db	Specifies the path of a character attribute database (to override the default path).
`-h`	Displays `cedit` syntax.
`-r` *ref_db*	Specifies the path of the reference character attribute database (to override the default path). This database provides a model for the UDC database on which you are working with the `cedit` utility. The Reference Database item on the `cedit` File menu is an alternative to specifying the `-r` option on the `cedit` command line.

The `cedit` command returns an error message if your locale setting is not supported for creation of UDCs. Locales supported for UDCs include those for the Chinese and Japanese languages. After you invoke `cedit`, you can use the Options menu on the `cedit` user interface screen to change the language of user interface messages and help text back to English.

Note

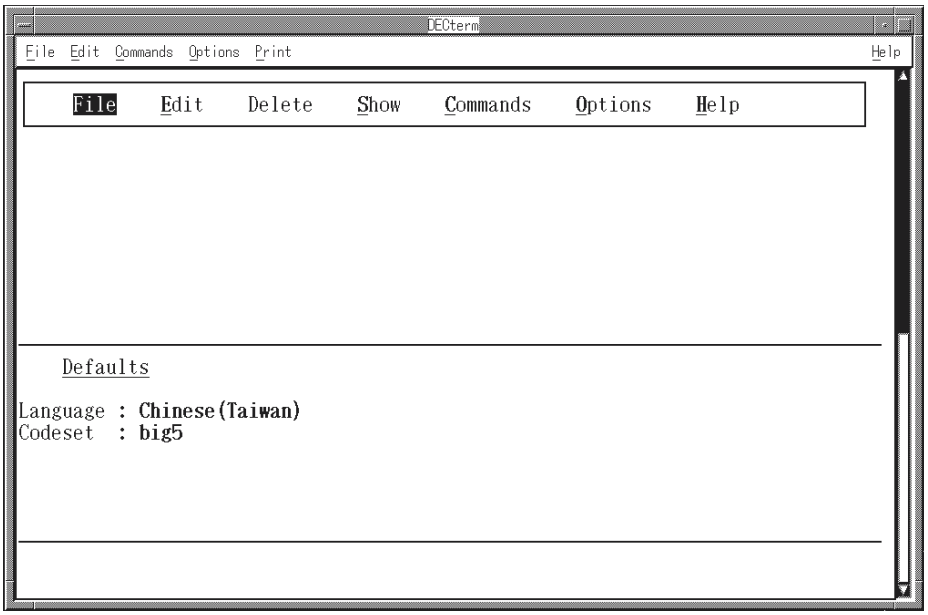
The `dtterm` terminal emulator does not support `cedit` functions and an attempt to use `cedit` functions under `dtterm` may hang the UDC Manager utility. Use the `dxterm` terminal emulator, which does support `cedit` functions.

The following sections discuss the screens, menu items, editing modes, and function keys of the `cedit` utility.

B.1.1 Working on the cedit User Interface Screen

When the LANG variable is set to a supported locale, such as zh_TW.big5, the cedit command displays the user interface screen (Figure B–2).

Figure B–2: The cedit User Interface Screen



ZK-0924U-AI

The user interface screen is divided into the following areas:

- Menu area

  This area contains a menu bar. When you choose and activate a particular menu, its items appear in the portion of the menu area below the menu bar.

- Status area

  Below the menu area is the status area, which displays the current language and codeset.

- Input and message area

  The bottom two lines of the screen accept user input and display warning or informational messages.

To see items on the menu, press the key for the letter that is underlined in the title of the menu. Alternatively, you can use the four arrow keys on the keyboard to choose a menu and then press either the Return key or the space bar.

Menu items are displayed in one of the following states:

- Active

  An active item is one that you can choose. Active items appear with one letter highlighted and underlined. You can press the key for that letter to start the function represented by the item.

- Inactive

  You cannot choose inactive items. Inactive items do not contain underlined and highlighted letters.

- Chosen

  If you press the down arrow key rather than the key for a highlighted letter, you can choose items without starting the functions they represent. The currently chosen item is in reverse video.

- Activated

  You activate an item when you press the key for a highlighted letter or when you press the Return key or the space bar after choosing the item with the down arrow key. Activating an item usually displays a pop-up menu, causes a particular function to start, or both. Activating an item that is followed by the characters >> displays a cascade menu.

To return to a higher menu level without activating items, press Ctrl/x.

Menus on the user interface screen provide the following options for managing user-defined characters and their attributes:

- File

  Use the File menu to peform the following tasks:

  – Save changes made to the current character
  – Cancel changes made to the current character
  – Change the reference character attribute database
  – Exit from the `cedit` program

- Edit

  Use the Edit menu to choose a character and create or change its font glyph, codeset value, collating value, input key sequence, class, or name.

  Section B.1.2 discusses editing a character's font glyph. For information on changing codeset or collating values and input key sequences, see the description of the Show, Commands, and Options menus in this section or see `cedit`(1).

- Delete

  Use the Delete menu to delete a character or some of its attributes.

- Show

  Use the Show menu to display attributes of the character you are working on or the status of databases (current character attribute database or reference character attribute database).

  The `cedit` utility keeps track of a character through its attribute record. This record contains fields to identify the following attributes:

  – Character number (unique for each character in the UDC database)

  – Codeset values (one for each codeset supported by a particular language and territory combination)

  – Font styles and sizes

  – Collation values (one for each collation sequence supported by the language)

  – Input key sequences (one for each input method supported by the language)

  – Class identifiers (reserved for future use)

  – Character mnemonic (reserved for future use)

  Some variation exists among Asian codesets in support for UDC attributes. For example, you cannot define an input key sequence through `cedit` for a Japanese user-defined character. For Chinese, you can define an input key sequence for use only with the DEC Hanyu codeset and TsangChi and QuickTsangChi input modes.

- Commands

  Use the Commands menu to perform the following tasks:

  – Copy character records from the reference character attribute database to the current character attribute database or, within the current character attribute database, copy records from one range of characters to another.

    You can implement the copy operation without confirmation (No Confirm), confirm the copy operation for each character in the range (Confirm All), or confirm the copy operation only for characters that will overwrite other characters (Confirm Conflict).

  – List all characters currently defined in the current character attribute database for the current language and codeset setting.

  – Scale the character's font from one size to another.

    After you define a character in one font size, you can use this option to make the character available in other sizes. The scaling algorithm is a simple one, so you might need to do some manual editing to refine font glyphs after they are scaled.

- Options

  Use the Options menu to change the current setting for language and codeset that is applied to your work on UDCs. You can also independently set the language of messages and help text in the `cedit` user interface. By default, the language of the `cedit` user interface is the same as the locale setting in effect when you invoked `cedit`.

- Help

  Use the Help menu to display introductory text for `cedit` functions. Help is also available for menu items through the Help key when this key is provided on your keyboard or, for workstation users, enabled by your terminal setting. In other words, you can first choose a menu item with the arrow keys and then press the Help key for a short description of the chosen item.

B.1.2 Editing Font Glyphs

To create or change the font glyph of a user-defined character, you must invoke the font editing screen of `cedit` as follows:

1. Choose a character by choosing the Character item from the Edit menu.

   The `cedit` program prompts you to enter the hexadecimal code value (without the `\x` prefix) for the character to be edited. The range of valid codes for UDC characters is defined in a set of configuration files. When more than one codeset is supported for the language and territory of your current locale, `cedit` attempts to supply values for the additional codesets so the character can be used with all the associated locales.

   If `cedit` cannot determine the character's value in other codesets, you can change the codeset setting through the Options menu and then explicitly specify the character's encoding in the additional codeset.

   In general, define UDCs to have values that can be mapped to other codesets supported for the language. For more information on codes for UDCs in specific Asian languages, see the following language-specific manuals:

   - *Technical Reference for Using Chinese Features*
   - *Technical Reference for Using Japanese Features*
   - *Technical Reference for Using Korean Features*
   - *Technical Reference for Using Thai Features*

   These manuals are available from the programming bookshelf of the operating system documentation Web site (**`http://www.tru64unix.compaq.com/docs/`**).

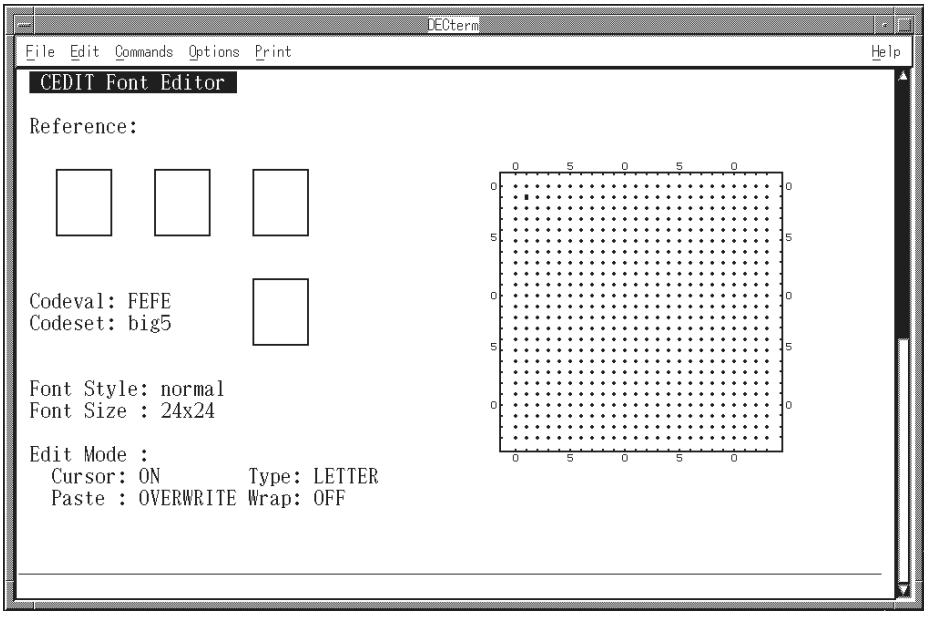
The `cedit` editor first searches your current UDC database for the code that you enter. If a character with that code is not found in the UDC database, the editor searches the current reference character database.

2. Choose the Font item from the Edit menu to see options for font style/size.
3. Choose one of the font style/size options.

   If you are creating a font glyph for use in a Motif application, the available size options may not be appropriate for the window area where you intend to use the font. In this case, choose the smallest size option that will accommodate both dimensions of your font.

The `cedit` editor then displays the full-screen font editor interface (Figure B–3).

Figure B–3: The cedit Font Editing Screen



ZK-0925U-AI

The `cedit` font editing screen has the following windows:

- The large window on the right side of the screen is where you edit the UDC font glyph. To edit, use the cursor movements and editing functions that `cedit` supports.

  Each dot on the editing window represents one pixel.
- The three small windows immediately under the Reference title display other font glyphs that you can refer to while editing the current one. You

use the `cedit` Refer function to control which font glyphs appear in these windows.

- The small window under the three reference windows is called the display window. The display window contains the font glyph you are editing in its actual size. The display window does not automatically reflect changes you make in the editing window. You must press the KP. key to update the font glyph in the display window.

