

Digital UNIX

Writing Software for the International Market

Part Number: AA-Q0R4C-TE

March 1996

Product Version: Digital UNIX Version 4.0 or higher

This guide provides an overview of writing international software and discusses using the tools provided on Digital UNIX (formerly known as DEC OSF/1) systems.

© Digital Equipment Corporation 1994, 1996
All rights reserved.

The following are trademarks of Digital Equipment Corporation: ALL-IN-1, Alpha AXP, AlphaGeneration, AXP, Bookreader, CDA, DDIS, DEC, DEC Ada, DEC Fortran, DEC FUSE, DECnet, DECstation, DECsystem, DECterm, DECUS, DECwindows, DTIF, Massbus, MicroVAX, OpenVMS, POLYCENTER, Q-bus, TruCluster, ULTRIX, ULTRIX Mail Connection, ULTRIX Worksystem Software, UNIBUS, VAX, VAXstation, VMS, XUI, and the Digital logo.

Open Software Foundation, OSF, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. X/Open is a trademark of X/Open Company Limited. Adobe, PostScript, and Display PostScript are registered trademarks of Adobe Systems, Inc.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii).

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

Digital conducts its business in a manner that conserves the environment and protects the safety and health of its employees, customers, and the community.

Contents

About This Manual

1 Introduction

1.1	Language	1-1
1.2	Cultural Data	1-2
1.3	Character Sets	1-2
1.4	Localization	1-3
1.4.1	Collating Sequence	1-3
1.4.2	Character Classification	1-4
1.4.3	Case Conversion	1-4
1.4.4	Language Information	1-4
1.4.5	Message Catalogs	1-4
1.5	Language Announcement	1-5
1.6	Terms and Definitions	1-5
1.6.1	Characters and Strings	1-5
1.6.2	Portable Character Set	1-6

2 Developing Internationalized Software

2.1	Using Codesets	2-2
2.1.1	Ensuring Data Transparency	2-5
2.1.2	Using In-Code Literals	2-5
2.1.3	Manipulating Multibyte Characters	2-7
2.1.4	Converting Between Multibyte-Character and Wide-Character Data	2-7

2.1.5	Rules for Multibyte Characters in Source and Execution Codesets	2-9
2.1.6	Classifying Characters	2-10
2.1.7	Converting Characters	2-11
2.1.8	Comparing Strings	2-12
2.2	Handling Cultural Data	2-13
2.2.1	The langinfo Database	2-13
2.2.2	Querying the langinfo Database	2-14
2.2.3	Generating and Interpreting Date and Time Strings That Observe Local Customs	2-14
2.2.4	Formatting Monetary Values	2-16
2.2.5	Formatting Numeric Values in Program-Specific Ways ...	2-16
2.2.6	Using the langinfo Database for Other Tasks	2-17
2.3	Handling Text Presentation and Input	2-17
2.3.1	Creating and Using Messages	2-17
2.3.2	Formatting Output Text	2-19
2.3.3	Scanning Input Text	2-20
2.4	Binding a Locale to the Run-Time Environment	2-21
2.4.1	Binding to the Locale Set for the System or User	2-22
2.4.2	Changing Locales During Program Execution	2-22
3	Creating and Using Message Catalogs	
3.1	Creating Message Text Source Files	3-2
3.1.1	General Rules	3-4
3.1.2	Message Sets	3-6
3.1.3	Message Entries	3-8
3.1.4	Quote Directive	3-10
3.1.5	Comment Lines	3-11
3.1.6	Style Guidelines for Messages	3-12

3.2	Extracting Message Text from Existing Programs	3-15
3.3	Editing and Translating Message Source Files	3-18
3.4	Generating Message Catalogs	3-18
3.4.1	Using the mkcatdefs Command	3-20
3.4.2	Using the gencat Command	3-22
3.4.3	Design and Maintenance Considerations for Message Catalogs	3-22
3.5	Displaying Messages and Locale Data Interactively or from Scripts	3-26
3.6	Accessing Message Catalogs in Programs	3-28
3.6.1	Opening Message Catalogs	3-28
3.6.2	Closing Message Catalogs	3-34
3.6.3	Reading Program Messages	3-34
4	Using Screen-Handling Routines for Character-Cell Terminals	
4.1	Handling Wide-Character Data with curses Routines	4-1
4.1.1	Writing a Wide Character to a curses Window	4-2
4.1.1.1	Add Wide Character (Overwrite) and Advance Cursor	4-2
4.1.1.2	Insert Wide Character (no Overwrite) and Do Not Advance Cursor	4-3
4.1.2	Writing a Wide-Character String to a curses Window	4-4
4.1.2.1	Add Wide-Character String (Overwrite) and Do Not Advance Cursor	4-4
4.1.2.2	Add Wide-Character String (Overwrite) and Advance Cursor	4-5
4.1.2.3	Insert Wide-Character String (no Overwrite) and Do Not Advance Cursor	4-7
4.1.3	Removing a Wide Character from a curses Window	4-8
4.1.4	Reading a Wide Character from a curses Window	4-9
4.1.5	Reading a Wide-Character String from a curses Windows	4-10
4.1.5.1	Reading Wide-Character Strings with Attributes	4-10

4.1.5.2	Reading Wide-Character Strings Without Attributes .	4-11
4.1.6	Reading a String of Characters from a Terminal	4-12
4.1.7	Reading or Queuing a Wide Character from the Keyboard	4-13
4.1.8	Converting Formatted Text in a curses Window	4-14
4.1.9	Printing Formatted Text on a curses Window	4-15
4.2	Drawing Ruled Lines in a DECterm Window	4-16
4.2.1	Drawing Ruled Lines in a Pattern	4-17
4.2.2	Erasing Ruled Lines in a Pattern	4-19
4.2.3	Erasing All Ruled Lines in an Area	4-20
4.2.4	Interaction of Ruled Lines and Other DECterm Escape Sequences	4-21
4.2.5	Determining if the DECterm Device Setting Supports Ruled Lines	4-24
4.3	DECterm Programming Restrictions	4-24
4.3.1	Downline Loadable Characters	4-24
4.3.2	DRCS Characters	4-24

5 Creating Internationalized X, Xt, and Motif Applications

5.1	Using Internationalization Features in the X Toolkit Intrinsic	5-2
5.1.1	Establishing a Locale with Xt Functions	5-2
5.1.2	Using Font Set Resources with Xt Functions	5-3
5.1.3	Filtering Events During Text Input with Xt Functions ...	5-3
5.1.4	Including the Codeset Component of Locales with Xt Functions	5-3
5.2	Using Internationalization Features of the OSF/Motif and DECwindows Motif Toolkits	5-3
5.2.1	Setting Language in a Motif Application	5-4
5.2.2	Using Compound Strings and the XmText, XmTextField, and DXmCSText Widgets	5-4
5.2.3	Internationalization Features of Widget Classes	5-6

5.3	Using Internationalization Features in the X Library	5-7
5.3.1	Using the X Library to Manage Locales	5-7
5.3.2	Displaying Text for Different Locales	5-10
5.3.2.1	Creating and Manipulating Font Sets	5-11
5.3.2.2	Obtaining Metrics for Font Sets	5-13
5.3.2.3	Drawing Text with Font Sets	5-13
5.3.2.4	Handling Text With the X Output Method	5-16
5.3.2.5	Converting Between Different Font Set Encodings ...	5-17
5.3.3	Handling Interclient Communication	5-18
5.3.4	Handling Localized Resource Databases	5-20
5.3.5	Handling Text Input with the X Input Method	5-21
5.3.5.1	Opening and Closing an Input Method	5-21
5.3.5.2	Querying Input Method Values	5-23
5.3.5.3	Creating and Using Contexts for an Input Method ...	5-26
5.3.5.4	Providing Preediting Callbacks for the On-the-Spot Input Style	5-29
5.3.5.5	Filtering Events for an Input Method	5-33
5.3.5.6	Obtaining Composed Strings from the Keyboard	5-34
5.3.5.7	Handling Failure of the Input Method Server	5-36
5.3.6	Using X Library Features: A Summary	5-37

6 Using Internationalized Software

6.1	Working in a Multilanguage Environment: Introduction	6-1
6.2	Setting Locale and Language	6-2
6.3	Selecting Keyboard Type	6-5
6.3.1	Determining Keyboard Layout	6-6
6.4	Determining Input Method	6-7
6.5	Determining the Input Mode Switch State	6-10
6.6	Setting Parameters in the .Xdefaults File	6-11
6.7	Defining the Search Path for Specialized Components	6-12

6.8	Using Terminal Interface Features for Asian Languages	6-13
6.8.1	Converting Between Application and Terminal Codesets .	6-15
6.8.2	Command Line Editing That Supports Multibyte Characters	6-16
6.8.3	Kana-Kanji Conversion: Customization of Japanese Input Options	6-18
6.9	Setting Up and Using User-Defined Character Databases	6-23
6.9.1	Creating User-Defined Characters	6-25
6.9.1.1	Working on the credit User Interface Screen	6-27
6.9.1.2	Editing Font Glyphs	6-30
6.9.2	Creating UDC Support Files That System Software Uses	6-42
6.9.3	Processing UDC Fonts for Use with DECwindows	6-44
6.9.3.1	Using fontconverter Command Options	6-45
6.9.3.2	Controlling Output File Format	6-48
6.10	Setting Up and Using the Chinese Phrase Input Method	6-49
6.10.1	Enabling the SIM Service	6-50
6.10.2	Creating and Maintaining a Chinese Phrase Database ...	6-51
6.10.3	Using a Chinese Phrase Database	6-56
6.10.3.1	Phrase Input Supported Through the SIM Service ...	6-56
6.10.3.2	Phrase Input in the DECwindows Motif Environment	6-58
6.11	Modifying the Database Location Configuration File	6-58
6.12	Using Printer Interface Features That Support Local Languages	6-60
6.12.1	Print Filters for Local Language Printers	6-61
6.12.2	Support for Local Language Printers in /etc/printcap	6-62
6.12.3	Enhancements to the lprsetup Command	6-67
6.12.4	Printing Commands and Printer Daemon	6-69
6.12.5	Font Handling for PostScript Printers	6-70
6.12.5.1	Choosing Fonts for Different Locales	6-70

6.12.5.2	Setting Up Print Queues With the pfsetup Command	6-74
6.12.5.3	Downloading Fonts to the DEClaser 1152	6-74
6.12.5.4	Downloading Fonts to the DEClaser 5100	6-75
6.13	Using Mail in a Multilanguage Environment	6-76
6.13.1	The sendmail Utility	6-76
6.13.2	The mailx Command and MH Commands	6-76
6.13.3	The comsat Server	6-78
6.14	Applying Sort Orders to Non-English Characters	6-79
6.15	Processing Reference Pages in Languages Other Than English	6-80
6.15.1	The nroff Command	6-80
6.15.2	The tbl Command	6-82
6.15.3	The man Command	6-83
6.16	Converting Data Files from One Codeset to Another	6-83
6.17	Miscellaneous Information for Base System Commands	6-85
6.18	Using Language Support Enhancements for DECwindows Motif Applications	6-86
6.18.1	Tuning the X Server for Ideographic Languages	6-86
6.18.2	Using Font Renderers for Multibyte PostScript Fonts	6-89
6.18.2.1	Setting Up the Font Renderer for Double-Byte PostScript Fonts	6-90
6.18.2.2	Setting Up the Font Renderer for UDC Fonts	6-91
6.18.3	Changing the Language of the Start Session Window	6-92
6.18.4	Setting Fonts for Display of Local Languages	6-92
6.18.4.1	Using MwmFontSetup to Update a Private Mwm File	6-93
6.18.4.2	Accessing Local Language Fonts for Remote Displays	6-93
6.18.5	Customizing the DECterm Window for Local Languages	6-111
6.18.6	Using the CDA Viewer and Converters with Asian Language Text	6-112

7 Creating Locales

7.1	Creating a Character Map Source File for a Locale	7-1
-----	---	-----

7.2	Creating Locale Definition Source Files	7-7
7.2.1	Defining the LC_CTYPE Locale Category	7-9
7.2.2	Defining the LC_COLLATE Locale Category	7-11
7.2.3	Defining the LC_MESSAGES Locale Category	7-15
7.2.4	Defining the LC_MONETARY Locale Category	7-17
7.2.5	Defining the LC_NUMERIC Locale Category	7-20
7.2.6	Defining the LC_TIME Locale Category	7-20
7.3	Building Libraries to Convert Multibyte/Wide-Character Encodings	7-23
7.3.1	Required Methods	7-24
7.3.1.1	Writing the <code>__mbstombs</code> Method for the <code>fgetws</code> Function	7-24
7.3.1.2	Writing the <code>__mbstombs</code> Method for the <code>getwc()</code> Function	7-27
7.3.1.3	Writing the <code>__pcstombs</code> Method for the <code>fputws()</code> Function	7-32
7.3.1.4	Writing a <code>__pctomb</code> Method	7-34
7.3.1.5	Writing a Method for the <code>mblen</code> Function	7-35
7.3.1.6	Writing a Method for the <code>mbstowcs</code> Function	7-38
7.3.1.7	Writing a Method for the <code>mbtowc</code> Function	7-40
7.3.1.8	Writing a Method for the <code>wcstombs</code> Function	7-45
7.3.1.9	Writing a Method for the <code>wctomb</code> Function	7-48
7.3.1.10	Writing a Method for the <code>wcswidth</code> Function	7-51
7.3.1.11	Writing a Method for the <code>wcwidth</code> Function	7-53
7.3.2	Optional Methods	7-55
7.3.3	Building a Shareable Library to Use with a Locale	7-57
7.3.4	Creating a methods File for a Locale	7-57
7.4	Building and Testing the Locale	7-59

A Summary Tables of Worldwide Portability Interfaces

A.1	Locale Announcement	A-1
-----	---------------------------	-----

A.2	Character Classification	A-1
A.3	Case Conversion	A-3
A.4	Character Collation	A-4
A.5	Access to Data That Varies According to Language and Custom	A-4
A.6	Conversion and Format of Date/Time Values	A-5
A.7	Printing and Scanning Text	A-5
A.8	Number Conversion	A-7
A.9	Conversion of Multibyte and Wide-Character Values	A-7
A.10	Input and Output	A-9
A.11	String Handling	A-10
A.12	Codeset Conversion	A-13
A.13	Miscellaneous Functions	A-13

B Sample Locale Source Files

B.1	Character Map (charmap) Source File	B-1
B.2	Locale Definition Source File	B-8

Glossary

Index

Examples

3-1	Message Text Source File	3-2
3-2	Generating a Message Catalog Interactively	3-19
5-1	Setting Locale in an X Windows Application	5-9
5-2	Creating and Using Font Sets in an X Windows Application .	5-11
5-3	Drawing Text in an X Windows Application	5-14
5-4	Communicating with Other Clients in an X Windows Application	5-19
5-5	Opening and Closing an Input Method in an X Windows Application	5-23

5-6	Obtaining the User Interaction Styles for an Input Method ..	5-24
5-7	Creating and Destroying an Input Method Context in an X Windows Application	5-26
5-8	Using Preediting Callbacks in an X Windows Application	5-29
5-9	Filtering Events for an Input Method in an X Windows Application	5-33
5-10	Obtaining Keyboard Input in an X Windows Application	5-34
5-11	Handling Failure of the Input Method Server	5-36
6-1	Sample .Xdefaults File	6-11
6-2	Default cp_dirs File	6-59
6-3	Setting Up a Local Language Printer with lprsetup	6-67
7-1	The charmap File for a Sample Locale	7-2
7-2	Fragment from a charmap File for a Multibyte Codeset	7-5
7-3	Structure of Locale Source Definition File	7-7
7-4	LC_CTYPE Category Definition	7-9
7-5	LC_COLLATE Category Definition	7-11
7-6	LC_MESSAGES Category Definition	7-16
7-7	LC_MONETARY Category Definition	7-17
7-8	LC_NUMERIC Category Definition	7-20
7-9	LC_TIME Category Definition	7-21
7-10	The __mbstopcs_sdeckanji Method for the ja_JP.sdeckanji Locale	7-25
7-11	The __mbtopc_sdeckanji Method for the ja_JP.sdeckanji Locale	7-28
7-12	The __pcstombs_sdeckanji Method for the ja_JP.sdeckanji Locale	7-32
7-13	The __pctomb_sdeckanji Method for the ja_JP.sdeckanji Locale	7-34
7-14	The __mblen_sdeckanji Method for the ja_JP.sdeckanji Locale	7-35

7-15	The <code>__mbstowcs_sdeckanji</code> Method for the <code>ja_JP.sdeckanji</code> Locale	7-38
7-16	The <code>__mbtowc_sdeckanji</code> Method for the <code>ja_JP.sdeckanji</code> Locale	7-41
7-17	The <code>__wcstombs_sdeckanji</code> Method for the <code>ja_JP.sdeckanji</code> Locale	7-45
7-18	The <code>__wctomb_sdeckanji</code> Method for the <code>ja_JP.sdeckanji</code> Locale	7-48
7-19	The <code>__wcswidth_sdeckanji</code> Method for the <code>ja_JP.sdeckanji</code> Locale	7-51
7-20	The <code>__wctype_sdeckanji</code> Method for the <code>ja_JP.sdeckanji</code> Locale	7-54
7-21	Building a Library of Methods Used with the <code>ja_JP.sdeckanji</code> Locale	7-57
7-22	The <code>methods</code> File for the <code>ja_JP.sdeckanji</code> Locale	7-58
7-23	Building the <code>de_DE.ISO8859-1@example</code> Locale	7-59
7-24	Setting the <code>LOCPATH</code> Variable and Testing a Locale	7-60

Figures

3-1	Converting an Existing Program to Use a Message Catalog ..	3-17
4-1	Drawing Ruled Lines with the <code>DECDDLBR</code> Sequence	4-18
4-2	Bit Pattern for <code>DECDDLBR</code> Parameters	4-18
6-1	Components That Support User-Defined Characters	6-25
6-2	The <code>cedit</code> User Interface Screen	6-27
6-3	The <code>cedit</code> Font Editing Screen	6-32
6-4	Interpretation of Font Editing Screen for Sizing <code>DECwindows</code> Font	6-34
6-5	Keymap for <code>cedit</code> Functions	6-37
6-6	User Interface Screen of the <code>phrase</code> Utility	6-52

Tables

3-1	Coding of Special Characters in Message Text Source Files ..	3-4
-----	--	-----

4-1	Behavior of Standard Escape Sequences with Ruled Lines ...	4-22
5-1	Locale Announcement Functions in the X Library	5-8
5-2	X Library Functions That Create and Manipulate Font Sets .	5-11
5-3	X Library Functions That Measure Text	5-13
5-4	X Library Functions That Draw Text	5-14
5-5	X Library Functions for Output Method and Context	5-16
5-6	X Library Functions for Interclient Communication	5-18
5-7	X Library Functions That Handle Localized Resource Databases	5-20
5-8	X Library Functions That Manage Input Context (XIC)	5-29
6-1	The stty Command Options for Controlling Terminal Line Discipline	6-14
6-2	The stty Options to Explicitly Set Application and Terminal Code	6-16
6-3	The stty Options to Enable/Disable History Mode	6-17
6-4	Command Line Editing in History Mode	6-17
6-5	The stty Options to Enable and Customize Japanese Input ..	6-20
6-6	The stty Options for On-Demand Loading of UDC Support Files	6-24
6-7	The cedit Command Options	6-26
6-8	Keys for Miscellaneous Font Editing Functions	6-37
6-9	Keys for cedit Mode Switching	6-38
6-10	Keys for Fine Control of Cursor Movement	6-38
6-11	Keys for Moving Cursor to Window Areas	6-39
6-12	Keys for Drawing Font Glyphs	6-39
6-13	Keys for Editing Font Glyphs	6-40
6-14	The cgen Command Options	6-43
6-15	Options and Arguments of the fontconverter Command	6-46
6-16	Chinese Phrase Input Definitions	6-50

6-17	The stty Options Used for the SIM Service	6-51
6-18	The phrase Command Options for the VT382-D Terminal	6-52
6-19	Symbols in /etc/printcap File for Local Language Printers	6-64
6-20	Local Language Printers Supported by the lprsetup Command	6-69
6-21	Supported Codeset Conversions for English	6-84
6-22	X Server Options for Tuning the Font-Cache Mechanism	6-88
6-23	XLFD Registry Names for UDC Characters	6-92
6-24	Bitmap Fonts for Asian Locales	6-94
6-25	Bitmap Fonts for *.ISO8859-2 Locales	6-95
6-26	Bitmap Fonts for *.ISO8859-4 Locales	6-98
6-27	Bitmap Fonts for *.ISO8859-5 Locales	6-101
6-28	Bitmap Fonts for *.ISO8859-7 Locales	6-103
6-29	Bitmap Fonts for *.ISO8859-8 Locales	6-105
6-30	Bitmap Fonts for *.ISO8859-9 Locales	6-107

About This Manual

The Digital UNIX internationalization features are tools and routines that allow you to write programs for use in a number of nations. These features let you write programs with:

- An interface that appears to a nation's users as if it was designed for them
- Source code that is independent of specific native languages and customs

Audience

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

New and Changed Features

This manual contains information related to the following new software features:

- Locales and other software to support Catalan, Lithuanian, and Slovene
- Support for character processing in UCS-4 format
- Curses library routines that handle multibyte characters and also conform to the X/Open Curses CAE specification
- Support for X11R6 libraries
- Support for the Common Desktop Environment (CDE)
- Internationalization enhancements to the printing and mail subsystems
- Font renderers for use by X applications
- Multilingual Emacs editor (`mule`)

- Codeset conversion improvements to support better the multiple codesets available for Chinese and Japanese
- Functions added to the Standard C Library by the 1994 amendment to the ISO C standard

In addition, this manual includes the following documentation changes:

- Text revisions to correct errors and omissions in the previous version
- A glossary

Organization

This guide includes seven chapters, two appendixes, and a glossary.

Chapter 1	<p>Introduction</p> <p>Introduces the basic concepts and procedures for writing programs that meet the needs of international users.</p>
Chapter 2	<p>Developing Internationalized Software</p> <p>Discusses techniques for handling character sets, cultural data, and language in an application.</p>
Chapter 3	<p>Creating and Using Message Catalogs</p> <p>Explains how to extract and translate text for messages, and how to generate and access message catalogs.</p>
Chapter 4	<p>Using Screen-Handling Routines for Character-Cell Terminals</p> <p>Describes the <code>curses</code> library routines for handling wide-character data and discusses terminal-programming extensions for drawing ruled lines in a DECterm window.</p>
Chapter 5	<p>Creating a Graphical User Interface (GUI) With X and OSF/Motif Libraries</p> <p>Discusses how to use GUI programming libraries (X, OSF/Motif, and DECwindows Extensions to OSF/Motif) when writing internationalized programs.</p>
Chapter 6	<p>Using Internationalized Software</p> <p>From a programmer's perspective, discusses setup requirements for using applications in different language environments. This chapter also explains how to use Digital UNIX commands and other applications in a multilanguage working environment.</p>

Chapter 7	<p>Creating Locales</p> <p>Discusses the source files for a locale and how to process them with the <code>localedef</code> utility.</p>
Appendix A	<p>Summary Tables of Worldwide Portability Interfaces</p> <p>Lists and summarizes internationalized functions for locale initialization, character classification, case conversion, character collation, language information, date and time interpretation, printing and scanning text strings, number conversion, handling multibyte characters, input/output, and string manipulation.</p>
Appendix B	<p>Sample Locale Source Files</p> <p>Contains complete source files for the sample locale discussed in Chapter 7.</p>
Glossary	<p>Defines terms and acronyms used in this book.</p>

Related Documentation

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

- *Programmer's Guide*
- *Programming Support Tools*
- *OSF/Motif Programmer's Guide*
- *DECwindows Motif Guide to Application Programming*
- *X Window System*
- *X Window System Toolkit*
- *Programmer's Supplement for Release 5 of the X Window System, Version 11*

This book is published by O'Reilly and Associates, Inc.