Note

There are some hardware restrictions regarding font glyph displays in the small windows.

Font glyph displays in the reference and display windows are enabled only on local language terminals that support the Dynamic Replacement Character Set (DRCS) function.

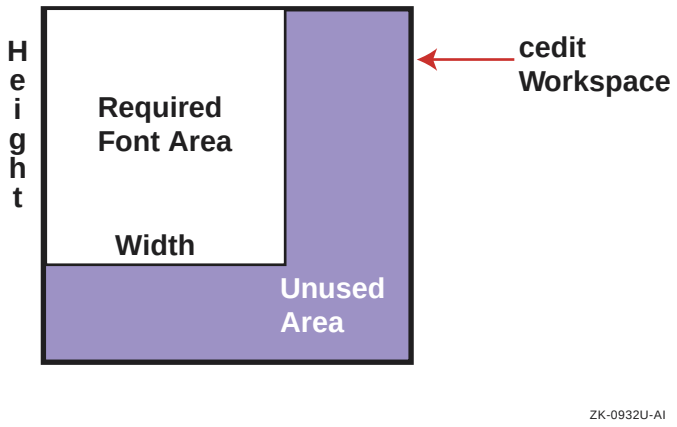
On terminal emulation windows, the font glyph in the Display window does not appear in its actual size.

Fonts created in the editing window for use with system software are processed to occupy the size dimensions you chose before the editor interface screen appeared.

You can also create a font for use with Motif applications and whose dimensions are smaller than those chosen. In this case, you confine your editing operations to a rectangle that originates at the upper-left corner of the editing window and has dimensions smaller than the available editing space (Figure B–4).

The UDC font converter that supports a Motif application considers the upper-left corner of the editing window as the font origin, generates dimensions needed to encompass the glyph based on this origin, and discards unused space outside these dimensions. This utility also allows you to explicitly specify the size dimensions for the compiled font glyphs.

Figure B–4: Interpretation of Font Editing Screen for Sizing a Font



All functions in `cedit` are bound to keys; in other words, you press a key to invoke a function. Press either the PF2 or the Help key to see a diagram of how keys are bound to editing functions. Because of differences in keypad design from system to system, your on-line diagram may vary from the one described in this section. The `cedit` editing screen has the following editing modes:

- Cursor modes

  Using the arrow keys to move the cursor does not affect the pixel state. However, when you use keypad keys to move the cursor, the following list describes how Cursor modes affect the pixel state:

  – On: Turns on the pixel under the cursor.

  – Off: Sets the pixel under the cursor off.

  – On/Off: Toggles the pixel under the cursor.

    You can also toggle the pixel under the cursor with any movement by pressing the KP5 key.

  – Move: Moves the cursor without changing the pixel state.

- Paste modes

  Paste modes control the pixel operation when you perform the paste function.

  – Overlay: Sets a pixel on if its corresponding pixel in the paste buffer is on.

  – Overwrite: Sets the pixel to the state of the corresponding pixel in the paste buffer.

- Type modes

Type modes determine whether the margin of one pixel width is maintained around the character.

  - Body: Allows you to edit the entire font glyph area.
  - Letter: Prevents you from editing the pixel value of the boundary area. Under this mode, you cannot set pixels to the on state when at the boundary of the editing window.

- Wrap modes

  Wrap modes enable or disable cursor wrapping.

  - On: Causes the cursor to wrap to the leftmost pixel when you move the cursor beyond the rightmost pixel in the editing area.

    Similar wrapping behavior occurs when you move the cursor beyond the leftmost, uppermost, and lowermost pixels in the editing area.
  - Off: Causes the bell to ring and stops cursor movement on attempts to move the cursor beyond the leftmost, rightmost, uppermost, and lowermost pixels in the editing area.

The `cedit` font editor uses four buffers to store bitmap data. Some of these buffers are used by editing functions, which are discussed following the buffer descriptions.

- Edit buffer

  This is the buffer whose contents normally appear in the editing window.

- Use buffer

  This buffer is associated with the Use function and contains a font glyph you retrieved from a UDC database or one of the reference windows.

- Cut-and-Paste buffer

  Use this buffer when pasting bitmap data in the editing window. The bitmap data being pasted is copied either from a Use buffer or the Edit buffer (if you are copying something from one section of the editing window to another).

- Undo buffer

  This buffer contains the changes made during the last edit operation and is used by the `cedit` Undo function to delete those changes.

When you are working on windows in the font-editing screen, you invoke editing functions by using keystrokes or, in some cases, through a pop-up menu that appears when you press the Do key. The following functions are available on the pop-up menu:

- Scale

This function lets you scale the current font glyph to another size supported by the system. The SCALE function does not have a keystroke alternative and is available only on the pop-up menu.

- Use

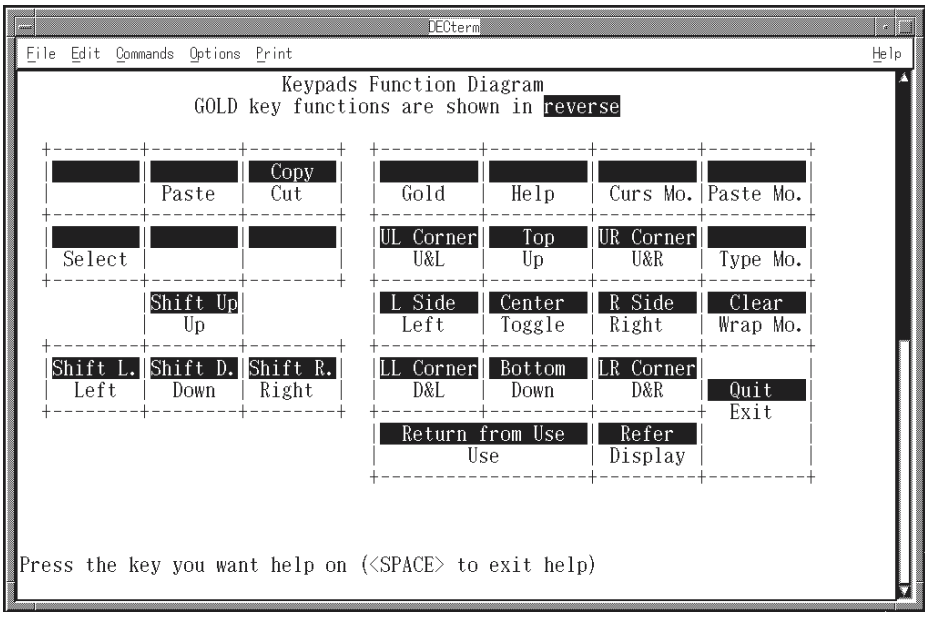
  This function retrieves a font glyph from a UDC database or from one of the reference windows.

- Refer

  This function saves a font glyph copied from a UDC database into one of the reference windows.

Figure B–5 describes the keypad keymaps for invoking different editing functions. The keypad functions, along with the letter keys used for drawing, are described in the following tables.

Figure B–5: Keymap for cedit Functions



ZK-0926U-AI

Table B–3: Keys for Miscellaneous Font Editing Functions

Key	Description
Help or PF2	Describes which keys are bound to which editing functions. Press Help along with another key in the diagram for more information on a particular key's editing function.
PF1	Toggles the GOLD state, a word processing term for a key with alternate functions. Some keypad keys represent more than one function; in this case, one of those functions is invoked by pressing PF1 and then the other keypad key.
KP.	Displays the font glyph in actual size on the display window.
GOLD KP.	Clears the font glyph displayed in the editing window.
U or u	Undoes the previous operation.
Ctrl/L	Redraws the screen.
Ctrl/z	Suspends the `cedit` program.
Do	Displays the pop-up menu for invoking SCALE, USE, and REFER functions.
Enter	Saves changes and exits from the font editor.
GOLD Enter	Quits the font editor without saving changes.

Table B–4: Keys for cedit Mode Switching

Key	Description
PF3	Toggles Cursor mode.
PF4	Toggles Paste mode.
KP-	Toggles Type mode.
KP.	Toggles Wrap mode.

Table B–5: Keys for Fine Control of Cursor Movement

Key	Description
Up-arrow	Moves the cursor up.
Down-arrow	Moves the cursor down.
Left-arrow	Moves the cursor left.
Right-arrow	Moves the cursor right.
KP7	Depending on Cursor mode, moves the cursor up and left.
KP8	Depending on Cursor mode, moves the cursor up.
KP9	Depending on Cursor mode, moves the cursor up and right.
KP4	Depending on Cursor mode, moves the cursor left.

Table B–5: Keys for Fine Control of Cursor Movement (cont.)

Key	Description
KP6	Depending on Cursor mode, moves the cursor right.
KP1	Depending on Cursor mode, moves the cursor down and left.
KP2	Depending on Cursor mode, moves the cursor down.
KP3	Depending on Cursor mode, moves the cursor down and right.
KP5	Toggles the pixel under the cursor without moving the cursor.

Table B–6: Keys for Moving Cursor to Window Areas

Key[a]	Description
GOLD KP7	Moves the cursor to the upper-left corner.
GOLD KP8	Moves the cursor to the top row.
GOLD KP9	Moves the cursor to the upper-right corner.
GOLD KP4	Moves the cursor to the leftmost column.
GOLD KP5	Moves the cursor to the center of the window.
GOLD KP6	Moves the cursor to the rightmost column.
GOLD KP1	Moves the cursor to the lower-left corner.
GOLD KP2	Moves the cursor to the bottom row.
GOLD KP3	Moves the cursor to the lower-right corner.

[a] The PF1 key toggles the GOLD state.

Table B–7: Keys for Drawing Font Glyphs

Key	Description
L or l	Draws a line connecting two selected points.
C or c	Draws a circle centered at a selected point.
r	Draws an open rectangle in a selected area.
R	Draws a solid rectangle in a selected area.
e	Draws an open ellipse in a selected area.
E	Draws a solid ellipse in a selected area.
X or x	Mirrors the font glyph along the horizontal axis (X axis).
Y or y	Mirrors the font glyph along the vertical axis (Y axis).
/	Mirrors the font glyph along the 45-degree diagonal axis.
\	Mirrors the font glyph along the 135-degree diagonal axis.

Table B–7: Keys for Drawing Font Glyphs (cont.)

Key	Description
F or f	Depending on cursor mode, fills an area.
T or t	Inverts the state of all pixels.

Table B–8: Keys for Editing Font Glyphs

Key[a]	Description
KP0	Changes the display in the Edit window from the font glyph in the Edit buffer to the font glyph in the Use buffer.
GOLD KP.	Displays font glyphs in the reference windows.
GOLD KP0	Changes the display in the Edit window from the font glyph in the Use buffer to the font glyph in the Edit buffer.
Select	Starts or cancels a selected area.
Insert	Inserts the contents of the CUT-AND-PASTE buffer.
Remove	Cuts a selected area to the CUT-AND-PASTE buffer.
GOLD Remove	Copies a selected area to the CUT-AND-PASTE buffer.
GOLD Up arrow	Shifts the font glyph up by one line.
GOLD Down arrow	Shifts the font glyph down by one line.
GOLD Left arrow	Shifts the font glyph left by one column.
GOLD Right arrow	Shifts the font glyph right by one column.

[a] The PF1 key toggles the GOLD state.

The following summary discusses the recommended method to accomplish common `cedit` operations. Keep in mind that there is often more than one way to perform the same editing operation.

- Drawing the glyph

  Use the KP1 to KP9 keys to draw and navigate in the editing window. These keys are bound to cursor movement. With the exception of KP5, you can think of these keys as points on a compass; each point represents the direction in which drawing occurs. Drawing is affected by cursor mode, which is controlled using the KP3 key. When cursor mode is set to Move, the drawing keys move the cursor without drawing anything.

  Use the KP5 key (in the middle of the compass) to toggle the pixel state on or off.

  Cursor movement is affected by Type and Wrap modes, which are bound to the KP- and KP, keys, respectively.

- Editing the glyph

  Use the drawing keys to change pixels one at a time. Several operations (cut, paste, and copy) affect pixels as a block. Use the Select function to define a select area. Then use Cut or Copy to move the block of pixels to a paste buffer. You can then move the cursor to another position and use the Paste function to move the pixels in the paste buffer to the new position. The paste operation is affected by the Paste mode setting.

  To move the entire glyph in a particular direction, you can press the GOLD or PF1 key and the appropriate arrow key.

  To undo the last editing operation, press the U key.

- Displaying the glyph in actual size

  If you are working on an Asian terminal rather than in a terminal emulation window, you can press the KP. key to display the glyph in actual size. This operation is not supported in a desktop windows environment.

- Creating multiple prototypes of a glyph

  You can create several versions of a glyph, store the versions in reference windows, and later choose the one you like best. Press the KP. key to move a glyph from the editing window to a reference window. The three reference windows are used in round-robin fashion, from left to right.

  The Refer function available from the pop-up menu allows you to move an existing glyph from the current or reference database to a reference window.

- Replacing the glyph in the editing window with another glyph

  The Use function moves a glyph into the editing window. The Use function that is bound to the keypad copies a glyph from another codepoint in the current or reference database. The Use function that is accessed from the pop-up menu moves a glyph from one of the reference windows into the editing window.

  The Use function saves a copy of the current glyph in the editing window to the Use buffer. You can retrieve the glyph from this buffer by pressing the KP0 key. Unlike the contents of the Undo buffer, the glyph in the Use buffer is available across editing operations.

- Creating multiple sizes of glyphs

  The Scale option on the `cedit` main menu creates multiple sizes of all glyphs in the database with the currently selected size. The Scale option available for the font-editing screen creates multiple sizes of only the character currently being edited. If you are working with an existing UDC database, use the Scale option from the font-editing screen rather than the `cedit` main menu. When scaling is implemented from the

`cedit` main menu and affects an entire database, the operation undoes any manual refinements that may have been made to fonts after scaling.

- Quitting the font-editing screen

  Press the Enter key to save your edits and to exit from the font editing screen.

  Press the GOLD or PF2 and Enter keys to quit without saving your edits.

After you create a font glyph, you need to specify its name, input key sequence, collating value, and, optionally, the name of the class to which the character belongs. Use the Edit menu items on the `cedit` user interface screen to specify these attributes.

B.2 Creating UDC Support Files That System Software Uses

The character attributes stored in the UDC database must be directed to specific kinds of files to meet the needs of different kinds of system software. Terminal driver software and the `asort` utility, for example, must recognize user-defined character attributes but cannot directly access information in UDC databases. Therefore, after you create or change character attributes in a UDC database, you use the `cgen` command to create the following support files:

- Font files that the SoftODL (Software On-Demand Loading) service uses
- Font files that can be directly loaded to the device
- Collating value tables for sorting characters
- Files of input key sequences for UDCs
- Font files that X and Motif applications use

The following command creates some of these files for the UDC database in `~wang/.udc`:

```
% cgen -odl -pre -col -iks ~wang/.udc
```

If you enter the `cgen` command without specifying options, statistical information about the specified database is displayed. If you are a nonprivileged user and you enter the command without specifying a UDC database, the private user database is used. If you are a superuser and you enter the command without specifying a UDC database, the system database is used. In other words, the database specification in the preceding example would not be needed if the user who entered the command was logged on as `wang`.

Table B–9 describes cgen command options. In this table, bdf format stands for Bitmap Distribution Format and pcf format stands for Portable Compiled Format. For information on these formats, see bdftopcf(1X).

Table B–9: The cgen Command Options

Option	Description
-bdf	Creates .bdf (Bitmap Distribution Format) files needed for X and DECwindows Motif applications.
-col	Creates collating value tables. You must use the asort command, rather than the sort command, if you want to apply these tables during sort operations.
-dpi 75|100	Sets resolution to either 75 or 100 when creating .bdf and .pcf files with the -bdf and -pcf options.
-fprop *property*	Sets the font property when creating .bdf and .pcf files with the -bdf and -pcf options.
-iks	Creates the input key sequence file.
-merge *font_pattern*	Invokes the fontconverter command to merge the UDC fonts with an existing .pcf font file that matches the specified *font_pattern* (for example, '*-140-*jisx0208*'). If you specify the -merge option, you must also specify the -pcf and -size options. The output .pcf file is in the form *registry_width_ height*.pcf, where *registry* is the font registry field of the specified font file.
-osiz *width*-x*height*	Specifies the font size for bdf output format. The font size in bdf format may be different from the size of the font defined in the UDC database. The font sizes that the cedit command supports are limited; the -osiz option lets you override these size restrictions both in the .bdf file and the .pcf file generated from the .bdf file. If the size parameters specified for the -osiz option are smaller than the size parameters specified for the -size option, only the upper-left portion of the UDC font glyph is used. If the size parameters specified for the -osiz option are larger than the size parameters specified for the -size option, the lower-right portion of the resulting font glyph is filled with OFF pixels.

Table B–9: The cgen Command Options (cont.)

Option	Description
`-pcf`	Invokes the `bdftopcf` command to create the `.pcf` files needed for X and Motif applications. When you use this option, the `cgen` command also invokes the `mkfontdir` and `xset` commands to make the fonts known to the font server and available to applications.
`-pre`	Creates preload font files. Preload font files are files that are directly and completely loaded to a terminal and some printers. Preload files are not useful when UDC databases are large because of the limited memory available on most devices. On-Demand Loading (ODL), which uses ODL font files, is an alternative to using preload font files.
`-odl`	Creates ODL font files. The terminal driver handles loading of fonts from ODL font files on an incremental basis, according to need and available memory.
`-win` *`userfont`*	Generates a font file with the name *`userfont`*, which can be copied to a Windows Version 3.1 or Windows NT Version 3.5 system. You must also specify the `-size` flag because only one size can apply to the specified file. Supported codesets for font files created by this option are `big5` (for Chinese Windows systems), `SJIS` (for Japanese Windows systems), and `deckorean` (for Korean Windows systems).

B.3 Processing UDC Fonts for Use with X11 or Motif Applications

The preload font files created with the `-pre` option of the `cgen` utility must be converted to BDF (Bitmap Distribution Format) or PCF (Portable Compiled Format) for use by X11 or Motif applications. The `fontconverter` command performs this conversion and can do one of the following with the converted output:

- Create independent `pcf` and `bdf` font files, which you must then install on your workstation for use by an application.
- Merge the fonts into an existing (`pcf`) font file.

The remainder of this section discusses the `fontconverter` command and its options. The `cgen` command has comparable options; in other words, you can perform `fontconverter` operations indirectly by using similar options on the `cgen` command line.

B.3.1 Using fontconverter Command Options

The following example demonstrates the simplest form of the `fontconverter` command, which produces a default name for the output files. Assume for this example and the following discussion that the locale is set to a Japanese locale when the command is entered and that 24x24 was specified in the `cedit` editor when the font glyphs were created.

```
% fontconverter \
-font -jdecw-screen-medium-r-normal--24-240-75-75-m-240-jisx0208-kanji11 \
my_font.pre
```

The preceding command converts fonts in the `my_fonts.pre` file. By default, the command creates the `JISX.UDC_24_24.pcf` and `JISX.UDC_24_24.bdf` font files.

The default base name for the output font files varies according to language, as follows:

- Japanese: `JISX.UDC`
- Hanyu: `DEC.CNS.UDC`
- Hanzi: `GB.UDC`

Font width and height are automatically appended to the base name in the names of output font files. The base name is also used in the XLFD (X Logical Font Description) as the registry name. For the fonts to be available to applications, perform one of the following actions with the compiled (`pcf`) fonts:

- In the directory where the fonts reside, enter the following commands:

  ```
  % /usr/bin/X11/mkfontdir
  % /usr/bin/X11/xset +fp ‘pwd‘
  % /usr/bin/X11/xset fp rehash
  ```

  These commands make the fonts available for testing until a server restart or system shutdown occurs.

  Alternately, you can include the `-pcf` option on the `cgen` command line to execute the `fontconverter` and `mkfontdir` commands.

- To make the fonts available on a more permanent basis (that is, after a server restart or system shutdown), use the following commands:

  1. Copy the `pcf` fonts to an existing font directory, such as `/usr/i18n/usr/lib/X11/fonts/decwin/100dpi`:

     ```
     % cp JISX.UDC_24_24.pcf \
     /usr/i18n/usr/lib/X11/fonts/decwin/100dpi
     ```

  2. Change to that directory:

     ```
     % cd /usr/i18n/usr/lib/X11/fonts/decwin/100dpi
     ```

3. Enter the `mkfontdir` command at that location:

   ```
   % /usr/bin/X11/mkfontdir
   ```

4. Enter the following command `xset` command:

   ```
   % /usr/bin/X11/xset fp rehash
   ```

Table B–10 describes options of the `fontconverter` command. With the exception of `-preload`, the options are listed in command-line order. See Section B.3.2 for examples that use these options.

Table B–10: Options and Arguments of the fontconverter Command

Argument or Option	Description
`-merge`	Specifies that command output be merged with an existing font file. See also the entry for the `-font` option.
`-w`	Specifies the font width. Use this option when the fonts are created with a width smaller than the one specified for the `cedit` font editing window.
`-h`	Specifies the font height. Use this option when the fonts are created with a height smaller than the one specified for the `cedit` font editing window.
`-udc` *base_name*	Specifies the base file name of the output UDC font file. Use this option when you are creating a standalone output file (you are not merging output into an existing file) and you do not want your output file to have a default base name.

Table B–10: Options and Arguments of the fontconverter Command (cont.)

Argument or Option	Description
`-font` *reference_font*	Specifies a reference font. The reference font is the name of a font that is available on the current display. Use the `xlsfonts` command (see `xlsfonts`(1X)) to determine which fonts are available. If you use the `-font` option with the `-merge` option, *`reference_font`* indicates the font with which converted font glyphs are merged. If you use the `-font` option without the `-merge` option, the header of *`reference_font`* is used as a reference for generating the header of the standalone output file. Information in *`reference_font`* is also used to determine default characters in the standalone output file. A default character is a glyph (usually a square) that appears when the font does not contain any glyphs for a specified code.
`-preload` *preload_font*	Specifies the input file (created by the `cgen-pre` command). Use this option when you want to specify the *`preload_font`* argument at an arbitrary position in the `fontconverter` command line. You can omit `-preload` when placing *`preload_font`* at the end of the command line.