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

- *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.
- *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.
- *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.

- X/Open UNIX CAE specifications, specifically:
 - *Commands and Utilities, Issue 4, Version 2* (C436)
 - *Systems Interfaces and Headers, Issue 4, Version 2* (C435)
 - *System Interface Definitions, Issue 4, Version 2* (C434)
 - *Networking Services, Issue 4* (C438)
 - *X/Open Curses, Issue 4* (C437)

The following guide is an introduction to internationalization and is available from the X/Open Company, Ltd:

- *Guide to Internationalisation*

Articles in the following technical journal cover topics related to product internationalization:

- *Digital Technical Journal*, Volume 5 Number 3 (published Summer 1993)

The *Digital Technical Journal* is published quarterly by Digital Equipment Corporation. You can order individual issues through DECdirect. Recent back issues are also available on the Internet at `gatekeeper.dec.com` in the directory `/pub/DEC/DECinfo/DTJ`.

The printed version of the Digital UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General users	G	Blue
System and network administrators	S	Red
Programmers	P	Purple
Device driver writers	D	Orange
Reference page users	R	Green

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the Digital UNIX documentation set.

Reader's Comments

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

You can send your comments in the following ways:

- Fax: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

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

`/usr/doc/readers_comment.txt`

- Mail:

Digital Equipment Corporation
UEG Publications Manager
ZK03-3/Y32
110 Spit Brook Road
Nashua, NH 03062-9987

A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Digital UNIX that you are using.
- If known, the type of processor that is running the Digital UNIX software.

The Digital 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 Digital technical support office. Information provided with the software media explains how to send problem reports to Digital.

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.
<i>file</i>	<i>Italic (slanted)</i> type indicates variable values, placeholders, and function argument names.
[] { }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or 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 shown.
cat(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, <code>cat(1)</code> indicates that you can find information on the <code>cat</code> command in Section 1 of the reference pages.
Return	In an example, a key name enclosed in a box indicates that you press that key.

Ctrl/x

This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box; for example, **Ctrl/C**.

Alt x

Multiple key or mouse button names separated by spaces indicate that you press and release each in sequence. In examples, each key in the sequence is enclosed in a box; for example, **Alt Q**.

1

Introduction

Internationalization refers to the process of developing 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 modify their behavior at run time for operation in a specific language environment. The mnemonic **I18N** is frequently used as an abbreviation for internationalization.

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

1.1 Language

An internationalized program makes no assumptions about the language of character data (text) that the program is designed to handle. The term **data** refers to data generated internally, data extracted from or written to files, and message text used for communication with the program's user.

Language has implications for processing text for such things as character handling and word ordering. Digital UNIX 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. Digital UNIX 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 UIL (User Interface Language) 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, refer to the following books:

- *X Window System Toolkit*
- *OSF/Motif Programmer's Guide*
- *DECwindows Motif Guide to Application Programming*

1.2 Cultural Data

Cultural data refers to the conventions of a geographic area or territory for such things as date, time, and currency formats.

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

1.3 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 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.

Locales do not solve all of the problems that localization must address. 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.

1.4.1 Collating Sequence

The 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 more complex rules for sorting than English. The following list shows why some English assumptions about character sorting do not apply to other languages:

- A single letter is not necessarily represented by a single character. In 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 sort orders based on two visually recognized

components (radicals, which are pictograms for elements of meaning, and the number of strokes).

Each locale contains collating sequence-information 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 Character Classification

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

1.4.3 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. Note that case is not a characteristic of all of the letters, or even of any characters, in some languages.

1.4.4 Language Information

Language information (or **langinfo database**) refers to localization data that describes the format and setting of cultural data that can vary from one locale to another. This information includes the appropriate formats and characters for date and time, currency, and numeric values.

1.4.5 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 User Interface Language (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.

1.5 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. On Digital UNIX systems, system managers can set the default values for these variables for different shell environments; refer to the *System Administration* book 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 a particular instance of localization data to the programs' operational environment. However, programs can also set these variables 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.

1.6 Terms and Definitions

This section defines terms used extensively in this guide. Less common terms are defined when they first appear.

1.6.1 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 data type `char`, 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 as used in this guide can be represented by a multibyte or single-byte value. 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 UNIX standard) and `/usr/include/stdlib.h` (for conformance to the ANSI C standard). The file locations where this data type is defined are determined by standards organizations; however, the definition itself is implementation specific. For example, implementations that support only single-byte codesets (not the case for Digital 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.

1.6.2 Portable Character Set

The Portable Character Set (PCS) is supported in both compile-time (source) and run-time (executable) environments. The PCS contains:

- The 26 uppercase letters of the English 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 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 new line.

The Portable Character Set 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 Portable Character Set. An example of a character substitution might be the British pound sign (£) for the number sign (#) that is the default.

2

Developing Internationalized Software

This chapter explains how language, codeset, and cultural differences change the way you implement basic coding operations. After reading this chapter, you will be ready to examine a complete application that applies the program development techniques that have been suggested. Such an application is provided on line in the directory `/usr/examples/i18n/xpg4demo`. Refer to the `README` document in that directory for an introduction to the application and how you can compile and run it 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, some or all of which may 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 also 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-bit, 16-bit, and so on) and binary representation.

You can satisfy the preceding requirements 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.

Digital UNIX provides the following facilities for developing internationalized software, defining localization data, and announcing specific language requirements:

- Library functions that handle extended character codes and that provide language- and codeset-independent character classification, case conversion, number format conversion, and string collation

- Library functions that let programs determine cultural and language-specific data dynamically
- 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
- An initialization function that binds a program at run time to the specified language requirements of each user

The rest of this chapter describes each of these facilities in more detail.

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

2.1 Using Codesets

In the past, most UNIX systems were based on the 7-bit ASCII codeset. However, most languages other than English 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 guide 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 ISO codesets cover the major European languages. Several of these codesets allow for the mixing of major languages within a single codeset. All ISO codesets are a superset of the ASCII codeset and therefore allow systems to support languages other than English without invalidating existing software that is not internationalized. The Digital UNIX operating system provides locales that use the ISO 8859-1 (Latin 1) and ISO 8859-7 (Latin/Greek) codesets.

Subsets that support localized variants of the operating system may include locales based on additional ISO codesets. For example, the optional language variant subsets included with Digital UNIX to support Czech, Hungarian, Polish, Russian, Slovak, and Slovene provide locales based on the ISO 8859-2 (Latin 2) codeset. Following is a complete list of ISO codesets, along with the languages that they support:

- ISO 8859-1, Latin 1
Western European languages, including Catalan

- ISO 8859-2, Latin 2
Eastern European languages
- ISO 8859-3, Latin 3
Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, and Turkish
- ISO 8859-4, Latin 4
Danish, English, Estonian, Finnish, German, Greenlandic, Lappish, Latvian, and Lithuanian
- ISO 8859-5, Latin/Cyrillic
Bulgarian, Byelorussian, English, Macedonian, Russian, Serbo-Croatian, and Ukrainian
- ISO 8859-6, Latin/Arabic
Arabic
- ISO 8859-7, Latin/Greek
Greek
- ISO 8859-8, Latin/Hebrew
Hebrew
- ISO 8859-9, Latin 5
Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, and Turkish
- ISO 8859-10, Latin 6
Danish, English, Estonian, Finnish, German, Greenlandic, Icelandic, Sami (Lappish), Latvian, Lithuanian, Norwegian, Faroese, and Swedish

Another ISO codeset supported by utilities on a standard Digital UNIX 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. Digital UNIX supplies the following codesets through subsets that support Asian languages and countries:

- eucJP (Japanese Extended UNIX Code)
- SJIS (Shift JIS)
- deckanji (DEC Kanji)
- sdeckanji (Super DEC Kanji)
- deckorean (DEC Korean)
- eucKR (Korean Extended UNIX Code)
- TACTIS (Thai API Consortium/Thai Industrial Standard)
- dechanzi (DEC Hanzi)
- dechanyu (DEC Hanyu)
- eucTW (Taiwanese Extended UNIX Code)
- big5 (BIG-5)

These codesets are supplied when you install Asian-language variant subsets of the Digital UNIX product. A specialized terminal driver and associated utilities must be available on your system to support the input and display of Asian characters at run time. These components are also supplied when you install one of the Asian-language variant subsets.

The Unicode and ISO/IEC 10646 standards specify the Universal Character Set (UCS), a character set that allows character units to be processed for all languages, including Asian languages, by using the same set of rules. Digital UNIX supports the USC-4 encoding of this character set. An application parses UCS-4 character encoding in 32-bit units.

Reference pages are available for all the character sets that Digital UNIX supports. For more information on a particular character set, refer to its reference page.

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
- Using in-code literals
- Manipulating multibyte characters

- Converting between multibyte-character and wide-character data
- Rules for multibyte characters
- Classifying characters
- Converting characters (case)
- Comparing strings

2.1.1 Ensuring Data Transparency

As discussed in Section 2.1, 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.

Another legacy of the historical dependence of UNIX systems on 7-bit ASCII character encoding is that some programs use the most significant bit 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.

2.1.2 Using In-Code Literals

When writing internationalized software, using in-code literals can cause problems. 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. The following statement represents an improvement in that it substitutes a character constant for the octal value:

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

This example still presents problems, however, because the `getchar()` function operates on bytes. The statement would not work correctly if the next character in the input stream were a multibyte value. The following statement substitutes the `getwchar()` function for the `getchar()` function. The statement works correctly with any codeset because `a` is a

member of the Portable Character Set 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. It is therefore safe for you to use any of the characters in the Portable Character Set as a character constant or in string literals. Note that non-English characters are not included in the Portable Character Set 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, execution character set, or both; or the binary value of the accented character may not be translatable from one set to the other. When source files specify non-English characters in constants, the results are undefined.

The following example shows how to construct a test for a constant that for whatever reason may be a non-English 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;           ❶  
wchar_t wchar;       ❷  
⋮  
schar = catgets(catd,NL_SETD,MSG_ID,"a"); ❸  
if (mbtowl (&wchar,schar,MB_CUR_MAX) == -1) ❹  
    error();  
if ((c = getwchar()) == wchar) ❺  
⋮
```

❶ Declares a pointer to `schar` as `char`.

❷ Declares the variable `wchar` to be of type `wchar_t`.

❸ 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 Because the locale's codeset may contain multibyte characters, tests to make sure the value contained in `schar` represents a valid multibyte character; if yes, converts it to a wide-character value and stores the results in the variable `wchar`.

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

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

Refer to Chapter 3 for more information about message catalogs and the `catgets()` function. See Section 2.1.4 for information about converting multibyte characters and strings to wide-character data that your program can process.

2.1.3 Manipulating Multibyte Characters

Digital UNIX provides all the interfaces (such as `putwc()`, `getwc()`, `fputws()`, and `fgetws()`) that are needed to support multibyte Asian codesets. Language variant subsets of the operating system must be installed to supply the locales and facilities that make this support operational. On systems where multibyte locales are not available, or are available and 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()`). The interfaces provided for multibyte support are therefore appropriate for use with all locales, not just those with multibyte characters.

2.1.4 Converting Between Multibyte-Character and Wide-Character Data

Some languages, particularly Asian languages, can be encoded as either multibyte-character or wide-character data.

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

- Multibyte characters are not represented by a fixed number of bytes per character, even in the same codeset, so 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 the disadvantages of multibyte encoding, wide-character encoding, which allocates a fixed number of bytes per 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 Digital UNIX systems, the default size for a wide character is set to 4 bytes (32 bits), a setting that optimizes performance for the Alpha processor.

Library routines that print, scan, input, or output text have the capability of automatically converting 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 apply a locale-independent format to a record retrieved from a data file and 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\tS\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);

```

⋮

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

2.1.5 Rules for Multibyte Characters in Source and Execution Codesets

You should be aware that 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 both set variants must observe the following rules:

- 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 also 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
??=	#
??([
??/	\
??'	^
??<	{
??)	}
??!	
??>	}
??-	~

2.1.6 Classifying Characters

Another feature of program operation that depends on the codeset 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 codeset ISO 8859-1:1987, 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 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 `type` information in the user's locale and are independent of the language and codeset. Refer to Section A.2 for a complete list and description of character classification functions.

2.1.7 Converting Characters

You can do case conversion of ASCII characters with statements like the following ones, which convert 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 they:

- Assume ASCII-coded character values
- Do not check for valid input

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 where characters are defined. The functions return the argument unchanged if input is invalid. Refer to Section A.3 for more detailed discussion of case conversion functions.

2.1.8 Comparing Strings

UNIX systems have always provided 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);
⋮
```

Certain languages, however, require collation algorithms that must make multiple passes through the codeset. 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 thus 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:

- `wscoll()`, which performs the same operation as `strcmp()`, except that it operates on wide characters and uses locale-specific collating information
- `wcsxfrm()`, which 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 `wscmp()` function

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

2.2 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 are used in decimal numbers with exactly the opposite meaning.
- In the United States, the date October 7, 1986 is represented as 10/7/86, whereas in the United Kingdom, the same date is represented as 7/10/86. This example indicates that cultural data items can vary 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 7.10.86 rather than as 7/10/86.
- Currency symbols can vary both in terms of 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.

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.

2.2.1 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 on Digital UNIX systems 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
- 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)

2.2.2 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 header file `/usr/include/langinfo.h`. The function returns a pointer to the string with the associated value for *item*. The following example shows 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);
```

2.2.3 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()` function, which makes indirect use of the `langinfo` database.

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); ❸
:
:
strftime(buf, size, "%x", tm); ❹
puts(buf); ❺
:
:

```

- ❶ Binds the program at run time to the locale set for the system or individual user.
- ❷ Calls the `time()` subroutine to return the time value, relative to Coordinated Universal Time, to the `clock` variable.
- ❸ 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.
- ❹ 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 representation appropriate for the locale.

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

The following example shows how to use `strftime()` and `nl_langinfo()` in combination to generate a date and time string. Assume that the same calls to the `setlocale()`, `time()`, and `localtime()` interfaces have been made here as shown in the preceding example. The only difference is that a call to `nl_langinfo()` 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, the reverse of the operation performed by `strftime()`, you can use the `strptime()` function. The

`strptime()` supports a number of conversion specifiers that behave in a locale-dependent manner.

2.2.4 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); 1
```

¹ This statement formats the double-precision floating-point value contained in the variable `value`. The argument `"%n"` is the format specification that is replaced by the format defined in the run-time locale. The results are returned to the array `buf`, whose maximum length is contained in the variable `size`.

The `money` program demonstrates how the `strfmon()` function works. The source file for this sample program is available in the `/usr/i18n/examples/money` directory.

2.2.5 Formatting Numeric Values in Program-Specific Ways

You may want to perform your own conversions of numeric quantities, monetary or otherwise, by using 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 structure `lconv`, 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

2.2.6 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 `scanf()`, `printf()`, and `wcstod()` functions determine the appropriate radix character from information in the `langinfo` database.

2.3 Handling Text Presentation and Input

The language of the program user affects:

- The way program messages are defined and accessed
- How the program presents output text
- How the program processes input text

These considerations are discussed in the following sections.

2.3.1 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 a native-language message system that contains a definition of message text source files, the `genmsg` command to generate message catalogs from these source files, and a set of Standard C Library functions to retrieve individual messages from one or more catalogs at run time.

The following example shows 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 header file `/usr/include/locale.h`, which declares the `setlocale()` function and associated constants and variables.
- 3 Includes the header file `/usr/include/nl_types.h`, which declares the `catopen()`, `catgets()`, and `catclose()` functions.
- 4 Includes the program-specific header file, `prog_msg.h`, which sets constants to identify the message set (SETN) and specific messages (HELLO_MSG being one) that are used by this program module.

A message catalog can contain one or more message sets and 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 to settings for the user's locale environment variables.

The locale name set for the `LC_MESSAGES` variable is the locale used by the `catopen()` and `catgets()` functions in this example. Typically, the system manager or user sets only the `LANG` environment variable to a particular locale name and the same locale name is used for `LC_MESSAGES`.

- 7] Calls the `catopen()` function to open the message catalog named `prog.cat` 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. Note that the `catgets()` function allows one message translation to be held within the program source. This is the translation that will be used in the event that the program cannot get the message from the message catalog.

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

Refer to Chapter 3 for information about creating and using message catalogs.

2.3.2 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 format specifiers for the `printf()` routine have been extended so that format conversion applies to the *n*th argument in an argument list rather than 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 equivalent of:

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

2.3.3 Scanning Input Text

The string construction issues that are discussed for output text in Section 2.3.2 also apply to input text. For example, in different countries there are different conventions that apply to the order in which users specify the elements of a date or there are differences in characters that are input to delimit parts of monetary or other numeric strings. Therefore, the `scanf()` family of functions also support extended format conversion specifiers to 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 English format (mm/dd/yy) 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 where users use the format mm/dd/yy 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. For example:

- British English (dd/mm/yy):

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

- German (dd.mm.yy)

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

2.4 Binding a Locale to the Run-Time Environment

For an internationalized program to operate correctly, it 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:

- 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 can be one of the following:

- LC_ALL to use, change, or query all portions of the user's locale
- LC_CTYPE to use, change, or query the portion of the user's locale that classifies characters

- `LC_COLLATE` to use, change, or query the portion of the user's locale that specifies character collation order
- `LC_MESSAGES` to use, change, or query the portion of the user's locale that specifies yes/no responses and program messages
- `LC_MONETARY` to use, change, or query the portion of the user's locale that specifies special characters used in monetary values
- `LC_NUMERIC` to use, change, or query the portion of the user's locale that specifies the characters used for decimal point and thousands separator
- `LC_TIME` to use, change, or query the portion of the user's locale that specifies names and abbreviations for days of the week and 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 `()` to bind the program at run time to the locale name set for *category* by the system manager or user
- A locale name to change the locale that may already be set for *category*
- `NULL` to find out the locale name currently set for *category*

2.4.1 Binding to the Locale Set for the System or User

Typically, the system manager or user sets the `LANG` environment variable to the name of a locale; setting the `LANG` variable automatically sets all portions, or categories, of the locale to the same locale name. On occasion, system managers or individual users may set different locale categories to different locale names. Usually, internationalized programs contain the following call, which initializes all locale categories in the program to settings already in effect for the user:

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

2.4.2 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

shows how to call `setlocale()` when you want to explicitly initialize or reinitialize all locale categories to the same locale name:

```
:\n\nnl_catd catd; 1\nchar buf[BUFSIZ]; 2\n\n:\nsetlocale(LC_ALL, ); 3\ncatd = catopen(CAT_NAME, 0); 4\n\n:\nprintf(catgets(catd, NL_SETD, LOCALE_PROMPT_MSG,\n               \"Enter locale name: \")); 5\ngets(buf); 6\nsetlocale(LC_ALL, buf); 7\n\n:\n
```

- 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 constant `BUFSIZ`, 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; 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 identifier `LOCALE_PROMPT_MSG` 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, like the `langinfo` database, is a file of localization data that programs can access. The difference between the two sets of localization data is that 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. Message catalogs, on the other hand, 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, which include a graphical user interface, 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 have the additional option of using 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 and explains how to:

- Create, edit, extract, and translate message text source files
- Generate message catalogs
- Access message catalogs interactively and from scripts
- Access message catalogs from programs

Refer to the *OSF/Motif Programmer's Guide* for information on handling text with Motif routines in internationalized applications. Refer to *X Window System* for information about using text from message catalogs with X routines. X and Motif programmers can also find in Section 3.1.6 of this chapter, some guidelines that apply to text that will be translated, regardless of the method used to retrieve and display it.

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 that focus on particular kinds of file entries and processing operations. Example 3–1 shows 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
:
:
$set MSGInfo 4
I_COM_NEWEMP "New employee" 5
I_COM_YN_DELETE "Do you want to delete this record?" 5
I_COM_YN_MODIFY "Do you want to modify this record?" 5
I_COM_YN_REPLACE "Are these the changes you want to make?" 5
:
:
I_SCR_IN_DATE_FMT "%2$d/%3$d/%1$d" 6
$set MSGString 4
$
$ One-character commands.
$S_COM_CREATE "c" 7
S_COM_DELETE "d" 7
S_COM_EXIT "e" 7
:
:
S_COM_LIST_TITLE "Badge Name Surname \
CC DOJ\n" 8
S_COM_LIST_LINE "-----\n" 8
:
:
$
$ If surname comes before first name, "y" should be specified.
$
S_SCR_SNAME1ST "n" 9
:
:

```

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

- 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 This line specifies the quote character used to delimit message text. Section 3.1.4 discusses quote directives.
- 4 These lines define identifiers that 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). Refer to 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. Refer to 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 entry allows translators to vary the order in which users are prompted to input date elements. Note that you frequently use message entries to allow format control.
- 7 Message entries such as these define word abbreviations, which often need special attention to preserve uniqueness from one language to another.
- 8 Message entries also define 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.
- 9 In the `xpg4demo` program, you can change the order of first and last name (surname). This message entry defines a constant whose value controls how the program orders name fields.

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:

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 message text source files. A message text source file (.msg file) comprises 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. 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 shown 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

Table 3–1: Coding of Special Characters in Message Text Source Files (cont.)

Description	Symbol	Coding Sequence
Backspace	BS	\b
Carriage return	CR	\r
Form feed	FF	\f
Backslash	\	\\
Octal value	ddd	\ddd ^a
Hexadecimal value	dddd	\xdddd ^b

^aThe escape sequence \ddd consists of a backslash followed by one, two, or three octal digits that specify the value of the desired character.

^bThe 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. Note that the hexadecimal coding sequence is an extension to X/Open UNIX CAE specifications and therefore 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 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; you are therefore free to use blank lines wherever you choose 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 variable `set_id` can be one of the following:

- A number in the range [1 - {NL_SETMAX}]

The `NL_SETMAX` constant is defined in the file `/usr/include/limits.h`. 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 `genmsg` command to create a message catalog.

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

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

Any characters following the set identifier are treated as a comment.

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 file `/usr/include/nl_types.h`. 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 variable *n* 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:

```
$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 directory `/usr/lib/nls/msg`. 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 file `/usr/include/limits.h`. 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 file `/usr/include/nl_types.h`. 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 `{NL_TEXTMAX}` bytes. The constant `NL_TEXTMAX` is defined in the `/usr/include/limits.h` file.

The rest of this section discusses entries that delete specific messages from an existing message catalog. Refer to 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. Use only `gencat` to process the file if you are not replacing all messages in a set. The `mkcatdefs` utility 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.

The following example shows message text source input that could be specified to the `genccat` command to delete message 5 from message set 2:

```
$set 2  
5
```

If this source input were preprocessed by the `mkccatdefs` command, the addition of the `delset` directive would result in all messages in set 2 being deleted from the message catalog:

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

3.1.4 Quote Directive

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

```
$quote[ c ]
```

The `c` 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 `c` 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 \$, followed by a space or tab, is treated as a comment. Neither the `mkcatdefs` nor the `gencat` commands further interpret the line.

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"
:
:
```

Digital UNIX 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 English, you need to keep the following information in mind:

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

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

- If you must limit a text string to one line (for example, 80 characters), make sure the English 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 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 a particular character input method.
 - Do not embed text in a graphic. When 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 very different from English. For these reasons:
 - Do not create a message at run time by concatenating different kinds of strings; for example, 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 words and their counterparts in other languages. For example, you can create a negative statement in English by inserting a text variable that contains the word “not” into a verb phrase. The message could not be translated to French, 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.
- Create messages that are complete sentences; in particular, do not start messages (other than imperatives where the subject “you” is understood) with a verb.

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.

If your message is constructed of a facility identifier, followed by 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 is constructed, the facility identifier that appears with the message, and the kind of component (server, compiler, utility, and so forth) the identifier represents. Refer to Section 3.1.5 for information about adding comments to message source files.

- 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 2.3.2 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 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.
 - Avoid using contractions. The contracted word is very often the word “not,” which you do not want the translator to miss.
 - Avoid using most abbreviations, particularly terms, such as pkt, msg, tbl, ack, and max, that programmers commonly use in variable names and code comments. 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 abbreviations for names of standards, protocols, and so forth that appear in commercial literature.
 - Use words only in grammatically correct form. English 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 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. It is probable that these words either have no

equivalent in another language or would be misinterpreted. For example, the message “Server hang” may be meaningful to English 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.

3.2 Extracting Message Text from Existing Programs

If you have an existing program that you want to internationalize, Digital UNIX 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
<code>extract</code> 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 <code>catgets</code> function.
<code>strextact</code> command	Performs string extraction operation in batch.
<code>strmerge</code> command	Reads strings from the message file produced by <code>strextact</code> and, in the program source, replaces those strings with calls to the <code>catgets</code> function.

Consider the following call:

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

You can use the `extract` command, or the `strextact` command followed by the `strmerge` command, to:

- 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:

```
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 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 file `/usr/lib/nls/patterns`.

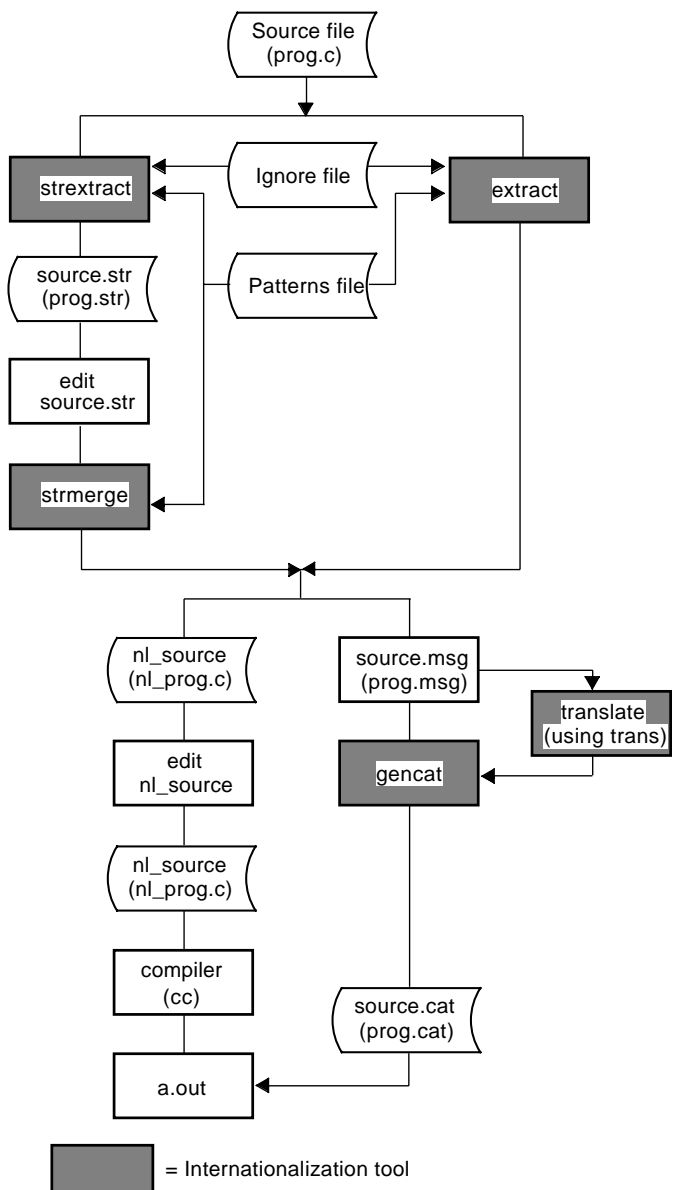
- An optional ignore file

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 the `langinfo` database.

Figure 3-1 shows the files and tools that help you change an existing program to use a message catalog.

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



ZK-0045U-R

For detailed instructions on using the `extract`, `strextract`, and `strmerge` commands, see the `extract(1)`, `strextract(1)`, `strmerge(1)`, and `patterns(4)` reference pages.

3.3 Editing and Translating Message Source Files

You can use any text editor to edit message text source files, provided that:

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

The `ed`, `ex`, and `vi` editors satisfy the second of the preceding requirements. Localized software subsets may also include enhanced versions of additional editors, such as Emacs, that can handle 8-bit and multibyte characters.

The standard Digital UNIX 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 on the `trans` command, refer to the `trans(1)` reference page. Refer to Section 3.1.5 for examples of comments that should be included in message text source files to ensure that messages are correctly translated.

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

```
% cd /usr/examples/xpg4demo/src
% ls *.msg
:
```

3.4 Generating Message Catalogs

The `genmsg` 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 that 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 root name to use for the header file that 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 the name of the header file it created to standard output. 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 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, this version of the catalog could be moved to the default directory `/usr/lib/nls/msg/fr_FR.ISO8859-1` or stored in a directory that is application specific.

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

Refer to the file `/usr/examples/xpg4demo/src/Makefile` 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 `genmsg` command as shown in Example 3-2 or use the `>` redirection specifier to print the output to a file
- Creates a header file that maps numbers that will identify 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 set and message identifiers. Symbolic identifiers improve the readability of your program source code and make the code independent of the order in which set and message 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 recompiling it with the new header file.

The option of defining symbolic identifiers for message sets and catalogs is not specified by the X/Open UNIX standard, so you should not assume that the `mkcatdefs` command is available on all operating systems that

conform to this standard. However, the source text message file and program header file produced by the `mkcatdefs` command should be portable among systems that conform to the X/Open UNIX standard.

The `mkcatdefs` command does not refer to the header file for an existing message catalog to map symbolic identifiers to the numbers assigned when that catalog was created. The command assigns numbers to symbols 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, it is your responsibility to ensure correct mapping between the symbols you specify in the source input to the `mkcatdefs` command and 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 to prevent 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 will need a complete header file for recompiling 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, it 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 will 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.

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 Digital 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.

Refer to the `gencat(1)` reference page for more details on `gencat` command syntax and use.

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. Adding and deleting message sets and entries in an existing catalog can, if not done carefully, change the ordinal position specifiers that identify messages occurring after the point in the file where a modification is made. Consider a message whose English 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)

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 on a per-module basis; in other words, you can synchronize any particular program module with the new version of a message catalog by recompiling with the new header file generated by the `mkcatdefs` utility.

The ability to recompile 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 descriptions of a few 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 per 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. Software, such as the Revision Control System (RCS) and the Source Code Control System (SCCS) are helpful in managing 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, without 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 both to the user's process and the system-wide open file table. There is a system-wide and process-specific maximum for the number of files that can be open simultaneously, and these limits vary from one system to another. On Digital 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 per program source, single catalog for application

- Advantages

The same advantages exist as discussed for the preceding option, plus the single catalog design eliminates any problems associated with numerous open operations if you port your application to systems other than Digital 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 per source file, for example, one set for errors, one set for informational displays, 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 only some of them do. The `mkcatdefs` and `gencat` commands concatenate input source files together 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.

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.

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.

- **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, you could generate one message catalog with a single message text source file where error messages are defined. This source file could 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 messages no longer used by using message deletion

directives (which remove messages from the catalog without changing the position numbers for subsequent messages in the same set).

3.5 Displaying Messages and Locale Data Interactively or from Scripts

After a message catalog is created, you may want to 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 three commands that display messages in a message catalog and one command that displays information for the current locale.

- `dspcat`

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 catalog named `xpg4demo.cat`:

```
% 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 that reformats the output stream for an entire catalog or message set so that it can be piped to the `genccat` 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 `genccat` 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 path name exceeded 255 bytes.\n"
4      "du: %s\n\
The generated path name 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`

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%
```

- `locale`

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][eE][sS]"
noexpr="^[nN][oO]"
yesstr="yes:y:Y"
nostr="no:n:N"
```

```
% locale yesexpr
^[yY]|[yY][eE][sS])
```

- `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
```

Refer to the `dspcat(1)`, `dspmsg(1)`, `locale(1)`, and `printf(1)` reference pages 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 the file
- `catclose()` to close the file
- `catgets()` to retrieve messages

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, if successful, the `catopen()` function returns a message catalog descriptor to the variable `MsgCat`. 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 two 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 `NL_CAT_LOCALE`, `catopen()` searches for a message catalog that supports the locale set for the `LC_MESSAGES` environment variable. If you specify 0, `catopen()` 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 current X/Open 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 X/Open 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:

`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:
 - 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:\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, it is worthwhile to 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:

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 Digital 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:

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

2. `xpg4demo.cat`

Finally, if neither `LC_MESSAGES` nor `NLSPATH` is set, `catopen()` would search for:

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

The `catopen()` function does not return an error status when the specified 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, given that 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, including its error-handling behavior, refer to the `catopen(3)` reference page.

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 path `/usr/lib/nls/msg/%L/%N`. If a program runs with an effective user ID of `root`, you must therefore do one of the following:

- Install all message catalogs used by the program in locale directories identified as `/usr/lib/nls/msg/%L`.
- 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 `root`. 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 four 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 this string when the program cannot retrieve the specified message from a catalog, usually because the catalog was not found or opened.

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);
⋮
```

Refer to the `catgets(3)` reference page for more information about the `catgets()`

Note

The `gettext()` 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, refer to the `gettext(3)` reference page.

4

Using Screen-Handling Routines for Character-Cell Terminals

This chapter discusses:

- Enhancements made to the `curses` library to support wide-character format
- Enhancements made to escape sequences used in terminal programming to support drawing of ruled lines

4.1 Handling Wide-Character Data with `curses` Routines

The `curses` library provides functions for developing user interfaces on character-cell terminals. The Digital UNIX `curses` library provides functions for processing single-byte and multibyte characters. The recommended functions for handling multibyte characters in wide-character or complex-character format conform to Version 4 of the X/Open Curses CAE specification and supercede those specified by the *System V Multi-National Language Supplement* (MNLS).

This chapter summarizes the `curses` functions and macros that process characters and character strings from the screen or keyboard. Tables in each section note if there is more than one `curses` interface available to perform the same operation, but only one handles wide-character or complex-character format and conforms to the X/Open Curses CAE specification. In such cases, 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 do; then use the `man` or `xman` command to display the reference page for the chosen interface.

Note

Some `curses` interfaces overwrite existing characters on the `curses` window. Only the interfaces 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 interfaces write extra blank characters to remove partial characters. For example, if the English character `a` overwrites the first column of a 2-column Chinese character, the second column of the Chinese character is overwritten with a blank.

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

4.1.1 Writing a Wide Character to a `curses` Window

The following sections discuss different categories of routines that add or insert individual wide characters on a `curses` window. There are two categories of routines and these differ, partly due to whether the cursor is advanced after the wide character is written.

4.1.1.1 Add Wide Character (Overwrite) and Advance Cursor

The functions and macros in the following table add a wide character, along with 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 interface depends on whether you need to:

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

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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>add_wch</code>	<code>addch</code> , <code>addwch</code>	Window: default Position: current Screen refresh: no
<code>wadd_wch</code>	<code>waddch</code> , <code>waddwch</code>	Window: specified Position: current Screen refresh: no
<code>mvadd_wch</code>	<code>mvaddch</code> , <code>mvaddwch</code>	Window: default Position: specified Screen refresh: no
<code>mvwadd_wch</code>	<code>mvwaddch</code> , <code>mvwaddwch</code>	Window: specified Position: specified Screen refresh: no
<code>echo_wchar</code>	<code>echowchar</code>	Window: default Position: current Screen refresh: yes
<code>wecho_wchar</code>	<code>wechowchar</code>	Window: specified Position: current Screen refresh: yes

4.1.1.2 Insert Wide Character (no Overwrite) and Do Not Advance Cursor

The following functions and macros 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 interfaces 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:

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

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

4.1.2 Writing a Wide-Character String to a curses Window

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

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

The functions and macros in the following table add a wide-character string, along with character attributes, to a window. These interfaces:

- 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 around to the next line

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

- 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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>add_wchstr</code>	<code>addwchstr</code>	Number of characters: all Window: default Position: current
<code>add_wchnstr</code>	<code>addwchnstr</code>	Number of characters: specified Window: default Position: current
<code>wadd_wchstr</code>	<code>waddwchstr</code>	Number of characters: all Window: specified Position: current
<code>wadd_wchnstr</code>	<code>waddwchnstr</code>	Number of characters: specified Window: specified Position: current
<code>mvadd_wchstr</code>	<code>mvaddwchstr</code>	Number of characters: all Window: default Position: specified
<code>mvadd_wchnstr</code>	<code>mvaddwchnstr</code>	Number of characters: specified Window: default Position: specified
<code>mvwadd_wchstr</code>	<code>mvwaddwchstr</code>	Number of characters: all Window: specified Position: specified
<code>mvwadd_wchnstr</code>	<code>mvwaddwchnstr</code>	Number of characters: specified Window: specified Position: specified

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

Like the functions and macros discussed in the preceding section, the interfaces in the following table also add a wide-character string (but

without video-character attributes) to a window and overwrite existing characters. However, these interfaces also:

- 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:

- 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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>addwstr</code>	<code>addstr</code>	Number of characters: all Window: default Position: current
<code>addnwstr</code>	–	Number of characters: specified Window: default Position: current
<code>waddwstr</code>	<code>waddstr</code>	Number of characters: all Window: specified Position: current
<code>waddnwstr</code>	–	Number of characters: specified Window: specified Position: current
<code>mvaddwstr</code>	<code>mvaddstr</code>	Number of characters: all Window: default Position: specified
<code>mvaddnwstr</code>	–	Number of characters: specified Window: default Position: specified

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>mvwaddwstr</code>	<code>mvwaddstr</code>	Number of characters: all Window: specified Position: specified
<code>mvwaddnwstr</code>	–	Number of characters: specified Window: specified Position: specified

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

The functions and macros discussed in this section insert a wide-character string before a target position in a `curses` window. These interfaces:

- Move further to the right any existing characters 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:

- 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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>ins_wstr</code>	<code>inswstr</code>	Number of characters: all Window: default Position: current
<code>ins_nwstr</code>	<code>insnwstr</code>	Number of characters: specified Window: default Position: current

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>wins_wstr</code>	<code>winswstr</code>	Number of characters: all Window: specified Position: current
<code>wins_nwstr</code>	<code>winsnwstr</code>	Number of characters: specified Window: specified Position: current
<code>mvins_wstr</code>	<code>mvinswstr</code>	Number of characters: all Window: default Position: specified
<code>mvins_nwstr</code>	<code>mvinsnwstr</code>	Number of characters: specified Window: default Position: specified
<code>mvwins_wstr</code>	<code>mvwinswstr</code>	Number of characters: all Window: specified Position: specified
<code>mvwins_nwstr</code>	<code>mvwinsnwstr</code>	Number of characters: specified Window: specified Position: specified

4.1.3 Removing a Wide Character from a curses Window

The function and macros in the following table 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 interfaces 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:

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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>delch</code>	same	Window: default Position: current
<code>wdelch</code>	same	Window: specified Position: current
<code>mvdelch</code>	same	Window: default Position: specified
<code>mvwdelch</code>	same	Window: specified Position: specified

4.1.4 Reading a Wide Character from a curses Window

The function and macros in this section read a wide character, along with 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:

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

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

4.1.5 Reading a Wide-Character String from a curses Windows

There are two sets of routines that allow you to read a wide-character string from a `curses` window. Routines in one set retrieve strings that include wide characters with their video attributes. Routines in the other set strip attributes from the characters in the string.

4.1.5.1 Reading Wide-Character Strings with Attributes

The function and macros in the following table read a wide-character string, along with character attributes, from a `curses` window. The string returned by the recommended interfaces is of the data type `wchar_t`.

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

- 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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>in_wchstr</code>	<code>inwchstr</code>	Number of characters: all Window: default Position: current
<code>in_wchnstr</code>	<code>inwchnstr</code>	Number of characters: specified Window: default Position: current
<code>win_wchstr</code>	<code>winwchstr</code>	Number of characters: all Window: specified Position: current
<code>win_wchnstr</code>	<code>winwchnstr</code>	Number of characters: specified Window: specified Position: current
<code>mvin_wchstr</code>	<code>mvinwchstr</code>	Number of characters: all Window: default Position: specified

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>mvin_wchnstr</code>	<code>mvinwchnstr</code>	Number of characters: specified Window: default Position: specified
<code>mvwin_wchstr</code>	<code>mvwinwchstr</code>	Number of characters: all Window: specified Position: specified
<code>mvwin_wchnstr</code>	<code>mvwinwchnstr</code>	Number of characters: specified Window: specified Position: specified

4.1.5.2 Reading Wide-Character Strings Without Attributes

The function and macros in the following table read a wide-character string from a `curses` window and return a string of data type `wchar_t`. Video attributes are stripped from the characters included in the string.

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

- 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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>inwstr</code>	–	Number of characters: all Window: default Position: current
<code>innwstr</code>	–	Number of characters: specified Window: default Position: current
<code>winwstr</code>	–	Number of characters: all Window: specified Position: current

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>winnwstr</code>	–	Number of characters: specified Window: specified Position: current
<code>mvinwstr</code>	–	Number of characters: all Window: default Position: specified
<code>mvinnwstr</code>	–	Number of characters: specified Window: default Position: specified
<code>mvwinwstr</code>	–	Number of characters: all Window: specified Position: specified
<code>mvinnwstr</code>	–	Number of characters: specified Window: specified Position: specified

4.1.6 Reading a String of Characters from a Terminal

The function and macros in the following table 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:

- 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

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

4.1.7 Reading or Queuing a Wide Character from the Keyboard

Most functions or macros in the following table 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 interfaces 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 whether you get the character for:

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

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

4.1.8 Converting Formatted Text in a curses Window

The following functions 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. Refer to the `scanf(3)` reference page for more information.

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

- 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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>scanw</code>	same	Window: default Position: current Number of arguments: fixed
<code>wscanw</code>	same	Window: specified Position: current Number of arguments: fixed
<code>mvscanw</code>	same	Window: default Position: specified Number of arguments: fixed
<code>mvwscanw</code>	same	Window: specified Position: specified Number of arguments: fixed
<code>vw_scanw</code>	<code>vwscanw</code>	Window: specified Position: current Number of arguments: variable

4.1.9 Printing Formatted Text on a curses Window

The functions in the following table 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. Refer to the `printf(3)` reference page for formatting information.

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

- 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

Recommended Routine	Used in Place of:	Behavior with Respect to:
<code>printw</code>	same	Window: default Position: current Number of arguments: fixed
<code>wprintw</code>	same	Window: specified Position: current Number of arguments: fixed
<code>mvprintw</code>	same	Window: default Position: specified Number of arguments: fixed
<code>mvwprintw</code>	same	Window: specified Position: specified Number of arguments: fixed
<code>vw_printw</code>	<code>wprintw</code>	Window: specified Position: current Number of arguments: variable

4.2 Drawing Ruled Lines in a DECterm Window

Programming guides 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.

Digital UNIX 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 feature is a market requirement for Japanese terminals but has been incorporated into DECterm software for all terminal types.

The following sections describe the escape sequences that draw and erase lines according to pattern and area parameters.

4.2.1 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 following table provides format information:

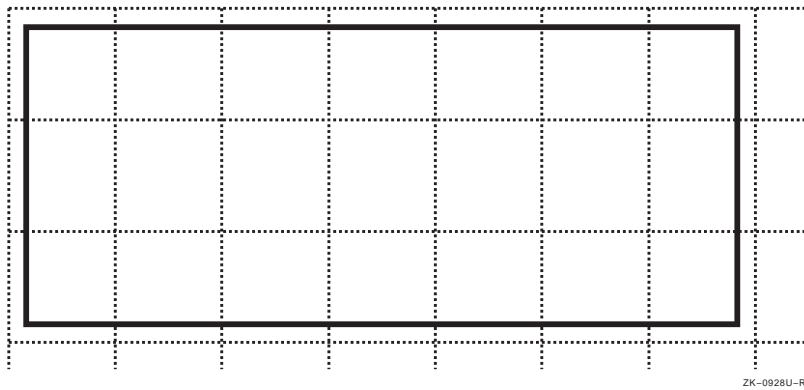
Mnemonic	Description	Sequence
DECDRLBR	Draws ruled lines on the boundaries of a rectangular area	<code>CSI P1;Px;Plx;Py;Ply ,r</code> where: <i>P1</i> indicates the pattern of drawing ruled lines. <i>P1</i> indicates whether lines are drawn on all sides of the rectangular area, on the left and right sides only, on the top and bottom only, and so forth. <i>Px</i> indicates the absolute position of the start point in columns. <i>Plx</i> indicates the width of the area in columns. <i>Py</i> indicates the absolute position of the start point in rows. <i>Ply</i> 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 DECterm software to draw ruled lines as shown in Figure 4-1.

Figure 4-1: Drawing Ruled Lines with the DECDRLBR Sequence

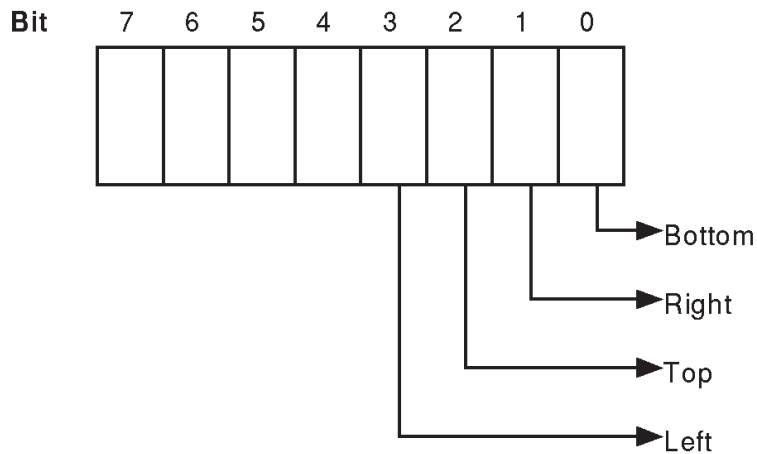


ZK-0928U-R

DECterm software draws ruled lines that are one pixel in width. When the display scrolls, these lines correctly scroll as if text.

Figure 4-2 and the table following the figure describe the bit pattern that the DECDRLBR parameters map to.

Figure 4-2: Bit Pattern for DECDRLBR Parameters



ZK-0931U-R

Figure 4–2: Bit Pattern for DECDRLBR Parameters (cont.)

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:

```
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.