B.3.2 Controlling Output File Format

X and Motif applications require loadable fonts in PCF format.

If you do not use the `-merge` option, the `fontconverter` command creates standalone font files in both PCF and BDF format. When you specify the `-merge` option, the command merges converted fonts with the standard PCF font specified by the `-font` option and creates a standalone file only in PCF format.

When you merge UDC fonts with standard fonts, you can use the combined file with all Motif applications.

When you create independent font files, you can use the fonts with applications that explicitly load the file. If the font registry is one of the UDC registries for a particular locale, you can also use the files with standard system applications.

Note that `fontconverter` processing time is longer when you merge fonts into an existing font file as compared to when you create independent files.

The following example of the fontconverter command:

- Converts preload format fonts in the udc_font.pre file to PCF format
- Merges the converted output with the standard font -jdecw-screen-medium-r-normal--24-240-75-75-m-240-jisx0208-kanji11
- Generates the JISX0208-Kanji11_24_24.pcf output file, which combines the standard and new fonts

```
% fontconverter -merge -font \
-jdecw-screen-medium-r-normal--24-240-75-75-m-240-jisx0208-kanji11 \
udc_font.pre
```

The following example of the fontconverter command:

- Creates thedeckanji.udc_24_24.bdf and deckanji.udc_24_24.pcf files
- Obtains the default characters and most header information for these files from the standard font -jdecw-screen-medium-r-normal--24-24-240-75-75-m-240-jisx0208-kanji11
- Sets the font registry field to deckanji.udc

```
% fontconverter -udc deckanji.udc -font \
-jdecw-screen-medium-r-normal--24-240-75-75-m-240-jisx0208-kanji11 \
udc_font.pre
```

C

Using DECterm Localization Features in Programs

This appendix discusses programming features for local language support that are available in the DECterm terminal emulator.

C.1 Drawing Ruled Lines in a DECterm Window

Programming manuals for video terminals discuss how you use ANSI escape sequences to perform operations, such as inserting and deleting characters, inserting and removing blank lines, and requesting character display in double height and width. Because a DECterm window is a terminal emulator, these escape sequences also apply to programs that display text and graphics in a DECterm window.

Operating system enhancements for Asian languages include additional escape sequences for drawing and removing ruled lines in a specified area of a DECterm window. These additional escape sequences allow applications to construct tables and diagrams.

The following sections describe the escape sequences that draw and erase lines according to pattern and area parameters.

C.1.1 Drawing Ruled Lines in a Pattern

The escape sequence identified by the mnemonic DECDRLBR draws ruled lines on the boundaries of a rectangular area according to a specified pattern. The DECDRLBR format is as follows:

```
CSI P1;Px;Plx;Py;Ply ,r
```

where:

- *P1* indicates the pattern of drawing ruled lines

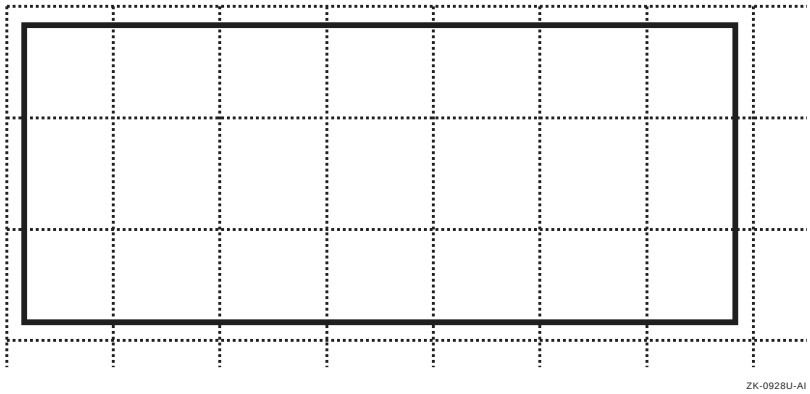
  P1 indicates whether lines are drawn on all sides of the rectangular area, on the left and right sides only, on the top and bottom only, and so forth.
- *Px* indicates the absolute position of the start point in columns
- *Plx* indicates the width of the area in columns
- *Py* indicates the absolute position of the start point in rows
- *Ply* indicates the height of the area in rows

When the DECDRLBR escape sequence is received from an application, DECterm software draws ruled lines on one or more of the boundaries of the area between the coordinates(*Px*,*Py*) and(*Px*+*Plx*-1, *Py*+*Ply* -1) according to the pattern specified in *P1*. Consider the following example:

```
CSI 15 ; 1 ; 5 ; 1 ; 2 , r
```

The preceding escape sequence causes the DECterm software to draw the ruled lines shown in Figure C–1.

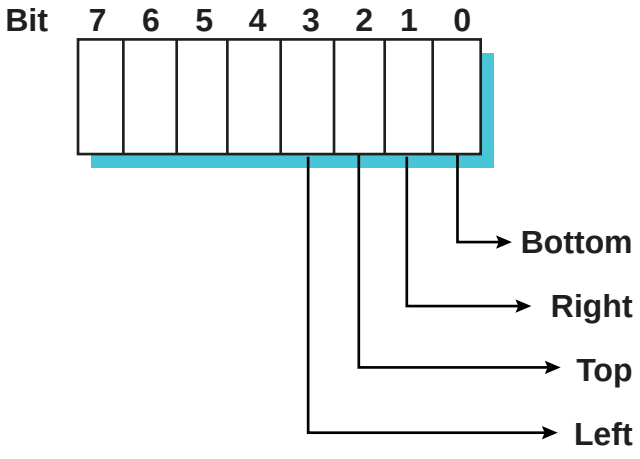
Figure C–1: Drawing Ruled Lines with the DECDRLBR Sequence



DECterm software draws ruled lines that are one pixel in width. When the display scrolls, these lines scroll in the same manner as text.

Figure C–2 and the table following the figure describe the bit pattern to which the DECDRLBR parameters map.

Figure C–2: Bit Pattern for DECDRLBR Parameters



Bit	Bit Value	Description
Bit 0	1	Draws line on the bottom boundary
Bit 1	2	Draws line on the right boundary
Bit 2	4	Draws line on the top boundary
Bit 3	8	Draws line on the left boundary

The DECDRLBR parameters are more completely described in the following list:

- Pattern of ruled lines(*P1*)

  The pattern is a bitmask that controls how the ruled lines are drawn on the boundaries of the area. Ruled lines are drawn according to whether the bits for the boundaries are set on or off. For example, ruled lines are drawn on all boundaries if *P1* is set to 15 and on the top and bottom boundary if *P1* is set to 5, for example:

```
Boundary : Bottom   Right   Top    Left
P1       = Bit0   + Bit1  + Bit2 + Bit3
P1       = 1      + 2     + 4    + 8      = 15
P1       = 1              + 4             =  5
```

- Absolute position of the start point(*Px,Py*)

  Px is the starting column position and *Py* is the starting row position. If you omit these parameters or explicitly set them to 0 (zero), the starting position is at column 1 and row 1. In other words, the upper left corner of the rectangle is at the coordinates (1,1).

- Size of the area(*Plx,Ply*)

Plx is the width of the area in columns and *Ply* is the height of the area in rows. If you omit these parameters or explicitly set them to 0 (zero), the area is 1 column in width and 1 row in height.

C.1.2 Erasing Ruled Lines in a Pattern

The DECERLBRP escape sequence erases ruled lines on the boundaries of a rectangular area according to a specified pattern. The DECERLBRP format is as follows:

```
CSI P1;Px;Plx;Ply;Py,s
```

where:

- *P1* indicates the pattern of drawing ruled lines

  P1 indicates whether lines are drawn on all sides of the rectangular area, on the left and right sides only, on the top and bottom only, and so forth.
- *Px* indicates the absolute position of the start point in columns
- *Plx* indicates the width of the area in columns
- *Py* indicates the absolute position of the start point in rows
- *Ply* indicates the height of the area in rows

C.1.3 Erasing All Ruled Lines in an Area

The escape sequence DECERLBRA erases all ruled lines in a rectangular area, not just those drawn on the area boundaries. The DECERLBRA format is as follows:

```
CSI P1;Px;Plx;Py;Ply ,t
```

where:

- *P1* determines whether the area encompasses the entire display screen or a specific section of the screen

  When *P1* is the value 1, DECterm software erases all ruled lines on the screen. In this case, the *Px*, *Plx*, *Py*, and *Ply* parameters are ignored. When *P1* is the value 2, DECterm software erases all ruled lines within a rectangular area defined by the *Px*, *Plx*, *Py*, and *Ply* parameters. When *P1* is omitted or explicitly set to 0 (zero), DECterm software erases all ruled lines on the screen (the same result as for the value 1, which is the default).
- *Px* indicates the absolute position of the start point in columns
- *Plx* indicates the width of the area in columns
- *Py* indicates the absolute position of the start point in rows
- *Ply* indicates the height of the area in rows

C.1.4 Interaction of Ruled Lines and Other DECterm Escape Sequences

Table C–1 describes the effect of using standard DECterm escape sequences when ruled lines are drawn on the screen.

Table C–1: Behavior of Standard Escape Sequences with Ruled Lines

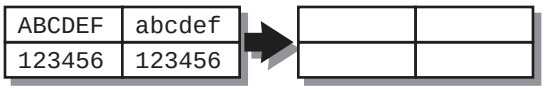
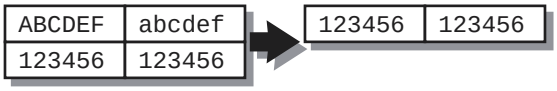
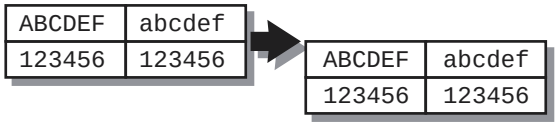
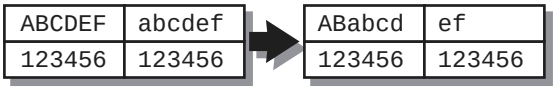
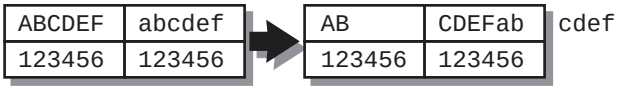
Mnemonic	Description	Effect on Ruled Lines
DECDWL, DECDHLT, DECDHLB	Display as double width or double height	These escape sequences have no effect on ruled lines, whose width is always one pixel. Furthermore, the parameter units for the escape sequences controlling ruled line display are always specified in terms of single width and single height columns and rows, even when the escape sequences are used with those that double the height and width of text.
GSM	Modify graphic size	These escape sequences have no effect on ruled lines, whose width is always one pixel. Comments made in the entry for DECDWL, DECDHLT, and DECDHLB also apply to GSM.
ED, EL, ECH	Erase display, erase line, and erase character	These escape sequences do not erase ruled lines, only the characters within the boundaries of the ruled lines. For example: 
DL	Delete line	This escape sequence erases both lines of characters and ruled lines at the active position of deletion. The text lines and accompanying ruled lines that follow the deletion point scroll up the screen. For example: 
IL	Insert line	This escape sequence causes insertion of blank lines at the active position. It causes both text and accompanying ruled lines currently at the active position to scroll down the screen. For example: 

Table C–1: Behavior of Standard Escape Sequences with Ruled Lines (cont.)