4.2.2 Erasing Ruled Lines in a Pattern

The escape sequence DECERLBRP erases ruled lines on the boundaries of a rectangular area according to a specified pattern. The following table provides format information:

Mnemonic	Description	Sequence
DECERLBRP	Erases ruled lines on the boundaries of a rectangular area	<p>CSI <i>P1</i>;<i>Px</i>;<i>Plx</i>;<i>Py</i> ,<i>s</i> where:</p> <p><i>P1</i> indicates the pattern of drawing ruled lines. <i>P1</i> 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.</p> <p><i>Px</i> indicates the absolute position of the start point in columns.</p> <p><i>Plx</i> indicates the width of the area in columns.</p> <p><i>Py</i> indicates the absolute position of the start point in rows.</p> <p><i>Ply</i> indicates the height of the area in rows.</p>

4.2.3 Erasing All Ruled Lines in an Area

The escape sequence DECERLBRA erases all ruled lines, not just those drawn on the area boundaries, in a rectangular area. The following table provides format information:

Mnemonic	Description	Sequence
DECERLBRA	Erases ruled lines within a rectangular area	<p><i>CSI P1;Px;Plx;Py;Ply ,t</i> where:</p> <p><i>P1</i> determines whether the area encompasses the entire display screen or a specific section of the screen. When <i>P1</i> is the value 1, DECterm software erases all ruled lines on the screen. In this case, the parameters <i>Px</i>, <i>Plx</i>, <i>Py</i>, and <i>Ply</i> are ignored. When <i>P1</i> is the value 2, DECterm software erases all ruled lines within a rectangular area defined by the parameters <i>Px</i>, <i>Plx</i>, <i>Py</i>, and <i>Ply</i>. When <i>P1</i> 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).</p> <p><i>Px</i> indicates the absolute position of the start point in columns.</p> <p><i>Plx</i> indicates the width of the area in columns.</p> <p><i>Py</i> indicates the absolute position of the start point in rows.</p> <p><i>Ply</i> indicates the height of the area in rows.</p>

4.2.4 Interaction of Ruled Lines and Other DECterm Escape Sequences

Table 4–1 describes the effect of using standard DECterm escape sequences when ruled lines are drawn on the screen.

Table 4–1: 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: <pre> +-----+-----+ +-----+-----+ ABCDEF abcdef +-----+-----+ --> +-----+-----+ 123456 123456 +-----+-----+ +-----+-----+ </pre>
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 are scrolled up. For example: <pre> +-----+-----+ +-----+-----+ ABCDEF abcdef 123456 123456 +-----+-----+ --> +-----+-----+ 123456 123456 +-----+-----+ </pre>
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 be scrolled down. For example: <pre> +-----+-----+ ABCDEF abcdef +-----+-----+ --> +-----+-----+ 123456 123456 ABCDEF abcdef +-----+-----+ +-----+-----+ 123456 123456 +-----+-----+ </pre>

Table 4–1: Behavior of Standard Escape Sequences with Ruled Lines (cont.)

Mnemonic	Description	Effect on Ruled Lines
DCH	Delete character	<p>This escape sequence does not delete ruled lines. The following example shows the result of deleting four characters at the third column position:</p> <pre> +-----+-----+ +-----+-----+ ABCDEF abcdef ABabcd ef +-----+-----+ --> +-----+-----+ 123456 123456 123456 123456 +-----+-----+ +-----+-----+</pre>
ICH	Insert character	<p>This escape sequence causes blank spaces to be inserted at the active position but has no effect on ruled lines. The following example shows the result of inserting four characters at the third column position:</p> <pre> +-----+-----+ +-----+-----+ ABCDEF abcdef AB CDEFab cdef +-----+-----+ --> +-----+-----+ 123456 123456 123456 123456 +-----+-----+ +-----+-----+</pre>
IRM	Invoke insert/replace mode	<p>Insert/replace mode has no effect on ruled lines. The following example shows the result of inserting the characters w, x, y, and z at the third column position and replacing the character f with s:</p> <pre> +-----+-----+ +-----+-----+ ABCDEF abcdef ABwxyz CDEFab cdes +-----+-----+ --> +-----+-----+ 123456 123456 123456 123456 +-----+-----+ +-----+-----+</pre>
DECCOLM	Invoke column mode	<p>Ruled lines are erased with accompanying text when column mode is in effect.</p>
RIS, DECSTR	Reset to initial state and soft terminal, invoke reset SETUP mode	<p>The RIS sequence erases all ruled lines displayed on the screen while the DECSTR sequence does not. Note that the Clear Display option on the DECterm Commands menu erases all ruled lines whereas the Reset Terminal option does not.</p>

4.2.5 Determining if the DECterm Device Setting Supports 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.

4.3 DECterm Programming Restrictions

This section discusses DECterm software restrictions with respect to terminal programming features discussed in hardware manuals.

4.3.1 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.

4.3.2 DRCS Characters

DECterm software supports only the Standard Character Set (SCS) component of 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.

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:

- The Toolkit Intrinsic Library available with Release 6 of the X Window System (`libXt`)
- The libraries available with Version 1.2 of OSF/Motif (`libXm`)
- Features provided as DECwindows Extensions to the OSF/Motif Toolkit (`libDXm`)
- The X Library available with Release 6 of the X Window System (`libX11`)

This chapter assumes that you are already familiar with these components. For more complete information on them, refer to the following documents:

- *X Window System*
- *X Window System Toolkit*
- *OSF/Motif Programmer's Guide*
- *DECwindows Motif Guide to Application Programming*

This guide includes a chapter that specifically addresses internationalization concerns of programmers writing Motif applications.

- *DECwindows Extensions to Motif*

In addition to these documents, you can refer to reference pages for individual functions.

This chapter does not discuss internationalization features specific to the Common Desktop Environment. Refer to the *Common Desktop*

Environment: Internationalization Programmer's Guide for information about using these features.

5.1 Using Internationalization Features in the X Toolkit Intrinsic

The X Toolkit Intrinsic 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 X Toolkit Intrinsic Library (`libXt`) in your applications, refer to *X Window System Toolkit* and the reference pages for individual components.

5.1.1 Establishing a Locale with Xt Functions

An internationalized X application must parse resources in a locale-dependent manner. Therefore, an application must establish its locale before initializing the resource database. The `XtSetLanguageProc()` function registers a 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 X Toolkit routines must call `XtSetLanguageProc()`; 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 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 functions then pass this language value to the language procedure registered by the call to `XtSetLanguageProc()`.

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 list, there is a fallback to a valid font set.

5.1.3 Filtering Events During Text Input with Xt Functions

Starting with Release 5 of the X Toolkit Intrinsic, 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 Functions

Starting with Release 5 of the X Toolkit Intrinsic, 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 chapter on internationalization features in the *OSF/Motif Programmer's Guide* discusses how you internationalize Motif applications. The following sections are a supplement to information in that chapter.

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 first introduced in Release 5 of the X Library (`libX`) and the X Toolkit (`libXt`). For example, to set locale and register a language procedure, you use the same set of functions and guidelines in a Motif application as described for an Xt application in Section 5.1.1. As is true for the X Toolkit library, the OSF/Motif and DECwindows Motif Toolkit libraries do not provide a default language procedure. Therefore, if your application fails to register and create a language procedure, the `XmText` and `XmTextField` widgets do not support the internationalization features discussed in subsequent sections; in other words, the widgets revert to behavior expected in releases earlier than the X Toolkit Release 5 and OSF/Motif Release 1.2.

The language for an application can be specified by:

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

Elements higher on the preceding list take precedence over lower elements. 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 `LANG` environment variable, even if the language resource is not defined in the `RESOURCE_MANAGER` property.

5.2.2 Using Compound Strings and the `XmText`, `XmTextField`, and `DXmCSText` 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 the following search pattern to locate the fonts:

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

Items in the preceding list are in precedence order from highest to lowest; the widgets stop the search when an item higher on the list determines a font set.

The internationalization features available through the text widgets have changed on the following two dimensions:

- The segments of a compound string can contain data from multiple character sets. This change is enabled through the font set construct and support for a locale's codeset rather than a single character set per language. (Codesets other than Latin ones usually support multiple character sets.) To take advantage of this change, your application must make sure that:
 - The font list structure defines the appropriate font set as the list element used to display segments of the compound string
 - The compound string includes a tag that will match the correct font set rather than a single font
- For input methods, the `XmText`, `XmTextField`, and `DXmCSText` widgets support the on-the-spot interaction style, as well as off-the-spot, over-the-spot, and root-window styles supported through Release 1.2 of OSF/Motif.

Note that 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 select the off-the-spot input 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, do not use toolkit functions to disable auto-resize 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:

- 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, which describe how to write an internationalized application with the X Library, cover the following topics:

- Managing locales
- Drawing and measuring native-language text
- Handling interclient communication
- Localizing X resource databases
- Handling text input and output

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

5.3.1 Using the X Library to Manage 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. In addition, the X Library includes two additional functions to determine the locale and configure locale modifiers:

XSupportsLocale() and XSetLocaleModifiers(). Table 5–1 provides brief descriptions of these functions. They are more fully described in the XSupportsLocale(3X11) and XSetLocaleModifiers(3X11) reference pages.

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

Function	Description
XSupportsLocale()	Determines if the X Library supports the current locale.
XSetLocaleModifiers()	<p>Specifies a list of X modifiers for the current locale setting.</p> <p>This list is a null-terminated string where list elements use the format <i>@category=value</i>. The only standard category currently defined as a locale modifier is <i>im</i>, which identifies the input method. However, several <i>im</i> entries can appear on a modifier list when a locale supports more than one input method.</p> <p>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 function call. For example, on Digital UNIX systems, the default value for the input method is DEC. The following command explicitly sets the XMODIFIERS variable to this value:</p> <pre>% setenv XMODIFIERS @im=DEC</pre> <p>In this example, the value <i>@im=DEC</i> is the string that would be appended to the modifier list specified on the call to XSetLocaleModifiers().</p>

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. There are five types of objects related 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 shows how an X application sets or determines locale.

Example 5–1: Setting Locale in an X Windows 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;
        }
:
:
        else if (!strcmp(argv[i], "-locale")) [2]
            locale = argv[++i];
        else if (!strcmp(argv[i], "-immodifier")) {
            strcpy(immodifier, "@im=");
            strcat(immodifier, argv[++i]);
        }
    }
}
```

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

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

If this constant had been set to `NULL` rather than `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 structure `XFontSet`, which replaces

the structure `XFontStruct` 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.

Digital's implementation of functions that draw and measure text allow 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, refer to its reference page.

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

Function	Description
<code>XCreateFontSet()</code>	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.
<code>XFreeFontSet()</code>	Frees a specified font set and any associated components, such as the base font name list, the font name list, the <code>XFontStruct</code> list, and <code>XFontSetExtents</code> .
<code>XFontsOfFontSet()</code>	Returns a list of <code>XFontStruct</code> structures and font names for the given font set.
<code>XBaseFontNameListOfFontSet()</code>	Returns the original base font name list supplied by the client when the font set was created.
<code>XLocaleOfFontSet</code>	Returns the name of the locale bound to the specified font set.

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

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

```

:
:
#define DEFAULT_FONT_NAME      "-*-SCREEN-*-*-R-Normal--*-*"  1
:
:

```

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

```
        char                *base_font_name = NULL;
        :
        :
        XFontSet            font_set;
        :
        :
        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 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, you should specify a generic name (using wildcard fields as shown 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
<code>XExtentsOfFontSet()</code>	Returns an <code>XFontSetExtents</code> structure, which contains information about the bounding box of the fonts in the specified font sets.
<code>XmbTextEscapement()</code> , <code>XwcTextEscapement()</code>	Calculate the escapement (in pixels) required to draw a given string by using the specified font set.
<code>XmbTextExtents()</code> , <code>XwcTextExtents()</code>	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 <code>XmbTextEscapement()</code> and <code>XwcTextEscapement()</code> , respectively.
<code>XmbTextPerCharExtents()</code> , <code>XwcTextPerCharExtents()</code>	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

These functions free applications from handling text encoding directly.

Table 5–4: X Library Functions That Draw Text

Function	Description
<code>XmbDrawText()</code> , <code>XwcDrawText()</code>	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, <code>XDrawText()</code> and <code>XDrawText16()</code> .
<code>XmbDrawString()</code> , <code>XwcDrawString()</code>	Using one font set, draw only the specified text with the foreground pixel. Use these functions in place of their single-font counterparts, <code>XDrawString</code> and <code>XDrawString16</code> .
<code>XmbDrawImageString()</code> , <code>XwcDrawImageString()</code>	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, <code>XDrawImageString()</code> and <code>XDrawImageString16()</code> .

Example 5–3 shows how internationalized functions draw text.

Example 5–3: Drawing Text in an X Windows 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;
```

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

```
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();
    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 that is described in 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 bi-directional text and context-sensitive text display.

To draw locale-dependent text, the application needs to know 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 therefore incorporates the following objects to address this problem:

- **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 abstraction.

The following table summarizes the X library functions related to XOM and XOC. For more information on these functions, refer to their reference pages.

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

Function	Description
<code>XOpenOM()</code>	Opens an output method to match the current locale and modifiers specification. The function returns an XOM object to which the current locale and modifiers are bound.
<code>XCloseOM()</code>	Closes the specified output method.
<code>XSetOMValues()</code>	Sets an output method's attributes.

Table 5–5: X Library Functions for Output Method and Context (cont.)

Function	Description
<code>XGetOMValues()</code>	Gets the properties or features of the specified output method.
<code>XDisplayOfOM()</code>	Returns the display associated with the specified output method.
<code>XLocaleOfOM()</code>	Returns the locale associated with the specified output method.
<code>XCreateOC()</code>	Creates an output context within the specified output method.
<code>XOMOfOC()</code>	Returns the output method associated with the specified output context.
<code>XSetOCValues()</code>	Sets the values of the XOC object.
<code>XGetOCValues()</code>	Gets the values of the XOC object.
<code>XDestroyOC()</code>	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, Digital's 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
<code>XmbSetWMProperties()</code>	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.
<code>XmbTextListToTextProperty()</code> , <code>XwcTextListToTextProperty()</code>	Convert text encoded in the current locale to text properties of type <code>STRING</code> or <code>COMPOUND_TEXT</code> .
<code>XmbTextPropertyToTextList()</code> , <code>XwcTextPropertyToTextList()</code>	Convert text properties of type <code>STRING</code> or <code>COMPOUND_TEXT</code> to a list of multibyte-character or wide-character strings.

Table 5–6: X Library Functions for Interclient Communication (cont.)

Function	Description
<code>XwcFreeStringList()</code>	Frees the memory allocated by <code>XwcTextPropertyToTextList()</code> .
<code>XDefaultString()</code>	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 <code>XDefaultString()</code> function queries that default string.

Example 5–4 shows interclient communication in an X application.

Example 5–4: Communicating with Other Clients in an X Windows 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")    ❶  
    }  
    XmbSetWMPProperties(display, window, title, title, NULL, \  
                        0, NULL, NULL, NULL);    ❷  
⋮
```

❶ 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 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

As is also true for font sets, 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. 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
<code>XrmLocaleOfDatabase()</code>	Returns the name of the locale bound to the specified database.
<code>XrmGetFileDatabase()</code>	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.
<code>XrmGetStringDatabase()</code>	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.
<code>XrmPutLineResource()</code>	Adds a single resource entry to the specified database. The entry string is parsed in the locale of the database.

Table 5–7: X Library Functions That Handle Localized Resource Databases (cont.)

Function	Description
<code>XrmPutFileDatabase()</code>	Stores a copy of the specified database in the specified file. The file is written in the locale of the database.
<code>XResourceManagerString()</code>	Converts the <code>RESOURCE_MANAGER</code> property from encoding of type <code>STRING</code> to the encoding of the current locale. This function converts encoding in the same way encoding is converted by the <code>XmbTextPropertyToTextList()</code> function.

5.3.5 Handling Text Input with the X Input Method

When developing internationalized X applications, programmers must be able to request data input in different locales from the same keyboard. The X Library incorporates two abstractions, or objects, that 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 input text data and allow users 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/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 and cannot be changed dynamically.

When the input method is no longer required, the application closes the XIM object with a call to `XCloseIM()`.

Two other functions are available to obtain information about an XIM object:

- `XDisplayOfIM()`

This function returns the display associated with the specified XIM object.

- `XLocaleOfIM()`

This function returns the locale associated with the specified XIM object.

In Digital's implementation of input methods, the input method opened by the `XOpenIM()` function is determined from the following (in order of highest to lowest priority):

- The value for the `im` modifier specified in the call to `XSetLocaleModifiers()`
- The input method specified for the `XMODIFIERS` environment variable
- The default input method, whose name is `DEC`

If `XOpenIM()` cannot obtain the input method from the preceding sources, the fallback is to support only ISO Latin-1 input. The `XOpenIM()` call fails 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 shows how to open and close an input method.

Example 5–5: Opening and Closing an Input Method in an X Windows 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

Behavior of input methods in some areas 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. Currently, only the `XNQueryInputStyle` attribute is defined, and this attribute specifies the user interaction styles supported by an input method.

Example 5–6 shows how to use the `XGetIMValues()` function with the `XNQueryInputStyle` attribute to obtain information for an input method.

Example 5–6: Obtaining the 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;    ❶
        }
    }
    :
    :
    /* set flags for the styles our application can support */
    app_supported_styles = XIMPreeditNone | XIMPreeditNothing |
XIMPreeditCallbacks;    ❷
    app_supported_styles |= XIMStatusNone | XIMStatusNothing;
    XGetIMValues(im, XNQueryInputStyle, &im_styles, NULL);
    n = 1;    ❸
    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) ==
xim_mode) { /* if we can handle it */
                n = 0;
                if (xim_mode & best_style) /* pick user
selected style */
                    break;    ❹
            }
        }
    }
}

```

Example 5–6: Obtaining the User Interaction Styles for an Input Method (cont.)

```
        if(n) {
IMserver.\n");    printf("warning : Unsupport InputStyle. or No
                  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 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.

Supported interaction styles for an input method can vary from one locale to another. Refer to the user guides provided with Digital UNIX language variant subsets to find out what interaction styles are supported for a particular input method.

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 function `XCreateIC()` 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 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 shows how to use the `XCreateIC()` and `XDestroyIC()` functions.

Example 5–7: Creating and Destroying an Input Method Context in an X Windows 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,
:                               W_WIDTH-2, W_HEIGHT-2, 1, bpixel, fpixel,
:                               font_set, &font_height);
:
:
:   if (xim_mode & XIMPreeditCallbacks) {
:       ximapicb[0].client_data = (XPointer)NULL;
:   }

```

Example 5–7: Creating and Destroying an Input Method Context in an X Windows Application (cont.)

```
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 function `XVaCreateNestedList()` to create a nested argument list for preediting and status attributes.

The attributes `XNPreeditAttributes` and `XNStatusAttributes` 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 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 (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 lists and 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.
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. There are a number of callbacks associated with XIC. Example 5–8 shows these callbacks.

Example 5–8: Using Preediting Callbacks in an X Windows Application

```
:\nint    Jxsize[Jxmax_line];\nchar   Jxbuff[Jxmax_line][128];\nint    Jxline_no;\nint    Jxline_height;\nint    sav_cx, sav_cy;\nint    sav_w_width, w_height;\nint    sav_size[Jxmax_line];\nint    sav_line_no;\nchar   preedit_buffer[12];\nvoid\nsave_value()\n{\n    int i;
```

Example 5–8: Using Preediting Callbacks in an X Windows Application (cont.)

```
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
}
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);
        }
    }
}
```

Example 5–8: Using Preediting Callbacks in an X Windows Application (cont.)

```
        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 */
        /* 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. Once 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.

In Digital's implementation of X windows, you can specify the encoding of preediting and status strings in a resource database that is passed to the `XOpenIM()` function. The resources associated with these strings are:

- `decxim.preeditdraw.encoding`
Specifies the encoding of the preediting string.
- `decxim.statusdraw.encoding`
Specifies the encoding of the status string.

Specify `wchar` as the encoding to indicate that wide-character encoding is used.

- 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 simply 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 shows the preceding process.

Example 5–9: Filtering Events for an Input Method in an X Windows 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 */
:
:         }
:
:     }
:
:
```

Example 5–9: Filtering Events for an Input Method in an X Windows Application (cont.)

❶ Filters the event.

Note that the function `XtDispatchEvent()` calls `XFilterEvent()`. Therefore, you could replace the `for` loop as shown 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 shows how to get keyboard input in an X application.

Example 5–10: Obtaining Keyboard Input in an X Windows 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 : ❶
                if(event.xany.window == window)
                    XSetInputFocus(display, client,
                                   RevertToParent, CurrentTime);
                else if(event.xany.window == client) {
                    XSetICFocus(ic);
                }
                break;
            case FocusOut : ❶
                if(event.xany.window == client) {
                    XUnsetICFocus(ic);
```

Example 5–10: Obtaining Keyboard Input in an X Windows Application (cont.)

```
        }
        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;
    }
    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. On Digital systems, 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 closing the XIC and XIM objects, results are unpredictable.

Example 5–11 shows 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)
```


Example 5–11: Handling Failure of the Input Method Server (cont.)

```
XIM im;
XPointer client_data;
XPointer call_data;
{
  Bool *Connected = (Bool *)client_data;
  *Connected =3D False; 5
}
```

- 1 Declare the function that closes the XIM if the input method server (IMS) fails for any reason.
- 2 Declare 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, print an error message and exit. Otherwise, set the `IMS_Connected` variable to `True`, fill the `cb` structure with appropriate client data, and call the `XSetIMValues()` function to register the `XNDestroyCallback` resource for the XIM.
- 4 If the input method server is running, use the `XmbLookupString()` function to process user input; otherwise, use the `XLookupString()` function.
- 5 Specify the prototype for the function that closes the XIM in the event that the input method server fails.

Note that 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 shows 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 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 Intrinsic (Xt) functions differs from programming with X Library functions. Refer to Section 5.1 for discussion of internationalization features of the X Toolkit Intrinsic.

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 `XSetLocaleModifier()` 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.

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 Intrinsic (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 Intrinsic (Xt) Library, use an event handler or a translation/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

Using Internationalized Software

This chapter explains how setup tasks and software features vary among language environments other than English. The chapter is aimed at programmers who are familiar with Digital UNIX in an English-language environment and who need to work with other languages, particularly those that use multibyte characters, to run and test their applications.

6.1 Working in a Multilanguage Environment: Introduction

To enable input and display in any language other than English, you must always set the locale in which your process runs. Depending on the language, you may need to perform additional tasks, for example, to:

- Select keyboard type
- Define search paths for specialized data and executable files that are language specific
- Set terminal code, application code, and other characteristics of the terminal driver to be appropriate for the codeset or codesets where a language's characters are defined
- Load the fonts required for displaying the characters in a particular language
- Enable one or more of the data input and editing methods used for defining and entering characters, words, and phrases
- Apply printer-control characters, filters, and fonts that are appropriate for local-language printers

Note that printing text in languages other than English, particularly Asian languages, may require specialized printer hardware.

This chapter discusses these topics as they apply to particular languages or groups of languages. The chapter also describes command and DECwindows environment features that English-language speakers do not normally use and that allow you to display, enter, print, and mail text in languages other than English. For information about using

internationalization features of applications that run in the Common Desktop Environment (CDE), see the *CDE Companion*.

Language-specific user guides provide additional information about customization and use of software provided for a particular language. These user guides are on the CD-ROM titled “Digital UNIX Online Documentation.” If one or more of the language variant subsets are installed on your system, you can use the following command to read language variant guides using Bookreader. If you did not mount the CD-ROM device to the `/mnt` directory, replace `/mnt` in the following example with the directory to which you mounted the CD-ROM device.

```
% dxbook /mnt/DOCUMENTATION/WORLDWIDE/L10N_guides.decw_bookshelf &
```

PostScript files for the language variant user guides are also available on the CD-ROM. The directories that contain the PostScript versions of these guides have pathnames that adhere to the following format:

```
/mnt/DOCUMENTATION/WORLDWIDE/language_territory/POSTSCRIPT
```

Non-English characters are embedded in the tables and text of these guides. Therefore, to print a guide in PostScript format, you must first:

1. Install the corresponding language variant software to obtain the appropriate printer support files
2. Set up a DEClaser 1152, DEClaser 5100, or PrintServer 17 printer to use the print filters and fonts that are appropriate for the language

Refer to Section 6.12 and `i18n_printing(5)` for information about setting up printers for local languages.

Digital UNIX documentation also provides introductory reference pages on the topics of internationalization (`i18n_intro(5)`) and localization (`l10n_intro(5)`), along with reference pages for all supported languages and codesets.

6.2 Setting Locale and Language

System software that supports different language environments may provide translated message files, application resource files, help files, or some combination of these. If translations are available for message files, you can vary the language of software messages and other text by selecting a locale.

For system software, you set locale by defining the `LANG` environment variable. For example:

```
% setenv LANG en_US.ISO8859-1
```

Refer to the discussion of internationalization in the *System Administration* book and in the *Command and Shell User's Guide* for more detailed information on using locales and defining the associated variables for system and user setup. You can also refer to the `il8n_intro(5)` reference page for a discussion of locale variables such as `LANG`. If these locale variables are not defined, internationalized applications assume the POSIX (C) locale, which supports only English.

Note

Locales sometimes have multiple variants, usually to support different sort orders. These variants have the same name as the base locale but include a file name suffix that begins with the at sign (@). You usually assign locale names with an @ suffix to variables for specific locale categories, such as `LC_COLLATE`, and not to the `LANG` environment variable. The exception to this restriction are @ suffixes associated with codeset variants, such as `@ucs4`.

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.

On a workstation, you also need to select a language to take advantage of text translations and local-language features available with Common Desktop Environment (CDE) and DECwindows Motif applications. For

Asian languages, the correct language selection is particularly important because it enables:

- Support for the appropriate input method in these applications
- Entry of file names and other parameters that use ideographic characters
- Cursor positioning on correct character and word boundaries
- Line wrapping at correct word boundaries

See the *CDE Companion* for information about setting language in the Common Desktop Environment. Use the following steps to select a language in the DECwindows environment:

1. From the Session Manager's Options menu, select Language....
2. In the pop-up Language Options window, click on one of the displayed languages. If you set the `LANG` environment variable before starting your current session, you can click on Default to set the language to be consistent with the value of that variable. Note, however, that `LANG` settings made during your current session do not affect the setting of Default. If you click on an entry other than Default, the selected language overrides the value of the `LANG` environment variable or the system default locale, whichever applies.
3. Click on the Apply button.
4. Click on the OK button to dismiss the Language Options window.

If there is an input method that supports the selected language, you should also start the input method server before starting a DECterm window or other window where you want to work in that language (see Section 6.4). Some languages also require a keyboard setting before you begin entering text in the window (see Section 6.3).

Note

When you set the language, the change applies to all DECterm windows or other DECwindows Motif applications that start after you make the setting. The setting change does not apply to windows that are already started. You can therefore have windows running in different languages at the same time during a DECwindows Motif session.

There is a cut and paste restriction to keep in mind if you simultaneously run windows in different languages. Cutting from one window and pasting to another is supported only when both windows are set to the same language. DECterm windows emulate terminals, so data is transferred as a byte stream that has no embedded language information. Data appears on the target (paste) window according to the language applied to the target window, not according to the language applied to the source (cut) window. For example, data will be meaningless if you cut text from a Chinese window and paste it in a German window. For Chinese or Japanese, codeset converters support cut and paste operations between windows set to the same language but different codesets.

6.3 Selecting Keyboard Type

To enter English text, a standard keyboard provides a sufficient number of keys (combined with shift states) to enter all uppercase and lowercase letters, numerals, and punctuation marks. For many other languages, the default keyboard does not provide enough keys and shift states to enter all characters.

Terminal users must be using a localized keyboard or, if their Digital keyboard includes the Compose key, using Compose-key sequences to enter non-English characters from single-byte codesets. Many Digital terminals also provide software emulation of a number of keyboard layouts for languages that are based on single-byte codesets. The user guide for each terminal explains how you can use its keyboard to enter non-English characters. Entry of multibyte characters in Asian languages requires special terminal hardware.

Workstation users can set keyboard type to be appropriate for languages for which there are standard keyboard types when appropriate support files

are installed on the system. You need to set keyboard type for Western and Eastern European languages, Japanese, Thai, and Hebrew. Keyboard setting is not required for Chinese and Korean languages.

If you are using the Common Desktop Environment, refer to the *CDE Companion* for information about changing keyboard setting. If you are using the DECwindows environment, you can change keyboard type by performing the following steps:

1. From the Session Manager window, select Keyboard... from the Options menu.
2. In the Keyboard Options dialog box, click on a keyboard choice.
3. Click on Apply and then on OK to dismiss the dialog box.

Unlike the language setting, keyboard setting is a global attribute that applies to all windows. Therefore, if you are working in windows created with different language settings, you may need to change the keyboard setting as you move from one window to another.

6.3.1 Determining Keyboard Layout

If you change your keyboard from the one whose characters are printed on the hardware keys, you need to know how characters are mapped to keys and whether any characters must be entered by using a mode-switch key or mode-switch key sequence. For some languages, such as Czech, up to four different characters can be mapped to the same key. In such cases, you use the key defined as the mode switch to toggle among different sets of characters mapped to the same key. Note that mode switching is a character entry mechanism that is different from Compose sequences. A particular keyboard setting may support Compose sequences (which require one key to be defined as a multi-key), mode switching (which requires at least one key to be defined as a mode-switch key), both, or neither of these input mechanisms.

You can access a keyboard layout for your current keyboard setting in one of two ways:

- By clicking on the Edit Keycaps button in the Keyboard Options dialog box (DECwindows environment) or in the Keyboard application window (Common Desktop Environment)

Refer to `dxkeycaps(1X)` for more information on the application invoked by the Edit Keycaps button.

- By using a command similar to the following to create a PostScript file that you can print:

```
% /usr/bin/X11/xkbprint -label symbols -o mykeyboard.ps :0
```

Refer to `xkbprint(1X)` for more information about the `xkbprint` command.

6.4 Determining Input Method

For some languages, such as Japanese, Chinese, and Korean, you use an input method to enter characters, phrases, or both. An input method lets you input 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 four user interaction 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 occlude the last line of text 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 `RootWindow`.

For some of the input styles selected in an application, the preediting and status windows are not redrawn correctly if the application window is occluded by other windows. To correct this problem, click on or refocus on the application window.

Input methods for different locales typically support more than one user interaction style but not all of them. If you are working 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:

```
OnTheSpot , OverTheSpot , OffTheSpot , Root
```

The preceding value means that on-the-spot interaction 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.

There are several ways to supply the `XmNpreeditType` resource value to an application:

- In an application-specific resource file
- On the command line that invokes an application

For example:

```
% dxnotepad -xrm '*preeditType: offthespot,onthespot' &
```

- In the DECwindows environment, through the Session Manager, as follows:
 1. From the Session Manager's Options menu, select Input Style....
 2. In the Input Style Options dialog box, select a preference for interaction style from the Style Preference list.

To position your selection in the list, click on the up-arrow or down-arrow button.
 3. Click on the Apply button and then the OK button to apply the new position to your selection and dismiss the dialog box.

Note that clicking on the Default button of the Input Style Options box restores the system default order to entries on the list.
 4. Repeat the preceding steps until the Style Preference list is in the order you want.
- In the Common Desktop Environment, by using the Input Methods application. See the *CDE Companion* for information on using this application.

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 must be running and made accessible to the application before the application starts. It is therefore important to start an input method server for the DECwindows Motif environment before starting a DECterm window or any other DECwindows Motif application where you want to input characters in a language that requires the server. Following are the input method servers available in the operating system, along with the input styles that each server supports:

- `dxhangulim`, the Korean input server, which supports all four input styles (over the spot, off the spot, root window, and on the spot)
- `dxhanyuim`, the Traditional Chinese input server, which supports the off-the-spot and root-window input styles
- `dxhanziim`, the Simplified Chinese input server, which supports the off-the-spot and root-window input styles
- `dxjim`, the Japanese input server, which supports the on-the-spot, over-the-spot, and root-window input styles

Each of these servers has a corresponding reference page.

The applications that you run may support more, fewer, or none of the input styles supported by a particular input server. The `preedit` option “None” applies when an input server rejects all input styles supported by the application.

In the DECwindows environment, if an input method server is not defined as an application and started through your `.Xdefaults` file at login time (see Section 6.6), you have to start the server from the command line. The following example starts the input server for the Korean language:

```
% /usr/bin/X11/dxhangulim &
```

Note

The Asian-language input methods are not supported for use in `xterm` windows. Therefore, always choose the `DECterm` application to start windows where you want to display and input Asian-language characters.

In the Common Desktop Environment, the appropriate input server is automatically started when you select the language.

6.5 Determining the Input Mode Switch State

The keyboard layout for an Asian language provides keys for only a small number of characters. For Asian languages, you also use an input methodology (incorporating control-key sequences, keypad-key sequences, or options in a DECwindows application) to convert one or more characters that you can input directly from the keyboard to other kinds of characters. Section 6.4 and the language-specific user guides discuss input methods for Asian languages.

If you are using a terminal and your keyboard has a mode-switch LED (light emitting diode), the Keyboard Indicator utility switches the LED on or off, depending on whether you last toggled the special input mode on or off. When using a terminal, invoke the Keyboard Indicator utility with the following command:

```
% /usr/bin/X11/kb_indicator &
```

If you are using a workstation and your language is set to an Asian language, invoke the Keyboard Indicator utility with the `-map` option, as follows:

```
% /usr/bin/X11/kb_indicator -map &
```

The `-map` option starts a DECwindows Motif application that emulates a mode-switch LED. The application window contains one button, which is displayed as on or off, corresponding to the input mode state. You can click on this button to toggle in and out of input mode. The window is insensitive if input mode switching is not supported for your current language setting.

You can have only one Keyboard Indicator application running during your session. To stop the application, enter `Ctrl/C` in the window from which you started the application or enter the following `kill` command with the application's process id:

```
kill -INT process_id
```

If Keyboard Indicator is stopped by any other means, you must enter the following command before restarting the application:

```
% /usr/bin/X11/kb_indicator -clear
```

The preceding command erases the server status for the application so that it can be restarted cleanly.

If your language is set to Hebrew, the Keyboard Manager application (`/usr/bin/X11/decwkm`) provides the same function as the Keyboard Indicator window provides for Asian languages.

6.6 Setting Parameters in the `.Xdefaults` File

In the DECwindows environment, if you want your session to be started with a particular language, input method, or keyboard setting as the default, you can manually edit the `.Xdefaults` file in your home directory to add appropriate entries for language, input method (if applicable), and keyboard. Alternatively, you can select the language and keyboard options you want from DECwindows Motif Session Manager menus, quit the session, and click on the affirmative answer when asked whether you want to save current settings. Saving current settings adds lines to or modifies existing entries in your `.Xdefaults` file. When you log back in to start a new session, the changed defaults take effect.

Example 6–1 shows an `.Xdefaults` file, modified by the choice to save current settings when quitting the session. The language and keyboard settings are Japanese (DECKanji) and LK401aj, respectively. The string `dxjim` has been added to several lines to define the Japanese input method server as a DECwindows Motif application and automatically start the server process.

Example 6–1: Sample `.Xdefaults` File

```
DXsession.x:      3
DXsession.y:      40
DXsession.AutoStart:  dxjim
DXsession.applications:  Bookreader,CDA Viewer,Calculator,\
Calendar,Cardfiler,Clock,DECTerm,Differences,\
Mail,Notepad,Paint,Print Screen,XTerm,dxjim
DXsession.dxjim.command:  /usr/bin/X11/dxjim
DXsession.num_AutoStart:  1
DXsession.num_applications:  15
DXsession.AppMenu:  Bookreader,CDA Viewer,Calculator,\
Calendar,Cardfiler,Clock,DECTerm,Differences,Mail,Notepad,\
Paint,XTerm,dxjim
DXsession.num_AppMenu:  13
*xnLanguage:  ja_JP
```

Example 6–1: Sample .Xdefaults File (cont.)

```
*keyboard_dialect:      japanese lk401aj
```

Refer to Section 6.18 for information about using specific DECwindows Motif applications with Asian languages. Section 6.18 also discusses X Server customization that is important when ideographic fonts are used in local and remote displays.

For information about customizing session defaults in the Common Desktop Environment, see the *CDE Companion*.

6.7 Defining the Search Path for Specialized Components

European languages are supported by data and executable files installed at system default locations. Asian-language support for some commands and programming libraries requires files that are subordinate to the directory `/usr/i18n`. These files supplement or replace files in system default locations. When you install one or more of the Asian language subsets, the installation procedure makes the following adjustments to variable settings on a systemwide basis:

- `I18NPATH`

The `I18NPATH` variable defines the location of files that provide Asian-language support and that are not in system default locations. This variable is set to:

```
/usr/i18n
```

Your system administrator can choose to install files for Asian-language support at a location different from `/usr/i18n`; however, there must be a link to the other location in the `/usr/i18n` directory.

- `PATH`

The `PATH` variable points to the location of commands and is set to:

```
$I18NPATH/usr/bin:$PATH
```

The file `/etc/i18n_profile` includes the preceding variable assignments on a systemwide basis for Bourne and Korn shell users. For C shell users, the installation process includes the file `/etc/i18n_login` in the file `/etc/csh.login` to set search paths correctly for Hebrew and Asian

languages. Unless specifically noted in descriptions of particular commands or utilities, individual users do not need to change process-specific search paths to find localized binaries and utilities.

6.8 Using Terminal Interface Features for Asian Languages

The Digital UNIX Asian terminal driver (`atty`) and Thai terminal driver (`ttty`) support input and output of English and other language characters over asynchronous terminal lines. When one or both of these drivers are installed, you can set terminal line characteristics to be appropriate for the language you are using. The driver's local-language capabilities are supported in the following terminal configurations:

- Terminal connected directly to the host machine via a serial line
- Terminal connected through LAT to the host system
- Terminal connected through TCP/IP to the host system

Refer to the `atty(7)` and `ttty(7)` reference pages for more information about these terminal drivers.

Asian-language software subsets provide an enhanced `stty` command that can enable support for multibyte codesets and special character manipulation capabilities, such as the following:

- Automatic codeset conversion between terminal and application
- Line editing of multibyte characters
- Japanese input method (Kana-Kanji conversion)
- User-defined character (UDC) databases and on-demand loading (ODL) of associated fonts
- Chinese phrase input method

This section provides general information about using the `stty` command to enable features added to the terminal subsystem for Asian languages.

The `stty` utility sets or reports on terminal input/output characteristics of the device that is the utility's standard input. Table 6-1 shows the `stty` options that set line discipline for Asian languages.

Table 6–1: The stty Command Options for Controlling Terminal Line Discipline

stty Option	Description
<code>adec</code>	Sets the terminal line discipline to handle multibyte data and the processing environment appropriate for simplified Chinese (Hanzi), traditional Chinese (Hanyu), and Korean codesets. This option is supported for both the STREAMS and BSD terminal drivers.
<code>jdec</code>	Sets the terminal line discipline to handle multibyte data and the processing environment appropriate for Japanese codesets. This option sets terminal code to <code>dec</code> and application code to <code>eucJP</code> . The <code>jdec</code> option is supported for both the STREAMS and BSD terminal drivers.
<code>tdec</code>	Sets the terminal line discipline to handle Thai characters and the processing environment appropriate for the Thai codeset. This option is supported for only the BSD terminal driver.
<code>dec</code>	Sets the terminal line discipline back to the default, or standard, <code>tty</code> line discipline and clears characteristics that preceding <code>stty</code> commands may have set for application and terminal code. This option is supported for both the STREAMS and BSD terminal drivers.

The `stty` command requires an appropriate locale setting to be in effect before changing terminal line discipline to support that locale. For example, to set your terminal line discipline to handle Korean, enter:

```
% setenv LANG ko_KR.deckorean
% stty adec
```

To set your terminal line discipline back to the `tty` default, enter:

```
% stty dec
```

Note

When your terminal line discipline is not set to the `tty` default and you want to switch to another nondefault option (to switch from `jdec` to `adec`, for example), first enter the `stty dec` command to clear any application or terminal characteristics that may not be appropriate for the new setting. The following example shows how to switch a terminal line discipline from its current setting of `adec` to `jdec`:

```
% stty dec
% stty jdec
```

The `stty` command entered with the `-all` option displays all settings for the current terminal line discipline:

```
% stty adec
% stty all
atty disc;speed 9600 baud; 24 rows; 80 columns
erase = ^?; werase = ^W; kill = ^U; intr = ^C; quit = ^\; susp = ^Z
dsusp = ^Y; eof = ^D; eol <undef>; eol2 <undef>; stop = ^S; start = ^Q
lnext = ^V; discard = ^O; reprint = ^R; status <undef>; time = 0
min = 1
-parenb -parodd cs8 -cstopb hupcl cread -clocal
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl -iuclc
ixon -ixany -ixoff imaxbel
isig icanon -xcase echo echoe echok -echonl -noflsh -mdmbuf -nohang
-tostop echoctl -echoprt echoke -altwerase iexten -nokerninfo
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tabs -onoet
-odl lru size=256
-sim key= class=
tcode=dec acode=dekanji
```

6.8.1 Converting Between Application and Terminal Codesets

Many terminals support only one codeset, which is a problem when you work on one terminal and need to run applications in locales (particularly Asian locales) that are based on a variety of codesets. Therefore, the `atty` driver provides a mechanism for converting between the codeset that an application uses and the codeset that a terminal supports. You control codeset conversion by using options on the `stty` command line.

Note that the `adec`, `jdec`, and `dec` options of the `stty` command set terminal code and application code appropriately for Digital terminals and workstations. You need to explicitly use the `tcode` option, for example, if

you are logging on from a Japanese terminal that does not support the standard codeset for Digital terminals.

Table 6-2 specifies `stty` options that explicitly set terminal and application code.

Table 6-2: The `stty` Options to Explicitly Set Application and Terminal Code

stty Option	Description
<code>acode codeset</code>	Sets application code to <code>codeset</code> .
<code>tcode codeset</code>	Sets terminal code to <code>codeset</code> . For most all Digital terminals and for workstations, <code>tcode</code> should be set to <code>dec</code> .
<code>code codeset</code>	Sets both terminal code and application code to <code>codeset</code> .

The following command lets you run an application that uses DEC Kanji (the default codeset for Japanese) on a terminal that supports only Shifted JIS (a codeset prevalent in the Japanese personal computer market):

```
% stty acode deckanji tcode sjis
```

The user guides for the Asian-language subsets provide additional details about supported application codesets and terminal codesets.

6.8.2 Command Line Editing That Supports Multibyte Characters

This section discusses how you enable and use command-line editing when Asian-language support is installed on your system.

When the terminal line discipline and terminal codeset characteristics are set appropriately for multibyte codesets, the `atty` driver handles command-line editing appropriately for languages supported by those codesets. For example, when you enter the control sequence to delete a character (assuming you have defined the control sequence), the entire character is deleted, regardless of how many bytes it occupies. The character being erased can be either a single-byte English character or a multibyte Asian character when both occur on the same command line.

Word deletion is also supported, even when words combine single-byte and multibyte characters. The `atty` driver accepts single-byte space characters, two-byte space characters (if applicable to the terminal code setting), or tab characters as word delimiters.

The `erase` and `werase` options of the `stty` command line let you define the control sequence for character and word deletion. For example:

```
% stty erase ^H
% stty werase ^J
```

The preceding example specifies that `Ctrl/H` deletes the character that precedes the cursor and `Ctrl/J` deletes the word preceding the cursor.

History mode is a mode of command-line editing that allows you to recall and optionally modify a command entered previously. The history mode implementation discussed here is one that is customized for Asian-language input and supported only for the BSD terminal driver. Table 6-3 specifies the `stty` options that enable or disable history mode editing.

Table 6-3: The `stty` Options to Enable/Disable History Mode

<code>stty</code> Option	Description
<code>history key</code>	Sets the toggle key for the history mechanism and enables it.
<code>-history</code>	Disables the history mechanism.

The `atty` driver can maintain a history of up to 32 commands, each with a maximum length of 127 characters. Table 6-4 describes the commands you can use to edit command lines after entering the history `key`.

Table 6-4: Command Line Editing in History Mode

Command/Key	Description
<code>Ctrl/A</code>	Move to the beginning of the line
<code>Ctrl/D</code>	Delete the character under the cursor
<code>Ctrl/E</code>	Move to the end of the line
Up-arrow	Recall the previous command line in the history list
Down-arrow	Recall the next command in the history list

Table 6–4: Command Line Editing in History Mode (cont.)

Command/Key	Description
Left-arrow	Move the cursor left by one character
Right-arrow	Move the cursor right by one character
<i>erase_sequence</i>	Delete the character preceding the cursor
<i>werase_sequence</i>	Delete the word preceding the cursor

In the preceding table *erase_sequence* and *werase_sequence* indicate the control sequences defined by the `stty` options `erase` and `werase`, respectively.

When editing a command line in history mode, you insert characters as follows:

1. Press the arrow keys to move the cursor to the position immediately to the right of the point where you want to insert characters.
2. Enter the characters you want to insert.

If you enter the control characters that represent “kill,” “interrupt,” or “suspend,” the `tty` driver breaks out of history mode and cancels the command line being edited.