Mnemonic	Description	Effect on Ruled Lines
DCH	Delete character	This escape sequence does not delete ruled lines. The following example demonstrates the deletion of four characters at the third column position: 
ICH	Insert character	This escape sequence causes blank spaces to be inserted at the active position but has no effect on ruled lines. The following example demonstrates the insertion of four characters at the third column position: 
IRM	Invoke insert/replace mode	Insert/replace mode has no effect on ruled lines. The following example demonstrates the insertion of the characters w, x, y, and z at the third column position and the replacement of the character f with s: 
DECCOLM	Invoke column mode	Ruled lines are erased with accompanying text when column mode is in effect.
RIS, DECSTR	Reset to initial state and soft terminal, invoke reset SETUP mode	The RIS sequence erases all ruled lines displayed on the screen while the DECSTR sequence does not. The Clear Display option on the DECterm Commands Menu erases all ruled lines whereas the Reset Terminal option does not.

C.1.5 Determining DECterm Support for Ruled Lines

The feature that allows applications to draw ruled lines is enabled only when a DECterm window is emulating a terminal type that supports this feature. Your application can check for device support by requesting primary device attributes from DECterm software.

VT terminals and DECterm software return a primary device attributes report on request from applications. If the extension value 43 is included in this report, drawing ruled lines is enabled for the device. This extension is

valid at a level–2 video display or higher. For example, if a DECterm window is emulating a VT382-J terminal, which is the Japanese version of a VT382, the primary device attributes are generated as follows:

```
CSI ? 63 ; 1 ; 2 ; 4 ; 5 ; 6 ; 7 ; 8 ; 10 ; 15 ; 43 c
```

Applications can send either the `CSI c` or `CSI 0 c` escape sequence to a VT terminal or DECterm software to request a device attributes report.

C.2 DECterm Programming Restrictions

This section discusses DECterm software restrictions with respect to terminal programming features discussed in hardware manuals.

C.2.1 Downline Loadable Characters

DECterm software does not support the downline loadable characters that are used for preloading and on-demand loading of terminals. The software ignores the escape sequence for these characters.

C.2.2 DRCS Characters

DECterm software supports only the Standard Character Set (SCS) component of the DIGITAL Replacement Character Set (DRCS). When DECterm software receives the SCS characters, it searches the X Window server for the fonts with XLFD named as `-*-dec-drcs` and treats them as a soft character set. The software ignores the DECDLD control string sent by the terminal programming application.

D

Sample Locale Source Files

This appendix contains complete source files for the sample locale discussed in Chapter 6.

D.1 Character Map (charmap) Source File

This section contains the `ISO8859-1.cmap` file used for the `fr_FR.ISO8859-1@example` locale.

```
#
#     Charmap for ISO 8859-1 codeset
#
#

<code_set_name>                 "ISO8859-1"
<mb_cur_max>                    1
<mb_cur_min>                    1
<escape_char>                   \
<comment_char>                  #

CHARMAP

#  Portable characters and other standard
#  control characters

<NUL>                           \x00
<SOH>                           \x01
<STX>                           \x02
<ETX>                           \x03
<EOT>                           \x04
<ENQ>                           \x05
<ACK>                           \x06
<BEL>                           \x07
<alert>                         \x07
<backspace>                     \x08
<tab>                           \x09
<newline>                       \x0a
<vertical-tab>                  \x0b
<form-feed>                     \x0c
<carriage-return>               \x0d
<SO>                            \x0e
<SI>                            \x0f
```

```
<DLE>                       \x10
<DC1>                       \x11
<DC2>                       \x12
<DC3>                       \x13
<DC4>                       \x14
<NAK>                       \x15
<SYN>                       \x16
<ETB>                       \x17
<CAN>                       \x18
<EM>                        \x19
<SUB>                       \x1a
<ESC>                       \x1b
<IS4>                       \x1c
<IS3>                       \x1d
<IS2>                       \x1e
<IS1>                       \x1f
<SP>                        \x20
<space>                     \x20
<exclamation-mark>          \x21
<quotation-mark>            \x22
<number-sign>               \x23
<dollar-sign>               \x24
<percent-sign>              \x25
<ampersand>                 \x26
<apostrophe>                \x27
<left-parenthesis>          \x28
<right-parenthesis>         \x29
<asterisk>                  \x2a
<plus-sign>                 \x2b
<comma>                     \x2c
<hyphen>                    \x2d
<hyphen-minus>              \x2d
<period>                    \x2e
<full-stop>                 \x2e
<slash>                     \x2f
<solidus>                   \x2f
<zero>                      \x30
<one>                       \x31
<two>                       \x32
<three>                     \x33
<four>                      \x34
<five>                      \x35
<six>                       \x36
<seven>                     \x37
<eight>                     \x38
<nine>                      \x39
<colon>                     \x3a
<semicolon>                 \x3b
<less-than-sign>            \x3c
<equals-sign>               \x3d
```

```
<greater-than-sign>           \x3e
<question-mark>               \x3f
<commercial-at>               \x40
<A>                           \x41
<B>                           \x42
<C>                           \x43
<D>                           \x44
<E>                           \x45
<F>                           \x46
<G>                           \x47
<H>                           \x48
<I>                           \x49
<J>                           \x4a
<K>                           \x4b
<L>                           \x4c
<M>                           \x4d
<N>                           \x4e
<O>                           \x4f
<P>                           \x50
<Q>                           \x51
<R>                           \x52
<S>                           \x53
<T>                           \x54
<U>                           \x55
<V>                           \x56
<W>                           \x57
<X>                           \x58
<Y>                           \x59
<Z>                           \x5a
<left-square-bracket>         \x5b
<backslash>                   \x5c
<reverse-solidus>             \x5c
<right-square-bracket>        \x5d
<circumflex>                  \x5e
<circumflex-accent>           \x5e
<underscore>                  \x5f
<low-line>                    \x5f
<grave-accent>                \x60
<a>                           \x61
<b>                           \x62
<c>                           \x63
<d>                           \x64
<e>                           \x65
<f>                           \x66
<g>                           \x67
<h>                           \x68
<i>                           \x69
<j>                           \x6a
<k>                           \x6b
<l>                           \x6c
```

```
<m>                             \x6d
<n>                             \x6e
<o>                             \x6f
<p>                             \x70
<q>                             \x71
<r>                             \x72
<s>                             \x73
<t>                             \x74
<u>                             \x75
<v>                             \x76
<w>                             \x77
<x>                             \x78
<y>                             \x79
<z>                             \x7a
<left-brace>                    \x7b
<left-curly-bracket>            \x7b
<vertical-line>                 \x7c
<right-brace>                   \x7d
<right-curly-bracket>           \x7d
<tilde>                         \x7e
<DEL>                           \x7f

#
# Extended control characters
# (names taken from ISO 6429)
#

<PAD>                           \x80
<HOP>                           \x81
<BPH>                           \x82
<NBH>                           \x83
<IND>                           \x84
<NEL>                           \x85
<SSA>                           \x86
<ESA>                           \x87
<HTS>                           \x88
<HTJ>                           \x89
<VTS>                           \x8a
<PLD>                           \x8b
<PLU>                           \x8c
<RI>                            \x8d
<SS2>                           \x8e
<SS3>                           \x8f
<DCS>                           \x90
<PU1>                           \x91
<PU2>                           \x92
<STS>                           \x93
<CCH>                           \x94
<MW>                            \x95
<SPA>                           \x96
```

```
<EPA>                           \x97
<SOS>                           \x98
<SGCI>                          \x99
<SCI>                           \x9a
<CSI>                           \x9b
<ST>                            \x9c
<OSC>                           \x9d
<PM>                            \x9e
<APC>                           \x9f


#
# Other graphic characters
#

<nobreakspace>                  \xa0
<inverted-exclamation-mark>     \xa1
<cent>                          \xa2
<sterling>                      \xa3
<pound>                         \xa3
<currency>                      \xa4
<yen>                           \xa5
<broken-bar>                    \xa6
<section>                       \xa7
<diaresis>                      \xa8
<diaeresis>                     \xa8
<copyright>                     \xa9
<feminine>                      \xaa
<guillemot-left>                \xab
<not>                           \xac
<dash>                          \xad
<registered>                    \xae
<macron>                        \xaf
<degree>                        \xb0
<ring>                          \xb0
<plus-minus>                    \xb1
<superscript-two>               \xb2
<superscript-three>             \xb3
<acute>                         \xb4
<mu>                            \xb5
<micro>                         \xb5
<paragraph>                     \xb6
<dot>                           \xb7
<cedilla>                       \xb8
<superscript-one>               \xb9
<masculine>                     \xba
<guillemot-right>               \xbb
<one-quarter>                   \xbc
<one-half>                      \xbd
<three-quarters>                \xbe
```

```
<inverted-question-mark>        \xbf
<A-grave>                       \xc0
<A-acute>                       \xc1
<A-circumflex>                  \xc2
<A-tilde>                       \xc3
<A-diaeresis>                   \xc4
<A-ring>                        \xc5
<AE-ligature>                   \xc6
<C-cedilla>                     \xc7
<E-grave>                       \xc8
<E-acute>                       \xc9
<E-circumflex>                  \xca
<E-diaeresis>                   \xcb
<I-grave>                       \xcc
<I-acute>                       \xcd
<I-circumflex>                  \xce
<I-diaeresis>                   \xcf
<ETH-icelandic>                 \xd0
<N-tilde>                       \xd1
<O-grave>                       \xd2
<O-acute>                       \xd3
<O-circumflex>                  \xd4
<O-tilde>                       \xd5
<O-diaeresis>                   \xd6
<multiplication>                \xd7
<O-slash>                       \xd8
<U-grave>                       \xd9
<U-acute>                       \xda
<U-circumflex>                  \xdb
<U-diaeresis>                   \xdc
<Y-acute>                       \xdd
<THORN-icelandic>               \xde
<s-sharp>                       \xdf
<a-grave>                       \xe0
<a-acute>                       \xe1
<a-circumflex>                  \xe2
<a-tilde>                       \xe3
<a-diaeresis>                   \xe4
<a-ring>                        \xe5
<ae-ligature>                   \xe6
<c-cedilla>                     \xe7
<e-grave>                       \xe8
<e-acute>                       \xe9
<e-circumflex>                  \xea
<e-diaeresis>                   \xeb
<i-grave>                       \xec
<i-acute>                       \xed
<i-circumflex>                  \xee
<i-diaeresis>                   \xef
<eth-icelandic>                 \xf0
```

```
<n-tilde>                       \xf1
<o-grave>                       \xf2
<o-acute>                       \xf3
<o-circumflex>                  \xf4
<o-tilde>                       \xf5
<o-diaeresis>                   \xf6
<division>                      \xf7
<o-slash>                       \xf8
<u-grave>                       \xf9
<u-acute>                       \xfa
<u-circumflex>                  \xfb
<u-diaeresis>                   \xfc
<y-acute>                       \xfd
<thorn-icelandic>               \xfe
<y-diaeresis>                   \xff

END CHARMAP
```