6.8.3 Kana-Kanji Conversion: Customization of Japanese Input Options

In the Japanese language, a particular language element, such as a vowel, can be represented by more than one character. These characters can have both phonetic and ideographic variants; furthermore, the phonetic character variants can print in either two-column or single-column width. The different classes of characters, listed in the following table, require different input schemes:

Character Class	Description
Kanji	Ideographic
Hiragana	Phonetic
Katakana	Phonetic Katakana characters exist in full width (two-column) and half width (single-column) formats. The single-column format of Katakana is referred to as Hankaku.

During a single session, a Japanese user can work with Kanji, Hiragana, and Katakana characters in various combinations. The user therefore must be able to customize terminal input mode to suit the character being entered. When the input device is a JIS terminal rather than a workstation, the user must adjust line discipline and terminal code settings in the software to match hardware capabilities (for example, whether the terminal uses 7-bit or 8-bit encoding).

The `tty` driver supports a mechanism known as **Kana-Kanji conversion**. This term refers to the conversion between phonetic and ideographic character encoding and the support for keyboard entry sequences that make Japanese character selection more efficient for the user. You use the `stty` command to enable or disable the Kana-Kanji conversion method and other aspects of Japanese input support. The `stty` options that support Japanese input are described in Table 6-5 and, unless noted otherwise, are used in conjunction with the `jdec` option. For example, the following command sets the terminal line discipline to support Japanese character encoding and also enables Kana-Kanji conversion:

```
% stty jdec ikk
```

Table 6–5: The stty Options to Enable and Customize Japanese Input

stty Option	Description
<code>clause mode</code>	<p>Sets the character attribute for marking a clause that results from Kana-Kanji conversion.</p> <p>The <i>mode</i> argument can be bold, underline, reverse, or none.</p>
<code>esc.alw</code>	<p>Changes the terminal state to “shift out” whenever a newline character is output.</p> <p>This option applies only when the <code>tcode</code> (terminal code) <code>stty</code> option is set to <code>jis7</code> or <code>jis8</code>.</p>
<code>-esc.alw</code>	<p>Does not change the current terminal state when a newline character is output.</p> <p>This option applies only when the <code>tcode</code> (terminal code) option is set to <code>jis7</code> or <code>jis8</code>.</p>
<code>henkan mode</code>	<p>Sets the character attribute for marking a Henkan, or conversion, region that results from Kana-Kanji conversion.</p> <p>The <i>mode</i> argument can be bold, underline, reverse, or none.</p>

Table 6–5: The stty Options to Enable and Customize Japanese Input (cont.)

stty Option	Description
<code>ikk</code>	<p>Enables the Japanese input method and spawns the Kana-Kanji conversion daemon, <code>kkcd</code>, if it does not already exist. Depending on the terminal driver (BSD or STREAMS), use either the <code>jx</code> or <code>jinkey</code> option before using the <code>ikk</code> option to enable the input method.</p> <p>By default, key map information is taken from (in highest to lowest priority order):</p> <ul style="list-style-type: none">• The file specified for the <code>kkseq</code> option of the <code>stty</code> command• The file defined for the environment variable <code>JSYKKSEQ</code>• The file <code>\$HOME/.jsykkseq</code> <p>System default key map files for the Japanese input method reside in the directory <code>/usr/i18n/skel/ja_JP</code>.</p> <p>Dictionaries used with the Japanese input method are taken from (in highest to lowest priority order):</p> <ul style="list-style-type: none">• The files defined for the environment variables <code>JSYTANGO</code>, <code>JSYKOJIN</code>, and <code>JSYLEARN</code>• The dictionary files <code>/usr/i18n/jsy/jsytango.dic</code>, <code>\$HOME/jsykojin.dic</code>, and <code>\$HOME/jsylearn.dic</code>
<code>-ikk</code>	<p>Disables the Japanese input method and kills the <code>kkcd</code> daemon.</p>
<code>jinkey sequence</code>	<p>Defines the escape sequence to activate the extended Japanese input method used with the STREAMS terminal driver. The parameter for this option can be more than one character.</p>

Table 6–5: The stty Options to Enable and Customize Japanese Input (cont.)

stty Option	Description
<code>imode mode</code>	Sets the mode for handling 8-bit code or Hankaku (single-column) Kana code when the terminal line discipline is set to <code>dec</code> . The <code>mode</code> argument can be one of the following keywords: <ul style="list-style-type: none">• <code>kanji</code>, where the 8-bit code is treated as encoding for Kanji• <code>hiragana</code>, where the 8-bit code is converted to 2-column Hiragana format• <code>katakana</code>, where the 8-bit code is converted to 2-column Katakana format• <code>hankaku</code>, where the 8-bit code is handled in Hankaku (1-column) Katakana format
<code>jx character</code>	Sets the toggle character for entering the extended, or <code>cbreak</code> , Kana-Kanji conversion mode used with the BSD terminal driver. Users need to enter <code>cbreak</code> mode when working in utilities, such as <code>dbx</code> , that do not support the full range of Japanese input options.
<code>-jx</code>	Undefines the toggle character for entering the extended Kana-Kanji conversion mode.
<code>kin esc_sequence</code>	Sets the JIS Kanji “shift in” escape sequence for the JIS terminal.
<code>kkmap</code>	Displays the current key map for Kana-Kanji conversion. The display is a traversal tree with a maximum length of 15 characters for each key sequence.
<code>kkseq file</code>	Sets the Kana-Kanji conversion key map file for the terminal (see also the table entry for the <code>ikk</code> option).
<code>knj.bsl</code>	Uses only one backspace to erase one Kanji character.
<code>-knj.bsl</code>	Uses two backspaces to erase one Kanji character.
<code>knj.sp</code>	Uses one 2-byte (zenkaku) space to blank out one Kanji character.

Table 6–5: The stty Options to Enable and Customize Japanese Input (cont.)

stty Option	Description
<code>-knj.sp</code>	Uses two ASCII spaces to blank out one Kanji character.
<code>kout esc_sequence</code>	Sets the JIS Kanji “shift out” escape sequence for the JIS terminal.

6.9 Setting Up and Using User-Defined Character Databases

The national character sets for Japan, Taiwan, and China do not include some of the characters that can appear in Asian place 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. You create characters with the `cedit` application, discussed in Section 6.9.1. You use the `cgen` utility, discussed in Section 6.9.2, to create font, collation, and other support files for user-defined characters. X applications can also obtain fonts for user-defined characters directly from a UDC database by using font renderers. Refer to Section 6.18.2 for information about font renderers.

Note

The system default `sort` command does not access the collation files created for user-defined characters. Refer to Section 6.14 for information on sorting strings that may contain these characters.

The rest of this section discusses some setup that is necessary before terminals or workstation monitors can display user-defined characters.

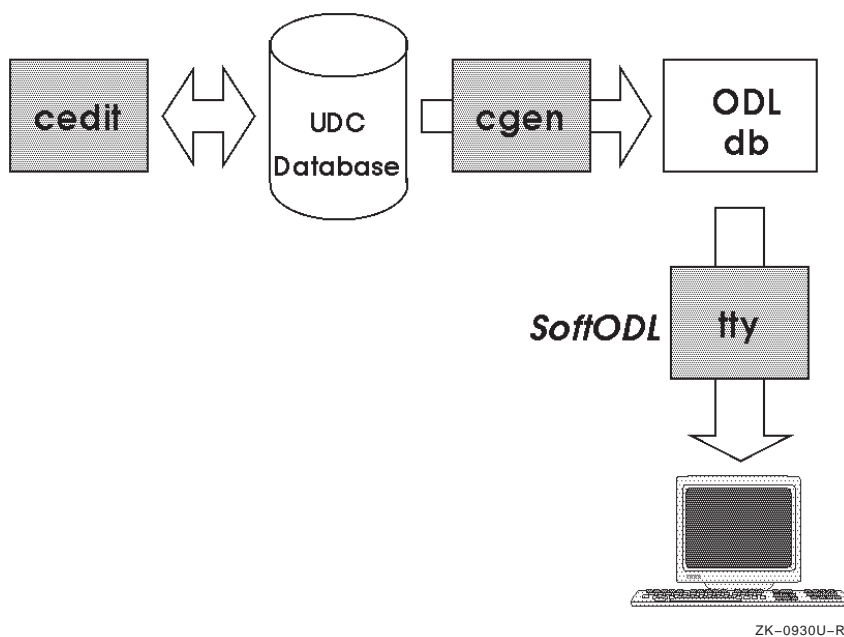
The `atty` driver includes a mechanism to allow on-demand loading of files associated with user-defined characters. You enable this mechanism and can change some of its default parameter values with the `stty` command. Table 6–6 describes the `stty` options that you use with on-demand loading.

Table 6–6: The stty Options for On-Demand Loading of UDC Support Files

stty Option	Description
<code>odl</code>	Enables the software on-demand loading (SoftODL) service.
<code>-odl</code>	Disables the software on-demand loading (SoftODL) service.
<code>odlsize size</code>	Sets the maximum size of the ODL buffer. This size should be the same as a terminal's font-cache size. By default, <i>size</i> is 256 characters.
<code>odltype type</code>	Sets the ODL buffer replacement strategy. Valid values for <i>type</i> are: <code>fifo</code> (first-in-first-out) and <code>lru</code> (least recently used)
<code>odldb path</code>	Sets the path to the database and other files that support user-defined characters. 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 file <code>/var/i18n/conf/cp_dirs</code> , which is described in a subsequent section of this chapter. The <code>cp_dirs</code> file specifies, for example, that the systemwide defaults are <code>/var/i18n/udc</code> and <code>/var/i18n/odl</code> , and that the process defaults are <code>\$HOME/.udc</code> and <code>\$HOME/.odl</code> . Use the <code>odldb</code> option when you want to change the default <code>odl</code> file.
<code>odlreset</code>	Resets the ODL service and clears the internal ODL buffers.
<code>odlall</code>	Displays the current settings for the ODL service.

Figure 6–1 shows the relationship among components mentioned in the preceding list and the SoftODL service.

Figure 6–1: Components That Support User-Defined Characters



6.9.1 Creating User-Defined Characters

The user-defined character editor (`cedit`) is a curses application for managing attributes of user-defined characters. The character attributes that you usually manipulate with the `cedit` application include:

- Styles and sizes (16x18, 24x24, 32x32, and 40x40) for bitmap fonts
- Codeset values
- Collating values
- Input key sequences

Each user-defined character has a character attribute record, which is stored in a character attribute, or 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 as well. The following command invokes the user-defined character editor:

```
% cedit
```

The preceding command, which includes no options, uses the default database. If you are superuser, the default database is `/var/i18n/udc`. If you are an unprivileged user, the default database is `$HOME/.udc`. There are a number of problems you can encounter when using user-defined characters that are maintained in private databases; therefore, Digital recommends that user-defined characters be maintained only in a systemwide database by a privileged user. The `cedit` command has a number of options and an argument, which are described in Table 6-7.

Table 6-7: The `cedit` Command Options

cedit Options and Arguments	Description
<code>-c old_db</code>	Converts a Japanese ULTRIX <code>fedit</code> font file or an Asian ULTRIX character attribute database file to the format used by <code>cedit</code> .
<code>cur_db</code>	Specifies the path of a character attribute database (to override the default path).
<code>-h</code>	Displays <code>cedit</code> syntax.
<code>-r ref_db</code>	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 <code>cedit</code> utility. The Reference Database item on the <code>cedit</code> File menu is an alternative to specifying the <code>-r</code> option on the <code>cedit</code> command line.

The following command displays the `cedit` syntax format:

```
% cedit -h
Usage : cedit [-h] [-c <old_db>] [-r <ref_db>] [<cur_db>]
```

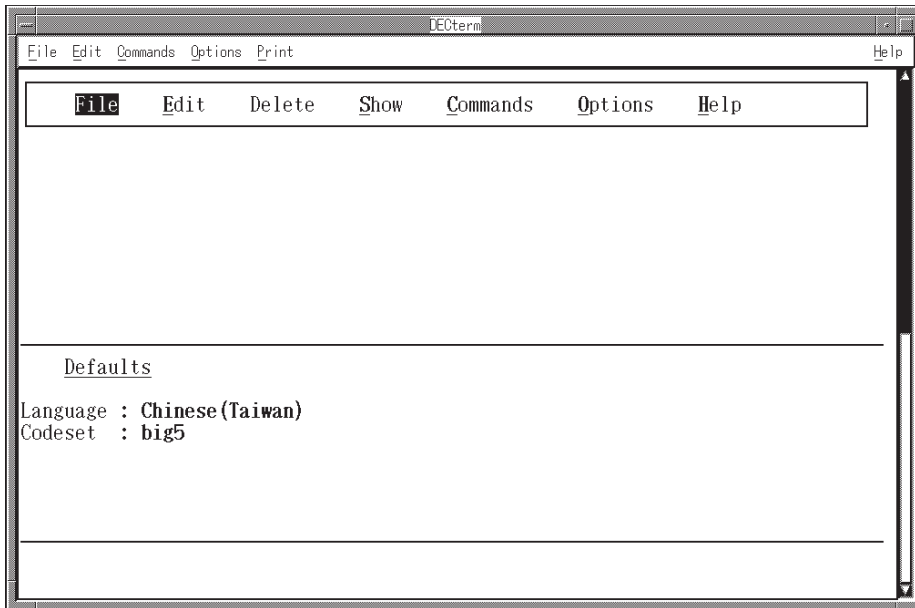
The `cedit` command returns an error message if your locale setting is one that is not supported for creation of user-defined characters. Locales supported for user-defined characters include those for the Chinese and Japanese languages. After you invoke `cedit`, you can use the Options menu on the `cedit` user interface screen to change the language of user interface messages and help text back to English.

The following sections discuss the screens, menu items, editing modes, and function keys of the `cedit` utility.

6.9.1.1 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 shown in Figure 6–2.

Figure 6–2: The cedit User Interface Screen



ZK-0924U-R

The user interface screen is divided into three areas:

- Menu area

This area contains a bar of menu names. When you select 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 navigate the menu interface, you can use the four arrow keys to select a menu and then press either Return or the space bar to see items on that menu. You can accomplish the same goal more directly by pressing the key for the letter that is underlined in the title of the menu.

Menu items are displayed in one of the following states:

- Active

An active item is one that you can select. 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 select inactive items. Inactive items do not contain underlined and highlighted letters.

- Selected

If you press the down arrow key rather than the key for a highlighted letter, you can select items without starting the functions they represent. The currently selected item is shown in reverse video.

- Activated

You activate an item when you press the key for a highlighted letter or when you press Return or the space bar after selecting 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.

In the text that follows, when you are told to choose an item, you should activate it.

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:

- Save changes made to the character you are currently working on
- Cancel changes made to the current character
- Change the reference character attribute database

- Exit from or quit the `cedit` program
- Edit

Use the Edit menu to select a character and create or change its font glyph, codeset value, collating value, input key sequence, class, or name. Section 6.9.1.2 discusses editing a character's font glyph.
- 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/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)

There is some variation among Asian codesets in terms of 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:

- 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 blindly (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 user-defined characters. 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 terminal or, for workstation users, enabled by your terminal setting. In other words, you can first select a menu item with the arrow keys and then press the Help key for a short description of the selected item.

6.9.1.2 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. Select a character by choosing the Character item from the Edit menu.

The `cedit` program then 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 locales for Asian languages. 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, it is a good idea to define user-defined characters to have values that can be mapped to other codesets supported for the language. For more information on codes for user-defined characters in specific Asian languages, refer to the language-specific user guides available with the Asian-language subsets of Digital UNIX.

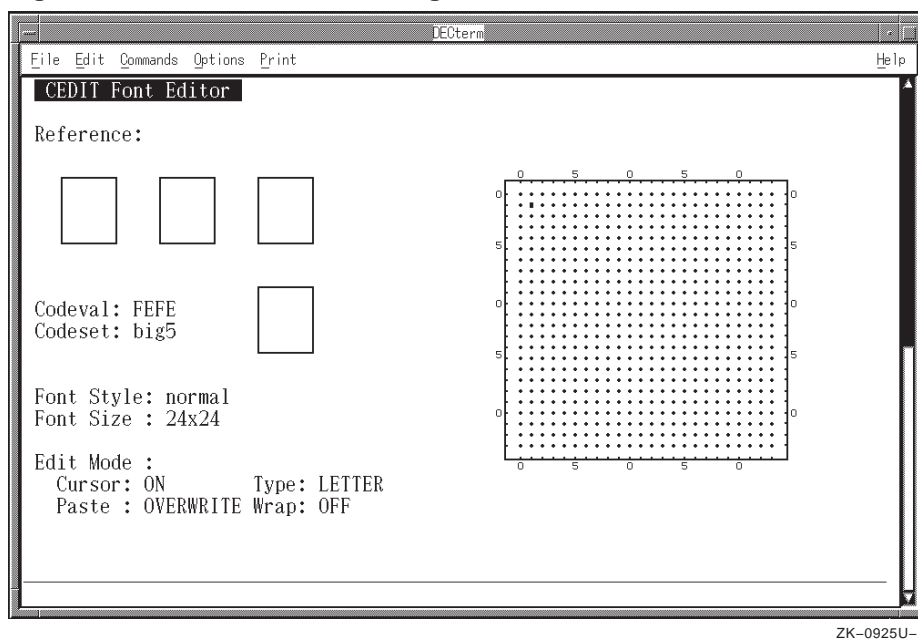
The `cedit` utility first searches your current UDC database for the code that you enter. If a character with that code is not found in the UDC database, the utility searches the current reference character database.

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 DECwindows 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 DECwindows font.

The `cedit` program then displays the full-screen font editor interface as shown in Figure 6-3.

Figure 6–3: The cedit Font Editing Screen



ZK-0925U-R

The *cedit* font-editing screen has several windows:

- The large window on the right-hand 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 shows 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.

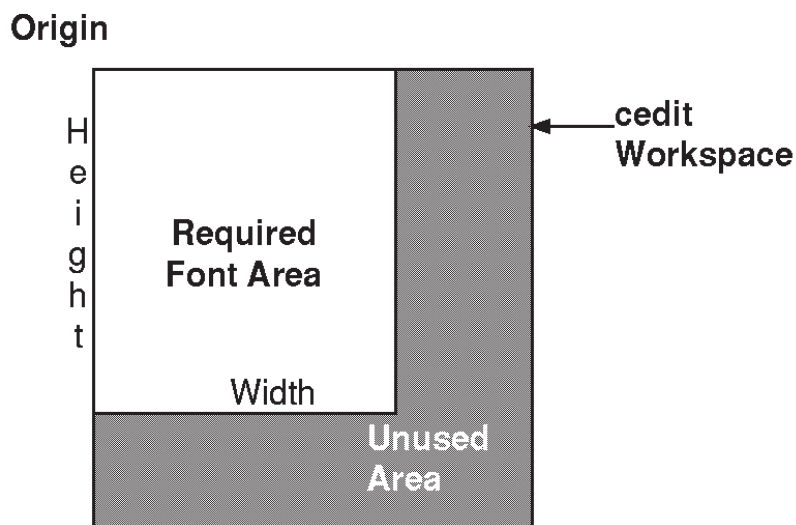
Font glyph displays in the reference and display windows are enabled only on certain terminals, specifically, on local-language terminals that support the Dynamic Replacement Character Set (DRCS) function.

On DECterm 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 selected before the editor interface screen appeared.

You can also create a font for use with DECwindows software and whose dimensions are smaller than those selected. 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 (see Figure 6-4). The UDC font converter that supports DECwindows 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 6–4: Interpretation of Font Editing Screen for Sizing DECwindows Font



ZK-0932U-R

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. Note that your online diagram may vary from the one shown here due to differences in keypad design on some systems. There are four kinds of editing modes for the `cedit` editing screen:

- **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 KP5.

- 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 it or 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.

Letter mode means that 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

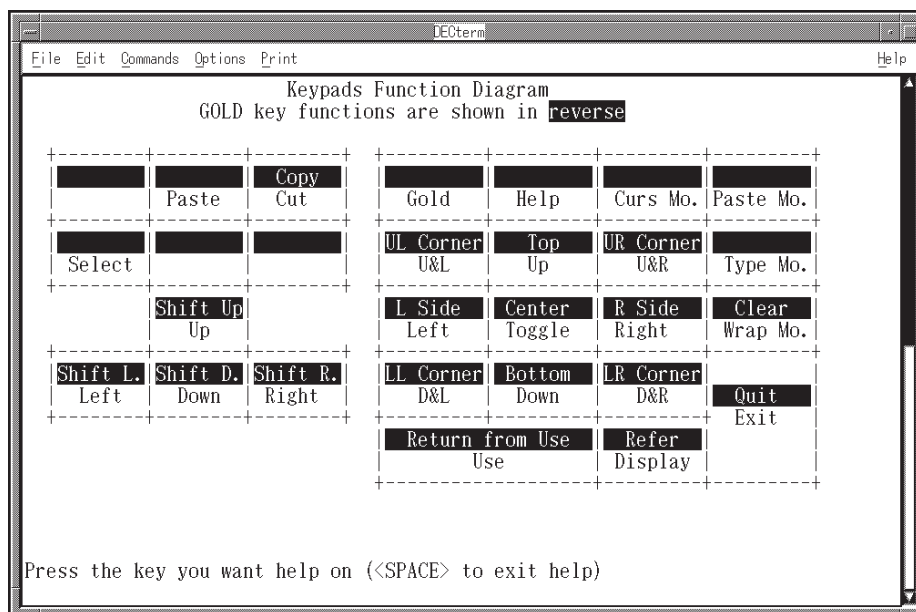
This function retrieves a font glyph from a UDC database or from one of the reference windows.

- Refer

This function saves a font glyph copied from a UDC database into one of the reference windows.

Figure 6-5 shows 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 6–5: Keymap for cedit Functions



ZK-0926U-R

Table 6–8: Keys for Miscellaneous Font Editing Functions

Key	Description
Help or PF2	Shows you 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. 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.

Table 6–8: Keys for Miscellaneous Font Editing Functions (cont.)

Key	Description
Enter	Saves changes and exits from the font editor.
GOLD Enter	Quits the font editor without saving changes.

Table 6–9: Keys for edit Mode Switching

Key	Description
PF3	Toggles Cursor mode.
PF4	Toggles Paste mode.
KP-	Toggles Type mode.
KP.	Toggles Wrap mode.

Table 6–10: 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.
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 6–11: Keys for Moving Cursor to Window Areas

Key	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.

Table 6–12: 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.
F or f	Depending on cursor mode, fills an area.
T or t	Inverts the state of all pixels.

Table 6–13: Keys for Editing Font Glyphs

Key	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.

There is often more than one way to perform the same editing operation. The following summary discusses one method to accomplish various operations:

- Drawing the glyph

Use the keys KP1 to KP9 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

Drawing keys 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 DECterm window, you can press the KP. key to display the glyph in actual size. This operation is not supported through DECterm windows.

- Creating multiple prototypes of a glyph

You can create several versions of a glyph, storing earlier 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.

Note that 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 bound to the keypad copies a glyph from another codepoint in the current or reference database. The Use function 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 in 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.

6.9.2 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 user-defined characters
- Font files that X and Motif applications use

The following command creates some of these files for the UDC database `~wang/.udc`:

```
% cgen -odl -pre -col -iks ~wang/.udc
```

If you enter the `cgen` command without specifying options, it displays statistical information about the specified database. If you enter the command without specifying a UDC database, the private user database is used for a nonprivileged user and the system database for the superuser. 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 6–14 describes `cgen` command options.