D.2 Locale Definition Source File

This section contains the `fr_FR.ISO8859-1@example.src` file used in the examples in Chapter 6.

```
###########################################################################
# Locale Source for fr_FR (French in France) locale                       #
###########################################################################
#############
LC_CTYPE
#############

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
        <A-grave>;\
        <A-circumflex>;\
        <AE-ligature>;\
        <C-cedilla>;\
        <E-grave>;\
        <E-acute>;\
        <E-circumflex>;\
        <E-diaeresis>;\
        <I-circumflex>;\
        <I-diaeresis>;\
        <O-circumflex>;\
        <U-grave>;\
        <U-circumflex>;\
        <U-diaeresis>

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
        <a-grave>;\
        <a-circumflex>;\
        <ae-ligature>;\
        <c-cedilla>;\
        <e-grave>;\
        <e-acute>;\
        <e-circumflex>;\
        <e-diaeresis>;\
```

```
       <i-circumflex>;\
       <i-diaeresis>;\
       <o-circumflex>;\
       <u-grave>;\
       <u-circumflex>;\
       <u-diaeresis>

space  <tab>;<newline>;<vertical-tab>;<form-feed>;\
       <carriage-return>;<space>

cntrl  <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;\
       <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
       <form-feed>;<carriage-return>;\
       <SO>;<SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
       <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
       <IS1>;<DEL>;\
       <PAD>;<HOP>;<BPH>;<NBH>;<IND>;<NEL>;<SSA>;<ESA>;\
       <HTS>;<HTJ>;<VTS>;<PLD>;<PLU>;<RI>;<SS2>;<SS3>;\
       <DCS>;<PU1>;<PU2>;<STS>;<CCH>;<MW>;<SPA>;<EPA>;\
       <SOS>;<SGCI>;<SCI>;<CSI>;<ST>;<OSC>;<PM>;<APC>

graph  <exclamation-mark>;<quotation-mark>;<number-sign>;\
       <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
       <left-parenthesis>;<right-parenthesis>;<asterisk>;<plus-sign>;\
       <comma>;<hyphen>;<period>;<slash>;\
       <zero>;<one>;<two>;<three>;<four>;<five>;<six>;<seven>;<eight>;<nine>;\
       <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
       <greater-than-sign>;<question-mark>;<commercial-at>;\
       <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
       <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
       <left-square-bracket>;<backslash>;<right-square-bracket>;\
       <circumflex>;<underscore>;<grave-accent>;\
       <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
       <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
       <left-brace>;<vertical-line>;<right-brace>;<tilde>;\
       <inverted-exclamation-mark>;<cent>;<sterling>;<currency>;<yen>;\
       <broken-bar>;<section>;<diaeresis>;<copyright>;<feminine>;\
       <guillemot-left>;<not>;<dash>;<registered>;<macron>;\
       <degree>;<plus-minus>;<superscript-two>;<superscript-three>;\
       <acute>;<mu>;<paragraph>;<dot>;<cedilla>;<superscript-one>;\
       <masculine>;<guillemot-right>;<one-quarter>;<one-half>;\
       <three-quarters>;<inverted-question-mark>;\
       <A-grave>;<A-acute>;<A-circumflex>;<A-tilde>;<A-diaeresis>;\
       <A-ring>;<AE-ligature>;<C-cedilla>;<E-grave>;<E-acute>;<E-circumflex>;\
       <E-diaeresis>;<I-grave>;<I-acute>;<I-circumflex>;<I-diaeresis>;\
       <ETH-icelandic>;<N-tilde>;<O-grave>;<O-acute>;<O-circumflex>;<O-tilde>;\
       <O-diaeresis>;<multiplication>;<O-slash>;<U-grave>;<U-acute>;\
       <U-circumflex>;<U-diaeresis>;<Y-acute>;<THORN-icelandic>;<s-sharp>;\
       <a-grave>;<a-acute>;<a-circumflex>;<a-tilde>;<a-diaeresis>;\
       <a-ring>;<ae-ligature>;<c-cedilla>;<e-grave>;<e-acute>;<e-circumflex>;\
       <e-diaeresis>;<i-grave>;<i-acute>;<i-circumflex>;<i-diaeresis>;\
       <eth-icelandic>;<n-tilde>;<o-grave>;<o-acute>;<o-circumflex>;<o-tilde>;\
       <o-diaeresis>;<division>;<o-slash>;<u-grave>;<u-acute>;\
       <u-circumflex>;<u-diaeresis>;<y-acute>;<thorn-icelandic>;<y-diaeresis>

print  <exclamation-mark>;<quotation-mark>;<number-sign>;\
       <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
       <left-parenthesis>;<right-parenthesis>;<asterisk>;<plus-sign>;\
       <comma>;<hyphen>;<period>;<slash>;\
       <zero>;<one>;<two>;<three>;<four>;<five>;<six>;<seven>;<eight>;<nine>;\
       <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
       <greater-than-sign>;<question-mark>;<commercial-at>;\
       <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
       <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
```

```
       <left-square-bracket>;<backslash>;<right-square-bracket>;\
       <circumflex>;<underscore>;<grave-accent>;\
       <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
       <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
       <left-brace>;<vertical-line>;<right-brace>;<tilde>;\
       <inverted-exclamation-mark>;<cent>;<sterling>;<currency>;<yen>;\
       <broken-bar>;<section>;<diaeresis>;<copyright>;<feminine>;\
       <guillemot-left>;<not>;<dash>;<registered>;<macron>;\
       <degree>;<plus-minus>;<superscript-two>;<superscript-three>;\
       <acute>;<mu>;<paragraph>;<dot>;<cedilla>;<superscript-one>;\
       <masculine>;<guillemot-right>;<one-quarter>;<one-half>;\
       <three-quarters>;<inverted-question-mark>;\
       <A-grave>;<A-acute>;<A-circumflex>;<A-tilde>;<A-diaeresis>;\
       <A-ring>;<AE-ligature>;<C-cedilla>;<E-grave>;<E-acute>;<E-circumflex>;\
       <E-diaeresis>;<I-grave>;<I-acute>;<I-circumflex>;<I-diaeresis>;\
       <ETH-icelandic>;<N-tilde>;<O-grave>;<O-acute>;<O-circumflex>;<O-tilde>;\
       <O-diaeresis>;<multiplication>;<O-slash>;<U-grave>;<U-acute>;\
       <U-circumflex>;<U-diaeresis>;<Y-acute>;<THORN-icelandic>;<s-sharp>;\
       <a-grave>;<a-acute>;<a-circumflex>;<a-tilde>;<a-diaeresis>;\
       <a-ring>;<ae-ligature>;<c-cedilla>;<e-grave>;<e-acute>;<e-circumflex>;\
       <e-diaeresis>;<i-grave>;<i-acute>;<i-circumflex>;<i-diaeresis>;\
       <eth-icelandic>;<n-tilde>;<o-grave>;<o-acute>;<o-circumflex>;<o-tilde>;\
       <o-diaeresis>;<division>;<o-slash>;<u-grave>;<u-acute>;\
       <u-circumflex>;<u-diaeresis>;<y-acute>;<thorn-icelandic>;<y-diaeresis>;\
       <space>

punct  <exclamation-mark>;<quotation-mark>;<number-sign>;\
       <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
       <left-parenthesis>;<right-parenthesis>;<asterisk>;\
       <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
       <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
       <greater-than-sign>;<question-mark>;<commercial-at>;\
       <left-square-bracket>;<backslash>;<right-square-bracket>;\
       <circumflex>;<underscore>;<grave-accent>;<left-brace>;\
       <vertical-line>;<right-brace>;<tilde>

digit  <zero>;<one>;<two>;<three>;<four>;\
       <five>;<six>;<seven>;<eight>;<nine>

xdigit <zero>;<one>;<two>;<three>;<four>;\
       <five>;<six>;<seven>;<eight>;<nine>;\
       <A>;<B>;<C>;<D>;<E>;<F>;\
       <a>;<b>;<c>;<d>;<e>;<f>

blank  <space>;<tab>

toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
       (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
       (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
       (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
       (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);\
       (<z>,<Z>);\
       (<a-grave>,<A-grave>);\
       (<a-circumflex>,<A-circumflex>);\
       (<ae-ligature>,<AE-ligature>);\
       (<c-cedilla>,<C-cedilla>);\
       (<e-grave>,<E-grave>);\
       (<e-acute>,<E-acute>);\
       (<e-circumflex>,<E-circumflex>);\
       (<e-diaeresis>,<E-diaeresis>);\
       (<i-circumflex>,<I-circumflex>);\
       (<i-diaeresis>,<I-diaeresis>);\
       (<o-circumflex>,<O-circumflex>);\
       (<u-grave>,<U-grave>);\
```

```
       (<u-circumflex>,<U-circumflex>);\
       (<u-diaeresis>,<U-diaeresis>)

tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);\
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);\
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);\
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);\
        (<Z>,<z>);\
        (<A-grave>,<a-grave>);\
        (<A-circumflex>,<a-circumflex>);\
        (<AE-ligature>,<ae-ligature>);\
        (<C-cedilla>,<c-cedilla>);\
        (<E-grave>,<e-grave>);\
        (<E-acute>,<e-acute>);\
        (<E-circumflex>,<e-circumflex>);\
        (<E-diaeresis>,<e-diaeresis>);\
        (<I-circumflex>,<i-circumflex>);\
        (<I-diaeresis>,<i-diaeresis>);\
        (<O-circumflex>,<o-circumflex>);\
        (<U-grave>,<u-grave>);\
        (<U-circumflex>,<u-circumflex>);\
        (<U-diaeresis>,<u-diaeresis>)

END LC_CTYPE


##############
LC_COLLATE
##############
#
#  The order is control characters, followed by punctuation
#  and digits, and then letters. The letters have a
#  multi-level sort with diacritics and case being
#  ignored on the first plass, then diacritics being
#  significant on the second pass, and then case being
#  significant on the third (last) pass.
#
order_start                     forward;backward;forward

<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
```

```
<CAN>
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<PAD>
<HOP>
<BPH>
<NBH>
<IND>
<NEL>
<SSA>
<ESA>
<HTS>
<HTJ>
<VTS>
<PLD>
<PLU>
<RI>
<SS2>
<SS3>
<DCS>
<PU1>
<PU2>
<STS>
<CCH>
<MW>
<SPA>
<EPA>
<SOS>
<SGCI>
<SCI>
<CSI>
<ST>
<OSC>
<PM>
<APC>
<space>                 <space>;<space>;<space>
<exclamation-mark>      <exclamation-mark>;<exclamation-mark>;<exclamation-mark>
<quotation-mark>        <quotation-mark>;<quotation-mark>;<quotation-mark>
<number-sign>           <number-sign>;<number-sign>;<number-sign>
<dollar-sign>           <dollar-sign>;<dollar-sign>;<dollar-sign>
<percent-sign>          <percent-sign>;<percent-sign>;<percent-sign>
<ampersand>             <ampersand>;<ampersand>;<ampersand>
<apostrophe>            <apostrophe>;<apostrophe>;<apostrophe>
<left-parenthesis>      <left-parenthesis>;<left-parenthesis>;<left-parenthesis>
<right-parenthesis>     <right-parenthesis>;<right-parenthesis>;<right-parenthesis>
<asterisk>              <asterisk>;<asterisk>;<asterisk>
<plus-sign>             <plus-sign>;<plus-sign>;<plus-sign>
<comma>                 <comma>;<comma>;<comma>
<hyphen-minus>          <hyphen-minus>;<hyphen-minus>;<hyphen-minus>
<period>                <period>;<period>;<period>
<slash>                 <slash>;<slash>;<slash>
<zero>                  <zero>;<zero>;<zero>
<one>                   <one>;<one>;<one>
<two>                   <two>;<two>;<two>
<three>                 <three>;<three>;<three>
<four>                  <four>;<four>;<four>
<five>                  <five>;<five>;<five>
<six>                   <six>;<six>;<six>
<seven>                 <seven>;<seven>;<seven>
```

```
<eight>                 <eight>;<eight>;<eight>
<nine>                  <nine>;<nine>;<nine>
<colon>                 <colon>;<colon>;<colon>
<semicolon>             <semicolon>;<semicolon>;<semicolon>
<less-than-sign>        <less-than-sign>;<less-than-sign>;<less-than-sign>
<equals-sign>           <equals-sign>;<equals-sign>;<equals-sign>
<greater-than-sign>     <greater-than-sign>;<greater-than-sign>;<greater-than-sign>
<question-mark>         <question-mark>;<question-mark>;<question-mark>
<commercial-at>         <commercial-at>;<commercial-at>;<commercial-at>
<left-square-bracket>   <left-square-bracket>;<left-square-bracket>;<left-square-bracket>
<backslash>             <backslash>;<backslash>;<backslash>
<right-square-bracket>  <right-square-bracket>;<right-square-bracket>;<right-square-bracket>
<circumflex>            <circumflex>;<circumflex>;<circumflex>
<underscore>            <underscore>;<underscore>;<underscore>
<grave-accent>          <grave-accent>;<grave-accent>;<grave-accent>
<left-brace>            <left-brace>;<left-brace>;<left-brace>
<vertical-line>         <vertical-line>;<vertical-line>;<vertical-line>
<right-brace>           <right-brace>;<right-brace>;<right-brace>
<tilde>                 <tilde>;<tilde>;<tilde>
<DEL>                   <DEL>;<DEL>;<DEL>
<nobreakspace>          <nobreakspace>;<nobreakspace>;<nobreakspace>
<inverted-exclamation-mark>  \
<inverted-exclamation-mark>;<inverted-exclamation-mark>;<inverted-exclamation-mark>
<cent>                  <cent>;<cent>;<cent>
<sterling>              <sterling>;<sterling>;<sterling>
<currency>              <currency>;<currency>;<currency>
<yen>                   <yen>;<yen>;<yen>
<broken-bar>            <broken-bar>;<broken-bar>;<broken-bar>
<paragraph>             <paragraph>;<paragraph>;<paragraph>
<diaeresis>             <diaeresis>;<diaeresis>;<diaeresis>
<copyright>             <copyright>;<copyright>;<copyright>
<guillemot-left>        <guillemot-left>;<guillemot-left>;<guillemot-left>
<not>                   <not>;<not>;<not>
<dash>                  <dash>;<dash>;<dash>
<registered>            <registered>;<registered>;<registered>
<macron>                <macron>;<macron>;<macron>
<degree>                <degree>;<degree>;<degree>
<plus-minus>            <plus-minus>;<plus-minus>;<plus-minus>
<superscript-two>       <two>;<superscript-two>;<superscript-two>
<superscript-three>     <three>;<superscript-three>;<superscript-three>
<acute>                 <acute>;<acute>;<acute>
<mu>                    <mu>;<mu>;<mu>
<section>               <section>;<section>;<section>
<dot>                   <dot>;<dot>;<dot>
<cedilla>               <cedilla>;<cedilla>;<cedilla>
<superscript-one>       <one>;<superscript-one>;<superscript-one>
<guillemot-right>       <guillemot-right>;<guillemot-right>;<guillemot-right>
<one-quarter>           <zero>;<one-quarter>;<one-quarter>
<one-half>              <zero>;<one-half>;<one-half>
<three-quarters>        <zero>;<three-quarters>;<three-quarters>
<inverted-question-mark> \
<inverted-question-mark>;<inverted-question-mark>;<inverted-question-mark>
<multiplication>        <multiplication>;<multiplication>;<multiplication>
<division>              <division>;<division>;<division>
<a>                     <a>;<a>;<a>
<A>                     <a>;<a>;<A>
<feminine>              <a>;<feminine>;<feminine>
<a-acute>               <a>;<a-acute>;<a-acute>
<A-acute>               <a>;<a-acute>;<A-acute>
<a-grave>               <a>;<a-grave>;<a-grave>
<A-grave>               <a>;<a-grave>;<A-grave>
<a-circumflex>          <a>;<a-circumflex>;<a-circumflex>
<A-circumflex>          <a>;<a-circumflex>;<A-circumflex>
<a-ring>                <a>;<a-ring>;<a-ring>
```

```
<A-ring>                <a>;<a-ring>;<A-ring>
<a-diaeresis>           <a>;<a-diaeresis>;<a-diaeresis>
<A-diaeresis>           <a>;<a-diaeresis>;<A-diaeresis>
<a-tilde>               <a>;<a-tilde>;<a-tilde>
<A-tilde>               <a>;<a-tilde>;<A-tilde>
<ae-ligature>           <a>;<a><e>;<a><e>
<AE-ligature>           <a>;<a><e>;<A><E>
<b>                     <b>;<b>;<b>
<B>                     <b>;<b>;<B>
<c>                     <c>;<c>;<c>
<C>                     <c>;<c>;<C>
<c-cedilla>             <c>;<c-cedilla>;<c-cedilla>
<C-cedilla>             <c>;<c-cedilla>;<C-cedilla>
<d>                     <d>;<d>;<d>
<D>                     <d>;<d>;<D>
<eth-icelandic>         <d>;<eth-icelandic>;<eth-icelandic>
<ETH-icelandic>         <d>;<eth-icelandic>;<ETH-icelandic>
<e>                     <e>;<e>;<e>
<E>                     <e>;<e>;<E>
<e-acute>               <e>;<e-acute>;<e-acute>
<E-acute>               <e>;<e-acute>;<E-acute>
<e-grave>               <e>;<e-grave>;<e-grave>
<E-grave>               <e>;<e-grave>;<E-grave>
<e-circumflex>          <e>;<e-circumflex>;<e-circumflex>
<E-circumflex>          <e>;<e-circumflex>;<E-circumflex>
<e-diaeresis>           <e>;<e-diaeresis>;<e-diaeresis>
<E-diaeresis>           <e>;<e-diaeresis>;<E-diaeresis>
<f>                     <f>;<f>;<f>
<F>                     <f>;<f>;<F>
<g>                     <g>;<g>;<g>
<G>                     <g>;<g>;<G>
<h>                     <h>;<h>;<h>
<H>                     <h>;<h>;<H>
<i>                     <i>;<i>;<i>
<I>                     <i>;<i>;<I>
<i-acute>               <i>;<i-acute>;<i-acute>
<I-acute>               <i>;<i-acute>;<I-acute>
<i-grave>               <i>;<i-grave>;<i-grave>
<I-grave>               <i>;<i-grave>;<I-grave>
<i-circumflex>          <i>;<i-circumflex>;<i-circumflex>
<I-circumflex>          <i>;<i-circumflex>;<I-circumflex>
<i-diaeresis>           <i>;<i-diaeresis>;<i-diaeresis>
<I-diaeresis>           <i>;<i-diaeresis>;<I-diaeresis>
<j>                     <j>;<j>;<j>
<J>                     <j>;<j>;<J>
<k>                     <k>;<k>;<k>
<K>                     <k>;<k>;<K>
<l>                     <l>;<l>;<l>
<L>                     <l>;<l>;<L>
<m>                     <m>;<m>;<m>
<M>                     <m>;<m>;<M>
<n>                     <n>;<n>;<n>
<N>                     <n>;<n>;<N>
<n-tilde>               <n>;<n-tilde>;<n-tilde>
<N-tilde>               <n>;<n-tilde>;<N-tilde>
<o>                     <o>;<o>;<o>
<O>                     <o>;<o>;<O>
<masculine>             <o>;<masculine>;<masculine>
<o-acute>               <o>;<o-acute>;<o-acute>
<O-acute>               <o>;<o-acute>;<O-acute>
<o-grave>               <o>;<o-grave>;<o-grave>
<O-grave>               <o>;<o-grave>;<O-grave>
<o-circumflex>          <o>;<o-circumflex>;<o-circumflex>
<O-circumflex>          <o>;<o-circumflex>;<O-circumflex>
```

```
<o-diaeresis>           <o>;<o-diaeresis>;<o-diaeresis>
<O-diaeresis>           <o>;<o-diaeresis>;<O-diaeresis>
<o-tilde>               <o>;<o-tilde>;<o-tilde>
<O-tilde>               <o>;<o-tilde>;<O-tilde>
<o-slash>               <o>;<o-slash>;<o-slash>
<O-slash>               <o>;<o-slash>;<O-slash>
<p>                     <p>;<p>;<p>
<P>                     <p>;<p>;<P>
<q>                     <q>;<q>;<q>
<Q>                     <q>;<q>;<Q>
<r>                     <r>;<r>;<r>
<R>                     <r>;<r>;<R>
<s>                     <s>;<s>;<s>
<S>                     <s>;<s>;<S>
<s-sharp>               <s>;<s><s>;<s><s>
<t>                     <t>;<t>;<t>
<T>                     <t>;<t>;<T>
<thorn-icelandic>       <t>;<t><h>;<t><h>
<THORN-icelandic>       <t>;<t><h>;<T><h>
<u>                     <u>;<u>;<u>
<U>                     <u>;<u>;<U>
<u-acute>               <u>;<u-acute>;<u-acute>
<U-acute>               <u>;<u-acute>;<U-acute>
<u-grave>               <u>;<u-grave>;<u-grave>
<U-grave>               <u>;<u-grave>;<U-grave>
<u-circumflex>          <u>;<u-circumflex>;<u-circumflex>
<U-circumflex>          <u>;<u-circumflex>;<U-circumflex>
<u-diaeresis>           <u>;<u-diaeresis>;<u-diaeresis>
<U-diaeresis>           <u>;<u-diaeresis>;<U-diaeresis>
<v>                     <v>;<v>;<v>
<V>                     <v>;<v>;<V>
<w>                     <w>;<w>;<w>
<W>                     <w>;<w>;<W>
<x>                     <x>;<x>;<x>
<X>                     <x>;<x>;<X>
<y>                     <y>;<y>;<y>
<Y>                     <y>;<y>;<Y>
<y-acute>               <y>;<y-acute>;<y-acute>
<Y-acute>               <y>;<y-acute>;<Y-acute>
<y-diaeresis>           <y>;<y-diaeresis>;<y-diaeresis>
<z>                     <z>;<z>;<z>
<Z>                     <z>;<z>;<Z>
UNDEFINED
order_end

END LC_COLLATE


#############
LC_MONETARY
#############

int_curr_symbol   "<F><R><F><space>"
currency_symbol   "<F>"
mon_decimal_point "<comma>"
mon_thousands_sep ""
mon_grouping      3;0
positive_sign     ""
negative_sign     "<hyphen>"
int_frac_digits   2
frac_digits       2
p_cs_precedes     0
p_sep_by_space    1
n_cs_precedes     0
```

```
n_sep_by_space    1
p_sign_posn       1
n_sign_posn       1

END LC_MONETARY


#############
LC_NUMERIC
#############

decimal_point     "<comma>"
thousands_sep     ""
grouping          3;0

END LC_NUMERIC


#############
LC_TIME
#############
# abbreviated day names
abday   "<d><i><m>";\
        "<l><u><n>";\
        "<m><a><r>";\
        "<m><e><r>";\
        "<j><e><u>";\
        "<v><e><n>";\
        "<s><a><m>"

# full day names
day     "<d><i><m><a><n><c><h><e>";\
        "<l><u><n><d><i>";\
        "<m><a><r><d><i>";\
        "<m><e><r><c><r><e><d><i>";\
        "<j><e><u><d><i>";\
        "<v><e><n><d><r><e><d><i>";\
        "<s><a><m><e><d><i>"

# abbreviated month names
abmon   "<j><a><n>";\
        "<f><e-acute><v>";\
        "<m><a><r>";\
        "<a><v><r>";\
        "<m><a><i>";\
        "<j><u><n>";\
        "<j><u><l>";\
        "<a><o><u-circumflex>";\
        "<s><e><p>";\
        "<o><c><t>";\
        "<n><o><v>";\
        "<d><e-acute><c>"

# full month names
mon     "<j><a><n><v><i><e><r>";\
        "<f><e-acute><v><r><i><e><r>";\
        "<m><a><r><s>";\
        "<a><v><r><i><l>";\
        "<m><a><i>";\
        "<j><u><i><n>";\
        "<j><u><i><l><l><e><t>";\
        "<a><o><u-circumflex><t>";\
        "<s><e><p><t><e><m><b><r><e>";\
        "<o><c><t><o><b><r><e>";\
```

```
        "<n><o><v><e><m><b><r><e>";\
        "<d><e-acute><c><e><m><b><r><e>"

# date/time format. The following designates this
# format: "%a %e %b %H:%M:%S %Z %Y"
d_t_fmt "<percent-sign><a><space><percent-sign><e>\
<space><percent-sign><b><space><percent-sign><H>\
<colon><percent-sign><M><colon><percent-sign><S>\
<space><percent-sign><Z><space><percent-sign><Y>"

# date format. The following designates this
# format: "%d.%m.%y"
d_fmt   "<percent-sign><d><period><percent-sign><m>\
<period><percent-sign><y>"

# time format. The following designates this
# format: "%H:%M:%S"
t_fmt   "<percent-sign><H><colon><percent-sign><M>\
<colon><percent-sign><S>"

am_pm   "<semicolon>"

# 12-hour time representation. This is empty, meaning
# this locale always uses 24-hour format.
t_fmt_ampm        ""

END LC_TIME


#############
LC_MESSAGES
#############

# yes expression. The following designates:
# "^([oO]|[oO][uU][iI])"
yesexpr "<circumflex><left-parenthesis>\
<left-square-bracket><o><O><right-square-bracket>\
<vertical-line><left-square-bracket><o><O>\
<right-square-bracket><left-square-bracket><u><U>\
<right-square-bracket><left-square-bracket><i><I>\
<right-square-bracket><right-parenthesis>"

# no expression. The following designates:
# "^([nN]|[nN][oO][nN])"
noexpr        "<circumflex><left-parenthesis>\
<left-square-bracket><n><N><right-square-bracket>\
<vertical-line><left-square-bracket><n><N>\
<right-square-bracket><left-square-bracket><o><O>\
<right-square-bracket><left-square-bracket><n><N>\
<right-square-bracket><right-parenthesis>"

# yes string. The following designates: "oui:o:O"
yesstr  "<o><u><i><colon><o><colon><O>"

# no string. The following designates: "non:n:N"
nostr   "<n><o><n><colon><n><colon><N>"

END LC_MESSAGES
```