Table 6–14: The `cgen` Command Options

Option	Description
<code>-bdf</code>	Creates <code>.bdf</code> files needed for X and DECwindows Motif applications.
<code>-col</code>	Creates collating value tables. You must use the <code>asort</code> command, rather than the <code>sort</code> command, if you want to apply these tables during sort operations.
<code>-dpi 75 100</code>	Sets resolution to either 75 or 100 when creating <code>.bdf</code> and <code>.pcf</code> files with the <code>-bdf</code> and <code>-pcf</code> options.
<code>-fprop property</code>	Sets the font property when creating <code>.bdf</code> and <code>.pcf</code> files with the <code>-bdf</code> and <code>-pcf</code> options.
<code>-iks</code>	Creates the input key sequence file.
<code>-merge font_pattern</code>	Invokes the <code>fontconverter</code> command to merge the UDC fonts with an existing <code>pcf</code> font file that matches the specified <code>font_pattern</code> (for example, <code>'*-140-*jisx0208*' </code>). If you specify the <code>-merge</code> option, you must also specify the <code>-pcf</code> and <code>-size</code> options. The output <code>.pcf</code> file is in the form <code>registry_width_height.pcf</code> , where <code>registry</code> is the font registry field of the specified font file.

Table 6–14: The cgen Command Options (cont.)

Option	Description
<code>-osiz widthxheight</code>	<p>Specifies the font size for <code>bdf</code> output format.</p> <p>The font size in <code>bdf</code> format may be different from the size of the font defined in the UDC database. The font sizes that the <code>cedit</code> command supports are limited; the <code>-osiz</code> option lets you override these size restrictions both in the <code>.bdf</code> file and the <code>.pcf</code> file generated from the <code>.bdf</code> file.</p> <p>If the size parameters specified for the <code>-osiz</code> option are smaller than the size parameters specified for the <code>-size</code> option, only the upper left portion of the UDC font glyph is used. If the size parameters specified for the <code>-osiz</code> option are larger than the size parameters specified for the <code>-size</code> option, the lower-right portion of the resulting font glyph is filled with OFF pixels.</p>
<code>-pcf</code>	<p>Invokes the <code>bdf2pcf</code> command to create the <code>.pcf</code> files needed for X and Motif applications.</p> <p>When you use this option, the <code>cgen</code> command also invokes the <code>mkfontdir</code> and <code>xset</code> commands to make the fonts known to the font server and available to applications.</p>
<code>-pre</code>	<p>Creates preload font files.</p> <p>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.</p>
<code>-odl</code>	<p>Creates ODL font files.</p> <p>The terminal driver handles loading of fonts from ODL font files on an incremental basis, according to need and available memory.</p>

6.9.3 Processing UDC Fonts for Use with DECwindows

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 DECwindows applications. The

`fontconverter` command performs this conversion and can do one of two things with the converted output:

- Create independent `pcf` and `bdf` font files, which you must then install on your workstation for application use
- Merge the fonts into an existing DECwindows (`pcf`) font file

The remainder of this section discusses the `fontconverter` command and when to use its available 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.

6.9.3.1 Using `fontconverter` Command Options

The following example shows the simplest form of the `fontconverter` command, which relies on defaults for file locations, output file names, input file name extensions, and font dimensions. 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` utility when the font glyphs were created.

```
% fontconverter my_fonts
```

The preceding command converts fonts in the `~/font/my_fonts.pre` file. By default, the command creates the font files `~/font/jisx.udc_24_24.pcf` and `~/font/jisx.udc_24_24.bdf`. For the fonts to be available to applications, you can 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`
```

These commands make the fonts available for testing until a server restart or system shutdown occurs.

Including the `-bdf` and `-pcf` options on the `cgen` command line is a one-step alternative to executing the `fontconverter` and the preceding commands as separate operations.

- Perform the following actions to make the fonts available on a more permanent basis (that is, after a server restart or system shutdown):
 1. Copy the `.pcf` fonts to an existing font directory, for example, `/usr/i18n/usr/lib/X11/fonts/decwin/100dpi`.


```
% cp ~/.fonts/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:


```
% /usr/bin/X11/xset fp rehash
```

Table 6–15 lists and describes options of the `fontconverter` command. With the exception of `-preload`, the options are listed in command-line order. See Section 6.9.3.2 for examples that use these options.

Table 6–15: Options and Arguments of the `fontconverter` Command

Argument or Option	Description
<code>-merge</code>	Specifies that command output be merged with an existing DECwindows font file. See also the entry for the <code>-font</code> option.
<code>-w</code>	Specifies the font width. Use this option when the fonts are created with a width smaller than the one specified for the <code>cedit</code> font editing window.
<code>-h</code>	Specifies the font height. Use this option when the fonts are created with a height smaller than the one specified for the <code>cedit</code> font editing window.

Table 6–15: Options and Arguments of the fontconverter Command (cont.)

Argument or Option	Description
<code>-udc <i>udc_font</i></code>	<p>Specifies the name of the output UDC font file. This name is also used in the XLFD (X Logical Font Description) as the registry name. 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 name. The default base names for output files vary according to language, as follows:</p> <ul style="list-style-type: none">• Japanese: <code>jisx.udc</code>• Hanyu: <code>dec.cns.udc</code>• Hanzi: <code>gb.udc</code> <p>The <code>fontconverter</code> command automatically appends font width and height to the output file base name.</p>
<code>-font <i>reference_font</i></code>	<p>Specifies a reference DECwindows font file. If you use this option with the <code>-merge</code> option, <i>reference_font</i> indicates the <code>pcf</code> format file with which converted font glyphs are merged. If you use this option without the <code>-merge</code> option, the header of <i>reference_font</i> is used as a reference for generating the header of the standalone output file. Information in <i>reference_font</i> 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.</p>
<code>-preload <i>preload_font</i></code>	<p>Specifies the input file (created by the <code>cgen -pre</code> command). Use this option when you want to specify the <i>preload_font</i> argument at an arbitrary position in the <code>fontconverter</code> command line. You can omit <code>-preload</code> when placing <i>preload_font</i> at the end of the command line.</p>

6.9.3.2 Controlling Output File Format

X and DECwindows 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 converted fonts are merged into the `pcf` file specified by the `-font` option and a `bdf` file is not created.

When you merge UDC fonts with standard DECwindows fonts, you can use the combined file with all DECwindows 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 file as compared to when you create independent files.

The following example:

1. Converts preload format fonts in the file `udc_font.pre`
2. Merges output into the DECwindows font `JISX.UDC_*`
3. Generates the output file `JISX.UDC_16_16.pcf`

```
% fontconverter -merge -font 'JISX.UDC_*' \  
udc_font.pre
```

The following command:

- Creates the files `dekanji.udc_16_16.bdf` and `dekanji.udc_16_16.pcf`
- Obtains the default characters and most header information for these files from the DECwindows font `JISX.UDC_*`
- Sets the font registry field to `dekanji.udc`

```
% fontconverter -udc dekanji.udc -font \  
'JISX.UDC_*' udc_font.pre
```

6.10 Setting Up and Using the Chinese Phrase Input Method

In Korea, Taiwan, and China, users can input a complete phrase by typing a keyword, abbreviation, or acronym. This capability is provided by a phrase database and one of the following:

- The VT382-D Traditional Chinese terminal

When using this terminal, a phrase database is loaded in its entirety to the terminal. Memory limitations restrict the size of the database to 100 phrases. The last line on the screen (line 26) is reserved for different input methods, phrase input being one of them, and users are prompted to enter phrase codes on this line.

- The SIM (Software Input Method) service

This service, which is enabled through the `-adec` option of the `stty` command, extends support of phrase input to other Asian terminals in the VT382 series. The SIM service loads phrases dynamically to the terminal; therefore, the size of the phrase database is not limited by memory restrictions of terminal hardware. When using a terminal supported by the SIM service, you press a user-defined key sequence to toggle in and out of phrase input mode. Entering phrase input mode shifts the site of user input to the 26th line of the terminal screen where you are prompted to enter phrase codes.

- The phrase input mechanism available in the DECwindows Motif environment on workstations

DECterm windows do not implement the 26th line of a terminal screen, so the SIM service does not work correctly on workstations. Phrase input, along with other kinds of input methods, is supported by the input method server for the Chinese and Korean languages. On workstations, you enter phrases by invoking the Input Method window and selecting the phrase item.

The `phrase` utility allows you to create and maintain a phrase database and, when using the VT382-D terminal, to load the database to the terminal.

Table 6-16 lists and describes basic terms associated with phrase input.

Table 6–16: Chinese Phrase Input Definitions

Term	Description
phrase	The string for the phrase that the user wants to retrieve. Each phrase is a string of any characters in the codeset of the current locale and can be a maximum of 80 bytes in length.
phrase code	The keyword entered by the user to retrieve a phrase. Each phrase code is a string of up to 8 ASCII alphanumeric characters.
class	A group of logically related phrases. Each class has an identifier that is a string of up to 8 ASCII characters.
database	<p>A set of two files: the phrase data file <code>phrase.dat</code> and the class data file <code>class.dat</code>. If a phrase database is moved from one directory to another, the two data files must be moved together.</p> <p>There are two types of phrase databases: system and user. The system database is shared by all users on the system and is maintained by the system administrator. User databases are defined and maintained by individual users.</p> <p>Pathnames for the system and user phrase database directories are set in the file <code>/var/i18n/conf/cp_dirs</code>, which is described in a subsequent section of this chapter. By default, this file sets the pathname for the system phrase database directory to be <code>/var/i18n/sim</code> and for the user phrase database directory to be <code>\$HOME/.sim</code>.</p> <p>Phrase database files are locale specific and reside in locale directories subordinate to the default path. For example, an individual user might create and maintain the following sets of files to support two different locales:</p> <pre>\$HOME/.sim/zh_TW.big5/phrase.dat \$HOME/.sim/zh_TW.big5/class.dat \$HOME/.sim/zh_TW.dechanyu/phrase.dat \$HOME/.sim/zh_TW.dechanyu/class.dat</pre>

6.10.1 Enabling the SIM Service

Table 6–17 lists and describes the options on the `stty` command line that enable and set certain characteristics for Chinese phrase input through the VT382 series of Asian terminals. These options do not apply to DECterm windows, for which phrase input is supported using mechanisms other than SIM.

Table 6–17: The stty Options Used for the SIM Service

stty Option	Description
<code>sim</code>	Enables the Software Input Method (SIM) service.
<code>-sim</code>	Disables the Software Input Method (SIM) service.
<code>simkey <i>key</i></code>	Sets the toggle key for entering phrase input mode.
<code>simclass <i>class</i></code>	Sets the current class name for locating the appropriate phrase in the phrase database. Classes identify subsets of information in the phrase database and are defined by using the <code>phrase</code> utility.
<code>simdb <i>path</i></code>	Sets the path for the phrase database.
<code>simall</code>	Displays current SIM service settings.

6.10.2 Creating and Maintaining a Chinese Phrase Database

You can create or maintain a phrase database by using the `phrase` utility. On workstations, you invoke this utility with the following command:

```
% phrase
```

The command assumes that you are using a private phrase database if you are a nonprivileged user and the systemwide phrase database if you are superuser. You can change these defaults by using the utility's menu interface.

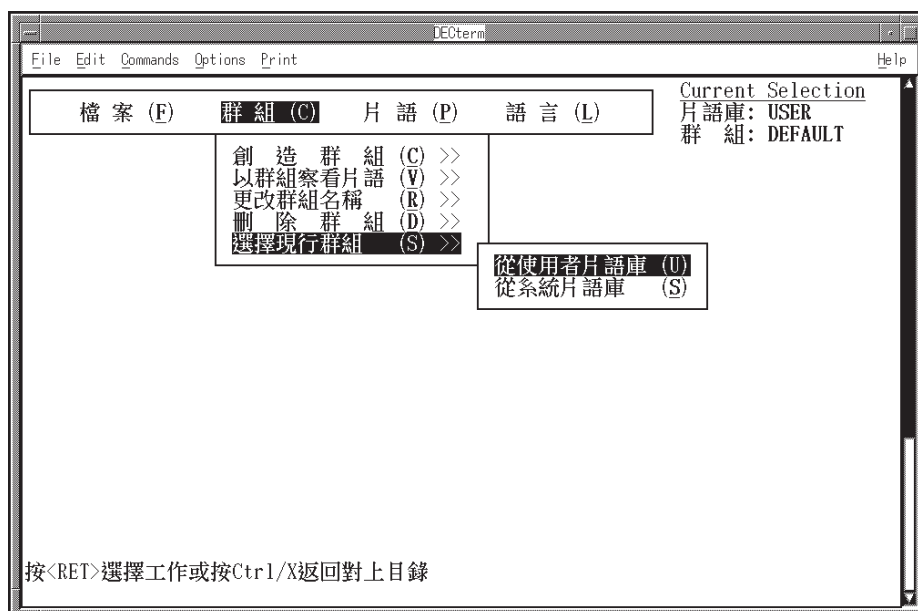
If you are working on a VT382-D traditional Chinese terminal, you may also include one of the options described in Table 6–18. These options allow you to use the hardware phrase input method supported by your terminal.

Table 6–18: The phrase Command Options for the VT382-D Terminal

phrase Option	Description
<code>-user class_name</code>	Downloads the phrase definitions for the specified class from your private phrase database to the terminal.
<code>-system class_name</code>	Downloads the phrase definitions for the specified class from the systemwide phrase database to the terminal.

On startup, the `phrase` utility displays a full-screen, menu-driven interface like the one in Figure 6–6.

Figure 6–6: User Interface Screen of the phrase Utility



Take the following steps to change the language of messages and other text on the user interface to English:

1. Press the L key.

This action displays items on the LANGUAGE menu.

2. Press the E key.

This action specifies English for the user interface.

The `phrase` utility is a `curses` application. To navigate the `phrase` utility user interface, use the following guidelines:

- Select a menu and menu items without activating them by using the arrow keys.
- Press either Return or the space bar to activate the selected menu or menu item.
- To select and activate in one operation, press the key for the underlined letter in the name of a menu or menu item, depending on your current level in the menu hierarchy.
- Press `Ctrl/X` to return to a higher level of the menu hierarchy without activating a selection.

Pressing `Ctrl/X` when a menu is not activated causes the `phrase` utility to exit.

The `phrase` user interface screen includes:

- A menu bar (upper-left corner of the screen)
- An area that specifies the current `phrase` database and class (to the right of the menu bar)
- Two lines for warning and informational messages (bottom of screen)
- A large area for menu expansion and user dialog (center of screen)

The different menus allow you to perform the following operations:

- FILE menu
 - Override the default path for the `phrase` database with which you want to work
 - Load phrases to a VT382D terminal
 - Exit from the `phrase` utility and save any changes made to the database
- CLASS menu
 - Create a class

- View phrases in the selected class
- Rename a class
- Delete a class
- Select (change) the current class
- PHRASE menu
 - Create a phrase within the selected class
 - If you do not explicitly select a class, class DEFAULT is assumed.
 - Modify a phrase
 - Delete a phrase
- LANGUAGE menu
 - Choose English or Chinese as the language in which screen text and messages appear

The following guidelines and restrictions apply to the phrase-management operations that you can perform:

- Creating and maintaining phrases
 - Phrases are always manipulated within the context of a phrase class. If you have not explicitly selected a class, the phrase is assumed to be in class DEFAULT. Otherwise, the phrase applies to the last class name you explicitly selected.
 - When you choose options that manipulate phrase definitions, a two-part window appears. The left side displays phrase codes while the right side displays phrases.

You input phrase names and definitions in an area below the two-part display window. Choose your phrase name carefully. This is the code used to invoke the phrase later. You cannot modify the phrase name without deleting and reentering the entire phrase definition.
 - Phrase names must be unique within a given class, but you can use the same phrase name in different phrase classes.
 - The phrase itself can contain up to 80 bytes of data, which correspond roughly to 80 columns on the screen. All 80 bytes of data appear in the user input area; however, the display window

provides fewer than 80 columns to display the phrase. As a result, long phrase definitions are truncated at the right boundary of the display window. In such cases, the right angle bracket (>) appears in the rightmost position to indicate that the phrase definition contains more data. This truncation is a restriction of the display window and does not apply to the phrase when it is invoked.

- Creating and maintaining classes
 - Classes are created and maintained within the context of a particular database. If you have not explicitly specified a database, the class operation applies to your default database.
 - Class names must be unique within a database.
 - Creating a new class causes that class to be the selected class and then automatically invokes the function to create new phrases for the class.
 - The hardware phrase input method used on the VT382D terminal can load up to 100 phrases in a class. Keep this limitation in mind if you use one of these terminals or are maintaining a database accessed by others who log on through terminals.

There are no restrictions on the number of phrases in a class when phrases are retrieved through other Asian terminals in the VT382 series or through the Input Method window in the DECwindows Motif environment.

- Using multiple phrase databases
 - Phrase databases are locale specific. You cannot invoke the phrase utility without setting the `LANG` environment variable to a locale; however, you can create phrase databases for any locale. Be sure that the `LANG` environment variable is set to the locale you want to create phrases for before invoking the `phrase` utility. Otherwise, you will be working with (or creating) phrase databases for a locale different from the one you want.
 - You can copy phrase definitions to your private database from the systemwide database and from databases of other users (assuming their file protections allow you read access). If you choose to copy phrases from another user's database, you are prompted for the absolute path of the database from which you want to copy. If the

- specified database is accessible to you, all its phrase definitions are listed and you select the ones you want to copy.
- You must own a database to create, delete, or modify classes in that database. Unprivileged users can perform write operations on their private databases. Only the superuser can perform write operations on the systemwide database.

6.10.3 Using a Chinese Phrase Database

How you use a phrase database depends on whether you are using the hardware input method or the software input method (SIM) service. You can use either the hardware input method or SIM service on a VT382D Traditional Chinese terminal. For other terminals in the VT382 series of Asian terminals or for a DECterm window on a workstation, you use the SIM service.

If you are using the hardware input method with a VT382D Traditional Chinese terminal, refer to your terminal user guide for phrase input instructions.

6.10.3.1 Phrase Input Supported Through the SIM Service

Before you can use a phrase database, you use the `stty` command to:

- Enable the SIM service

```
% stty sim
```

To enable the SIM service, make sure your locale is set to one that supports the Hanzi, Hanyu, or Korean codeset and that your terminal line discipline is set to `adec`.

- Define the key sequence for toggling in and out of phrase input mode

The following example sets this key sequence to be `Ctrl/B`:

```
% stty simkey ^B
```

When you define the key sequence to toggle in and out of phrase mode, pick one that you do not already use at the command line or in other applications. For example, do not define the key sequence to be `Ctrl/C` (abort operation) or `Ctrl/Z` (suspend operation).

If you do not want to use phrases from the class `DEFAULT` or from your default phrase database, use the `stty` command to:

- Specify the phrase class that the SIM service or specialized terminal software will use to interpret phrase codes

```
% stty simclass CORP
```

- Specify the database that specialized terminal software will access

The SIM service always searches your private phrase database first for a phrase name and, if the name is not found, then searches the systemwide phrase database. However, terminals that support the hardware phrase input method can load phrases from only one database at a time. Therefore, a nonprivileged user using the terminal hardware input method might enter the following command:

```
% stty simdb /var/i18n/sim
```

When the terminal setup is complete, you can perform the following actions to retrieve a phrase:

1. Press the key sequence specified for the `simkey` option of the `stty` command.

```
% Ctrl/B
```

At the bottom of your screen, you are then prompted to enter a phrase code.

2. Type the phrase code and then press either Return or the space bar.

The phrase is returned to the screen or, if the phrase code was not found, an error message appears.

When you want to exit from phrase input mode, press the `simkey` key sequence again.

While in phrase input mode, the characters that you enter are subject to the following rules:

- Lowercase alphanumeric characters, which are valid characters for phrase codes, are converted to uppercase.

- A space or Return character entered when the phrase code buffer is empty is sent directly to the application from which you entered phrase input mode.

This behavior means that you do not have to exit from phrase mode to enter a space or newline between phrases.

- If you enter printable characters other than alphanumeric ones, the bell rings to signal that they are invalid characters for a phrase code.
- Control key sequences other than the one used to toggle in and out of phrase mode are sent directly to the application from which you entered phrase input mode.

This behavior means that control sequences such as `Ctrl/Z` and `Ctrl/C` are handled as you would expect for the system command line, editor, or other application where the phrases are being entered.

- Pressing a function or arrow key produces undefined results.

6.10.3.2 Phrase Input in the DECwindows Motif Environment

When phrase input is supported by your language setting and the associated input method server is running, your DECwindows Motif environment includes an Input Options window. Click on the Options button in this window to:

- Select the phrase database (user or system)
- Select the phrase class within the database
- Start phrase input

To start phrase input, select Input Method Customization from the Input Options menu and, in the pop-up dialog box, select Phrase.

6.11 Modifying the Database Location Configuration File

This section discusses the content and format of the file `/var/il8n/conf/cp_dirs`. Software services or hardware use this file to locate various kinds of databases that support input of Asian user-defined characters and phrases.

Example 6-2 shows the default entries in the `cp_dirs` file. You can edit these entries to change the default locations.

Example 6–2: Default cp_dirs File

```
#
# Attribute directory configuration file
#
#           System location           User location
#           =====
udc  -           /var/il8n/udc           ~/.udc
odl  -           /var/il8n/odl           ~/.odl
sim  -           /var/il8n/sim           ~/.sim
cdb  /usr/il8n/.cdb /var/il8n/cdb           ~/.cdb
iks  -           /var/il8n/iks           ~/.iks
pre  -           /var/il8n/fonts          ~/.fonts
bdf  -           /var/il8n/fonts          ~/.fonts
pcf  -           /var/il8n/fonts          ~/.fonts
```

Each line in the `cp_dirs` file represents one entry and consists of 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 files in `pcf` 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 pathname (starting with `~/`) that is relative to a user's home directory
- `-` (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 (`#`).

6.12 Using Printer Interface Features That Support Local Languages

When you install Digital UNIX language variant subsets, your printing subsystem is enhanced with the following features:

- A set of print filters that support escape sequences used by local-language printers
- Entries in the `/etc/printcap` file to support printer code conversion and on-demand loading of font files
- An enhanced `lprsetup` command that lets you add entries for local-language printers to the `/etc/printcap` file
- `lp`, `lpr`, `lpc`, `lpq`, `lprm`, and `lpstat` commands that support additional options for printing and printer control
- Support for on-demand loading in the `lpd` printer daemon
- The `pfsetup` command and associated daemon for downloading fonts to PostScript printers

The following sections discuss these features.

6.12.1 Print Filters for Local Language Printers

A print filter processes text data for a particular model of printer. The filter handles the device dependencies of the printer and performs device accounting functions. When each print job is completed, the print filter writes an accounting record to the file specified by the `af` field of the printer's entry in the `/etc/printcap` file.

The print filters for local-language text printers can handle text files that contain ASCII and local-language characters, or output files created by the `nroff` command. When processing `nroff` output, the filter removes multibyte characters that extend beyond the page boundary and translates `nroff` control sequences for underlining, superscripting, and subscripting to control sequences appropriate for the printer. However, the filter does not support multiple `nroff` control sequences on the same character.