Glossary

ASCII
American Standard Code for Information Interchange. ASCII defines 128 characters, including control characters and graphic characters, represented by 7-bit binary values (see also ISO 646).

See also *character set*, *coded character set*

C locale
The standard, or default, language environment. This environment is always in effect for non-internationalized applications or when locales are not installed or are not active.

character
A sequence of one or more bytes that represents a single graphic symbol or control code. Unlike the `char` datatype in C, a character can be represented by a value that is one byte or multiple bytes. The expression "multibyte character" and the term "character" both refer to character values of any length, including single-byte values.

See also *wide character*

character set
A member of a set of elements used for the organization, control, or representation of text.

See also *ASCII*, *ISO 10646*

character string
A contiguous sequence of bytes that is terminated by, and includes, the null byte. A string is an array of type `char` in the C programming language. The null byte has all bits set to zero (0).

An empty string is a character string whose first element is the null byte.

See also *character*, *wide-character string*

code page
See *coded character set*

coded character set
A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation. On UNIX systems, the more common term is codeset. On

MS-DOS and Microsoft Windows systems, the more common term is code page.

codeset

See *coded character set*

collating sequence

The ordering rules applied to characters or groups of characters when they are sorted.

control character

A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text.

cultural data

The conventions of a geographical area for such things as date, time, numeric, and currency values.

data

Information generated internally, information extracted from or written to files, and message text used for communication with the program's user.

dense code

The operating system supports two types of locales; dense code and Unicode. Dense code locales use a wide-character encoding that minimizes table size by assigning codepoints consecutively with no empty positions. Under dense code locales, a wchar_t value for one locale may not represent the same character in another locale and, thus, is locale specific.

See also *Unicode*

euro

The currency adopted by European countries belonging to the Economic and Monetary Union (EMU) and scheduled to replace local currencies for EMU member countries in the year 2002. The euro currency has a monetary sign that looks like an equal sign (=) superimposed on the capital letter C and is identified by the string EUR in international currency documents.

file code

The encoding format that applies to data outside the program.

Contrast with *process code*

graphic character

A character, other than a control character, that has a visual representation when handwritten, printed, or displayed. Also, ideograph.

I18N

See *internationalization*

internationalization

The process of developing programs without prior knowledge of the language, cultural data, or character-encoding schemes that the programs are expected to handle. An internationalized program uses a set of interfaces that allows the program to modify its behavior at run time for operation in a specific native language environment. I18N is frequently used as an abbreviation for internationalization.

See also *locale*, *localization*

ISO 10646

The ISO Universal Character Set (UCS). The first 65,536 code positions in this character set are called the Base Multilingual Plane (BMP), in which each character is 16 bits in length. This form of ISO 10646 is also known as UCS-2. ISO 10646 also has a form called UCS-4, in which each character is 32 bits in length.

See also *Unicode*

ISO 646

ISO 7-bit codeset for information interchange. The reference version of ISO 646 contains 95 graphic characters, which are identical to the graphic characters defined in the ASCII codeset.

ISO 6937

ISO 7-bit or 8-bit codeset for text communication using public communication networks, private communication networks, or interchange media such as magnetic tapes and disks.

ISO8859–*

ISO 8-bit single-byte codesets. The asterisk (*) represents a number indicating the part of the associated ISO standard. For example, the ISO8859–1 codeset conforms to ISO 8859 Part 1, Latin Alphabet No. 1, which defines 191 graphic characters covering the requirements of most Western European languages.

L10N

See *localization*

langinfo database

A collection of information associated with the numeric, monetary, date and time, and messaging parts of a locale.

local language

See *native language*

locale

A set of data and rules that supports a particular combination of native (local) language, cultural data, and codeset. Also called language table.

See also *coded character set*, *cultural data*, *langinfo database*, *localization*

localization

The process of providing language- or culture-specific information for computer systems. Some of these requirements are addressed by locales. Other requirements are addressed by translations of program messages, provision of appropriate fonts for printers and display devices, and, in some cases, development of additional software. L10N is sometimes used as an abbreviation for localization.

See also *internationalization*, *locale*

message catalog

A file or storage area external to the program code that contains program messages, command prompts, and responses to prompts for a particular native language, territory, and codeset.

multibyte character

See *character*

native language

A computer user's spoken or written language, such as English, French, Japanese, or Thai.

process code

The encoding format used for manipulating data inside programs.

Contrast with *file code*

radix character

The character that separates the integer part of a number from the fractional part.

sign extension

The high bit of the value in the small data type is used to fill in bits that remain when the value is converted to the larger data type for comparison. For example, if `s[0]` is the value 0x8e, sign extension would cause it to be treated as 0xffffff8e.

string

See *character string*

territory

The geographic area, usually defined by a political entity such as nation or state, with particular cultural differences that must be accommodated in localization; for example, the currency or language of a territory.

UCS

See *ISO 10646*

Unicode
A standard that defines encoding for characters in most native languages. The Unicode standard specifies a Universal Character Set (UCS) and defines many thousands of characters, including a private use area for vendor defined characters. "Unicode" originally referred to encoding that was limited to the UCS-2 (16–bit) encoding defined by the ISO 10646 standard. The Unicode standard now encompasses UCS-4 (32–bit) encoding and defines a number of universal transformation formats (UTFs) for use with byte-oriented protocols that process data files.

See also *coded character set*, *ISO 10646*

Universal Character Set
See *ISO 10646*

wide character
An integral type that is large enough to hold any member of the extended execution character set. In program terms, a wide character is an object of type `wchar_t`, which is defined in the `/usr/include/stddef.h` (for conformance to X/Open specifications) and `/usr/include/stdlib.h` (for conformance to the ANSI C standard) header files. Although the file locations where the `wchar_t` data type is defined are determined by standards organizations, its definition is implementation specific. For example, implementations that support only single-byte codesets might define `wchar_t` as a byte value. On Tru64 UNIX systems, `wchar_t` is a 4-byte (32-bit) value.

The null wide character is a `wchar_t` value with all bits set to zero (0).

wide-character string
A contiguous sequence of wide characters that is terminated by and includes the null wide character. A wide-character string is an array of type `wchar_t`.

See also *character string*, *wide character*

worldwide portability interface (WPI)
Functions that allow programmers to create applications that support single-byte or multibyte codesets. WPI functions are similar to the C language interface, but WPI uses wide characters.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y