The PostScript print filters can print PostScript files in addition to text and `nroff` output files. The memory requirement for some Asian fonts exceeds what is available on most printers, so there are specific font-loading mechanisms for loading these fonts on PostScript printers (see Section 6.12.5).

A local-language print filter can be the specified filter in both the `of` and `if` fields in the `/etc/printcap` file. For general information on `/etc/printcap` entries, refer to *System Administration* and the `printcap(4)` reference page. Supplementary information is provided in the `il8n_printing(5)` reference page. A reference page for a specific language (for example, `Japanese(5)`) lists the names of print filters that support printing characters in that language.

The following print filters process text data for Asian languages:

Language	Filter	Printer
Japanese	1a84of	LA84-J
Japanese	1a86of	LA86-J
Japanese	1a90of	LA90-J
Japanese	1a280of	LA280-J
Japanese	1a380of	LA380-J
Japanese	1n03jaof	LN03-J
Japanese	1n05jaof	LN05-J

Language	Filter	Printer
Hanzi	la88cof	LA88-C
Hanzi	la380cbof	LA380-CB
Korean	la380kof	LA380-K
Korean	dl510kaof	DL510-KA
Hanyu	cp382dof	CP382-D
Thai	thailpof	EP1050+

The following print filters process PostScript and text data for Asian languages and for some of the languages supported by locales using the ISO8859-2, ISO8859-5, ISO8859-7, and ISO8859-9 codesets:

Language	Filter	Printer
Japanese	ln82rof	LN82R
Czech, Hanyu, Hanzi, Hungarian, Greek, Korean, Polish, Russian, Slovak, Slovene, and Turkish	dl1152wrof	DEClaser 1152
Thai	dl1152trof, dl1152ttmrof	DEClaser 1152
Czech, Hanyu, Hanzi, Hungarian, Greek, Korean, Polish, Russian, Slovak, Slovene, and Turkish	dl5100wrof	DEClaser 5100
Thai	dl5100trof, dl5100ttmrof	DEClaser 5100

See the reference page for a specific language (for example, `Japanese(5)`) to find the names of print filters that support printing characters in that language.

6.12.2 Support for Local Language Printers in `/etc/printcap`

The `/etc/printcap` file describes characteristics of each printer on the system. Printer characteristics are specified by symbol/value pairs, where

each symbol is a 2-character mnemonic. Each time a user submits a print job, the `lpd` printer daemon and printer spooling system uses information in the `/etc/printcap` file to determine how that job is handled.

Table 6–19 lists and describes `/etc/printcap` symbols that are specific to support for local-language printers. Refer to `printcap(4)` for descriptions of other symbols used in the `/etc/printcap` file. Refer to Section 6.12.3 for an example of using the `lprsetup` command to add several of these options to the `/etc/printcap` for a local-language printer.

Note

Some printers, such as the DEClaser 5100, support printing in a variety of Asian languages. In such cases, if you want to use the same printer for printing different Asian languages, the following restrictions apply:

- You must set up different print queues and use different spool areas for each language.
- You must differentiate the queues by defining different `sd` (spool directory) and `plocale` (printer locale) values in the `/etc/printcap` file.

If the preceding requirements are not met, files may occasionally be printed in the wrong locale, resulting in meaningless output. There is one problem that can result from setting up multiple print queues and directories for the same printer. If two or more jobs are sent to different queues for the same printer within a very short time, some jobs may be blocked so that they do not print. If this happens, the system manager must use the `lpc` command to restart the blocked jobs.

Table 6–19 lists and describes `/etc/printcap` symbols that are specific to local-language requirements.

Table 6–19: Symbols in /etc/printcap File for Local Language Printers

Symbol	Type	Default	Description
ya	str	None	Double-quoted list of keyword value assignments. This assignment list specifies most of the printer options related to country-specific support. The option keywords, which are explained following this table, include <code>flocale</code> , <code>font</code> , <code>line</code> , <code>odldb</code> , <code>odlstyle</code> , <code>onehalf</code> , <code>plocale</code> , <code>spcom</code> , <code>tacdata</code> , and <code>tm</code> .
yd	str	None	Secondary tty line or channel for font faulting Specify this entry for the DEClaser 1152 printer to support the font-faulting mechanism. The font-faulting mechanism, which is enabled by the <code>alpc</code> and <code>ffserver</code> commands, allows the printer to use fonts that are installed but not downloaded. Font faulting is required to support Chinese, Korean, and some other fonts. The font-faulting daemon (<code>ffd</code>) uses the secondary tty line to send font information to the printer.
yj	str	NULL	If <code>on</code> is specified as a value, restarts the filter specified for the <code>of</code> symbol for every print job. You need to define this symbol only for printers that are not country-specific and only if non-ASCII characters need to be printed on the flag page of printed output.

Table 6–19: Symbols in /etc/printcap File for Local Language Printers (cont.)

Symbol	Type	Default	Description
<code>ys</code>	num	NULL	<p>Size of the SoftODL character cache</p> <p>The <code>ys</code> entry is applied to text print filters. It must be present and its value must be greater than zero to enable on-demand loading of font files. These font files are the ODL support files created by the <code>cgen</code> utility for user-defined characters. The location of the SoftODL support files is identified by the path for systemwide ODL files in the database location configuration file <code>/usr/var/i18n/conf/cp_dirs</code>. ODL files for private UDC databases are not downloaded to printers.</p> <p>For optimal performance, the cache value specified for the <code>ys</code> field should match the printer cache size. To find out the cache size for a particular printer, refer to the printer's manual.</p>
<code>yt</code>	str	<code>fifo</code>	<p>The SoftODL character replacement method</p> <p>The <code>yt</code> entry applies to text print filters. The value for this entry can be either <code>fifo</code> (first-in-first-out) or <code>lru</code> (least recently used). You can type either uppercase or lowercase letters for these values. To find out which value is appropriate for a particular printer, refer to the printer's manual.</p>

The value assigned to the `ya` symbol is a quoted string that can include one or more of the following options:

- `flocale=locale_name`

Specifies the locale for interpretation of file text. The print filter uses this locale to validate characters in the text. For an Asian language that is supported by more than one codeset, a difference between the `flocale` and `plocale` values determines whether codeset conversion is done before the file is printed. If `flocale` is not specified, the filter interprets the file in the current locale.

- `font=font_name`

Specifies the name of the outline font for printing PostScript files. This font must be appropriate for the specified `plocale` value.

- `line=number_of_lines`

Specifies the number of lines per page. When used in combination with the `-w` flag of the `lpr` command, the line number can control the font size and orientation of printed output.

- `odldb=odl_database_path`

Specifies the pathname of the software on-demand (SoftODL) database. By default, the printer uses the systemwide database as specified in the `cp_dirs` file.

- `odlstyle=style-NxN`

Specifies the SoftODL font style and size to use, for example `normal-24x24`. If `odlstyle` is not specified, the default style and size set for the systemwide database is used.

- `onehalf`

For the Thai language, specifies that characters be printed on one and a half lines, rather than three lines, to produce more compressed and natural looking output. The `onehalf` option is valid only for the `thailpof` print filter.

- `plocale=locale_name`

Specifies the printer locale. Some printers, such as the LA380-CB printer, are country-specific and have built-in fonts that are encoded in a particular codeset. For these printers, the codeset part of `locale_name` should match the codeset of the built-in fonts. Other printers, such as the DEClaser 5100, are generic and suitable for printing files in a variety of languages. For these printers, the codeset part of `locale_name` should match the codeset of the font needed to print files in a particular language (or set of languages). Remember that to use the same generic printer for printing files in different languages, you must define a separate print queue and spool directory for each language (codeset) in which print jobs will be submitted.

- `spcom`

Enables space-compensation mode for languages, such as Thai, that contain nonspacing characters. These characters can combine with other characters for display and therefore do not occupy space. Many of the existing tools that align text do not handle nonspacing characters correctly. If you want to print the Thai output that these tools generate, you should specify the `spcom` option to ensure proper text alignment in the printed file. This option is valid only when used with a Thai print filter or the `th_TH.TACTIS plocale` value.

- `tacdata=tac_data_path`

Specifies the location of the character code tables used with the `thailpof` print filter. By default, `tac_data_path` is `/usr/lbin/tac_data`.

- `tm`

Enables text morphing for printing Thai characters. Text morphing replaces some characters with others to produce better printed output. Refer to the `Thai(5)` reference page for information on text morphing.

6.12.3 Enhancements to the `lprsetup` Command

The `lprsetup` command helps you manage the printers on your system. The command queries you for answers to questions about adding, deleting, or changing the characteristics of any printers on your system. The questions have default answers, which are delimited by brackets (`[]`). Online help is available for each question. Either press only the Return key to choose the default answer or enter a valid alternative. Follow instructions displayed by `lprsetup` to see the help message for each question.

After you enter characteristics for a particular printer and verify that your entries are correct, the `lprsetup` command creates the printer spooling directory, links the filters, and writes the entry for the printer in the `/etc/printcap` file.

Example 6–3 shows how you use the `lprsetup` command to set up a local-language printer, in this case, `ln05ja`.

Example 6–3: Setting Up a Local Language Printer with `lprsetup`

```
# /usr/sbin/lprsetup 1
Digital OSF/1 Printer Setup Program

Command < add modify delete exit view quit help >: add

Adding printer entry, type '?' for help.

Enter printer name to add [0] : ln05 2

For more information on the specific printer types Enter
'printer?'

Enter the FULL name of one of the following printer
types:
```

Example 6–3: Setting Up a Local Language Printer with lprsetup (cont.)

```
cp382d  dl1152w  dl510ka   dl5100w  ep1050+  fx80    fx1050   hpIIP
hpIIIP  hpIIID  hpIV    hp4M    ibmpro   la50    la70     la75
la84    la86    la88    la88c   la90     la280   la324    la380
la380cb la380k  la424   lf01r   lg02     lg06    lg12     lg31
lj250   ln03    ln03ja  ln03r   ln03s    ln05    ln05ja   ln05r
ln06    ln06r   ln07    ln07r   ln08     ln08r   ln09     ln10ja
ln82r   nec290  remote  unknown
```

or press RETURN for [unknown] : **ln05ja** **3**

:

Enter the name of the printcap symbol you wish to modify.
Other valid entries are:

```
'q' to quit (no more changes)
'p' to print the symbols you have specified so far.
'l' to list all of the possible symbols and defaults.
```

The names of the printcap symbols are:

```
af  br  cf  ct  df  dn  du  fc  ff  fo  fs  gf  ic  if  lf  lo
lp  mc  mx  nc  nf  of  op  os  pl  pp  ps  pw  px  py  rf  rm
rp  rs  rw  sb  sc  sd  sf  sh  st  tf  tr  ts  uv  vf  xc  xf
xs  ya  yd  yj  yp  ys  yt  Da  Dl  It  Lf  Lu  Ml  Nu  Or  Ot
Ps  Sd  Si  Ss  Ul  Xf
```

Enter symbol name: **ya** **4**

Enter a new value for symbol 'ya'? ["plocale=ja_JP.sdeckanji"]

Do you want to enable ODL? [n] **y** **5**

Enter symbol name: **yt** **6**

Enter a new value for symbol 'yt'? [fifo]

Enter symbol name: **q** **7**

:

- 1** Invokes the `lprsetup` program.
- 2** Selects a name for the printer (see Table 6–20).
- 3** Selects the printer type.
- 4** Specifies the printer locale.
- 5** Enables on-demand loading (ODL) of printer fonts for user-defined characters. An affirmative response also sets the cache size that the

SoftODL service uses. This value, by default the appropriate cache size for the printer, is stored as value of the `ys` symbol in the `/etc/printcap` file.

- 6 Specifies the character replacement method that the SoftODL service uses.
- 7 Quits the program to indicate no more changes are needed to the `/etc/printcap` file.

Table 6–20 lists Asian languages and the associated printer choices as displayed by the `lprsetup` script.

Table 6–20: Local Language Printers Supported by the `lprsetup` Command

Language	Printer
Japanese (text only)	la84j, la86j, la90j, la280j, la380j, ln03ja, ln05ja,
Japanese (PostScript)	ln83r
Traditional Chinese (text only)	cp382d
Simplified Chinese (text only)	la88c, la380c
Korean (text only)	la380k, dl510k
Czech, Hanyu, Hanzi, Hungarian, Greek, Korean, Polish, Russian, Slovak, Slovene, and Turkish (PostScript)	dl1152w, dl5100w
Thai (text only)	dp1050+
Thai (PostScript)	dl1152t, dl1152ttm, dl5100t, dl5100ttm

6.12.4 Printing Commands and Printer Daemon

The versions of the `lp`, `lpc`, `lpq`, `lpr`, `lprm`, and `lpstat` commands with support for local-language printers use the same syntax as the default versions of the commands. The same is true for the `lpd` program that is indirectly invoked to run the printer daemon. The difference between the default and enhanced versions of the commands and printer daemon is that

the enhanced versions can handle features added to the print subsystem for Asian languages.

6.12.5 Font Handling for PostScript Printers

The fonts for the Chinese or the Korean language do not fit in the memory of most PostScript printers. Fonts for the Thai language and some European languages do fit in memory, but are large enough that they cannot be downloaded together with fonts for other languages. Therefore, Digital UNIX provides a mechanism for selectively downloading fonts to PostScript printers. This section lists the fonts available to support different locales. It also explains how the font mechanism works and how you set up DEClaser 1152 and DEClaser 5100 printers to use fonts for languages that require specialized printer support.

6.12.5.1 Choosing Fonts for Different Locales

To support locales for most Asian and some European languages, you must set up PostScript printers so that they can use the appropriate fonts. Fonts for locales based on the ISO8859-1 and Kanji codesets are fonts that are built in to the DEClaser 1152 and DEClaser 5100 printers. However, locales based on other codesets require fonts that need to be downloaded to the printer. Section 6.12.5.2 discusses font downloading in more detail. The following list associates languages and codesets with the appropriate set of fonts:

- Hungarian, Czech, Slovak, Slovene (*.ISO8859-2)

```
Arial-Bold-ISOLatin2
Arial-BoldItalic-ISOLatin2
Arial-Italic-ISOLatin2
Arial-ISOLatin2
ArialNarrow-Bold-ISOLatin2
ArialNarrow-BoldItalic-ISOLatin2
ArialNarrow-Italic-ISOLatin2
ArialNarrow-ISOLatin2
BookAntiqua-Bold-ISOLatin2
BookAntiqua-BoldItalic-ISOLatin2
BookAntiqua-Italic-ISOLatin2
BookAntiqua-ISOLatin2
BookmanOldStyle-Bold-ISOLatin2
BookmanOldStyle-BoldItalic-ISOLatin2
BookmanOldStyle-Italic-ISOLatin2
```

BookmanOldStyle-ISOLatin2
CenturyGothic-Bold-ISOLatin2
CenturyGothic-BoldItalic-ISOLatin2
CenturyGothic-Italic-ISOLatin2
CenturyGothic-ISOLatin2
CenturySchoolbook-Bold-ISOLatin2
CenturySchoolbook-BoldItalic-ISOLatin2
CenturySchoolbook-Italic-ISOLatin2
CenturySchoolbook-Italic-ISOLatin2
CenturySchoolbook-ISOLatin2
Courier-Bold-ISOLatin2
Courier-BoldItalic-ISOLatin2
Courier-Italic-ISOLatin2
Courier-ISOLatin2
MonotypeCorsiva-ISOLatin2
TimesNewRoman-Bold-ISOLatin2
TimesNewRoman-BoldItalic-ISOLatin2
TimesNewRoman-Italic-ISOLatin2
TimesNewRoman-ISOLatin2

- **Russian (*.ISO8859-5)**

Arial-Bold-ISOLatinCyrillic
Arial-BoldInclined-ISOLatinCyrillic
Arial-Inclined-ISOLatinCyrillic
Arial-ISOLatinCyrillic
Courier-Bold-ISOLatinCyrillic
Courier-BoldInclined-ISOLatinCyrillic
Courier-Inclined-ISOLatinCyrillic
Courier-ISOLatinCyrillic
Nimrod-Bold-ISOLatinCyrillic
Nimrod-BoldInclined-ISOLatinCyrillic
Nimrod-Inclined-ISOLatinCyrillic
Nimrod-ISOLatinCyrillic
Plantin-Bold-ISOLatinCyrillic
Plantin-BoldInclined-ISOLatinCyrillic
Plantin-Inclined-ISOLatinCyrillic
Plantin-ISOLatinCyrillic
TimesNewRoman-Bold-ISOLatinCyrillic
TimesNewRoman-BoldInclined-ISOLatinCyrillic
TimesNewRoman-Inclined-ISOLatinCyrillic
TimesNewRoman-ISOLatinCyrillic

- **Greek (*.ISO8859-7)**

Arial-Bold-ISOLatinGreek
Arial-BoldInclined-ISOLatinGreek

Arial-Inclined-ISOLatinGreek
Arial-ISOLatinGreek
Courier-Bold-ISOLatinGreek
Courier-BoldInclined-ISOLatinGreek
Courier-Inclined-ISOLatinGreek
Courier-ISOLatinGreek
TimesNewRoman-Bold-ISOLatinGreek
TimesNewRoman-BoldInclined-ISOLatinGreek
TimesNewRoman-Inclined-ISOLatinGreek
TimesNewRoman-ISOLatinGreek

- **Turkish (* .ISO8859-9)**

Arial-Bold-ISOLatin5
Arial-BoldItalic-ISOLatin5
Arial-Italic-ISOLatin5
Arial-ISOLatin5
ArialNarrow-Bold-ISOLatin5
ArialNarrow-BoldItalic-ISOLatin5
ArialNarrow-Italic-ISOLatin5
ArialNarrow-ISOLatin5
BookAntiqua-Bold-ISOLatin5
BookAntiqua-BoldItalic-ISOLatin5
BookAntiqua-Italic-ISOLatin5
BookAntiqua-ISOLatin5
BookmanOldStyle-Bold-ISOLatin5
BookmanOldStyle-BoldItalic-ISOLatin5
BookmanOldStyle-Italic-ISOLatin5
BookmanOldStyle-ISOLatin5
CenturyGothic-Bold-ISOLatin5
CenturyGothic-BoldItalic-ISOLatin5
CenturyGothic-Italic-ISOLatin5
CenturyGothic-ISOLatin5
CenturySchoolbook-Bold-ISOLatin5
CenturySchoolbook-BoldItalic-ISOLatin5
CenturySchoolbook-Italic-ISOLatin5
CenturySchoolbook-ISOLatin5
Courier-Bold-ISOLatin5
Courier-BoldItalic-ISOLatin5
Courier-Italic-ISOLatin5
Courier-ISOLatin5
MonotypeCorsiva-ISOLatin5
TimesNewRoman-Bold-ISOLatin5
TimesNewRoman-BoldItalic-ISOLatin5
TimesNewRoman-Italic-ISOLatin5
TimesNewRoman-ISOLatin5

- **Traditional Chinese (*.dechanyu)**

Sung-Light-CNS11643
Hei-Light-CNS11643

- **Simplified Chinese (*.dechanzi)**

XiSong-GB2312-80
Hei-GB2312-80

- **Korean (*.deckorean)**

Munjo

- **Japanese (*.deckanji)**

None (uses printer built-in fonts)

- **Thai (*.TACTIS)**

AngsanaUPC-Bold
AngsanaUPC-BoldItalic
AngsanaUPC-Italic
AngsanaUPC-Light
CordiaUPC-Bold
CordiaUPC-BoldItalic
CordiaUPC-Italic
CordiaUPC-Light
EucrosiaUPC-Bold
EucrosiaUPC-BoldItalic
EucrosiaUPC-Italic
EucrosiaUPC-Light
FreesiaUPC-Bold
FreesiaUPC-BoldItalic
FreesiaUPC-Italic
FreesiaUPC-Light
IrisUPC-Bold
IrisUPC-BoldItalic
IrisUPC-Italic
IrisUPC-Light
JasmineUPC-Bold
JasmineUPC-BoldItalic
JasmineUPC-Italic
JasmineUPC-Light
KodchiangUPC-Bold
KodchiangUPC-BoldItalic
KodchiangUPC-Italic

```
KodchiangUPC-Light
LilyUPC-Bold
LilyUPC-BoldItalic
LilyUPC-Italic
LilyUPC-Light
WaterlilyUPC-Bold
WaterlilyUPC-BoldItalic
WaterlilyUPC-Italic
WaterlilyUPC-Light
YuccaUPC-Bold
YuccaUPC-BoldItalic
YuccaUPC-Italic
YuccaUPC-Light
```

6.12.5.2 Setting Up Print Queues With the `pfsetup` Command

The `pfsetup` utility is available to manage font downloading for print queues. This command identifies the correct downloading mechanism through the print filter name. The `pfsetup` command has the following format:

```
pfsetup[ -s | -d ] [ queue_name ]...
```

You can use the `pfsetup` command in the following ways:

- If you enter the `pfsetup` command without options, it displays setup information for the specified or all print queues.
- The `-s` option runs the utility in setup mode. In this mode, the utility lists all printer fonts available for downloading to the specified or all print queues.
- The `-d` option runs the utility in download mode. In this mode, you can download fonts for locales that are not supported by the built-in fonts.

6.12.5.3 Downloading Fonts to the DEClaser 1152

A mechanism called font faulting works around the problem of downloading very large fonts to the DEClaser 1152 printer. Font faulting is similar to the on-demand loading (ODL) mechanism used to load user-defined characters; in other words, a subset of fonts is in the device's memory at any particular time and new fonts are swapped in as needed.

For font faulting to work, there must be two channels for printer communication. The primary channel transfers file data from the host

system to the printer. The secondary channel transfers font requests and responses between the printer and the host system. You specify the secondary channel through the `yd` entry for the printer in the `/etc/printcap` file.

When the printer receives unrecognized characters, it sends font requests through its secondary communication channel. The `ffd` daemon serves this channel and responds to the font requests from the printer. The daemon searches the font files for the requested fonts and sends back the requested data.

You can manually start and stop the `ffd` daemon with the following commands:

- `/usr/sbin/init.d/ffserver start &`
- `/usr/sbin/init.d/ffserver stop &`

You have to download at least one font using the `pfsetup` command to activate the font-faulting mechanism (refer to Section 6.12.5.1 for lists of fonts and to Section 6.12.5.2 for information about the `pfsetup` command). After the font-faulting mechanism is activated and until the printer is turned off, the mechanism automatically sends information for any font to the printer as required. Therefore, the printer can use all fonts that are installed on the printer's host system, including fonts that are not explicitly downloaded.

Note

Although the font-faulting mechanism allows the printer to use any installed font, there is some overhead cost when a print job uses fonts that are not downloaded to the printer. Therefore, Digital recommends that you use the `pfsetup` command to download fonts that print jobs most frequently use.

6.12.5.4 Downloading Fonts to the DEClaser 5100

For a DEClaser 5100 printer with a font disk, you can use the `pfsetup` command to download any fonts installed on the printer's host system. The command prompts you to verify that the printer has a font disk and then downloads the fonts you choose (refer to Section 6.12.5.1 for lists of fonts and to Section 6.12.5.2 for information about the `pfsetup` command). The number of fonts you can download is limited by the amount of space

available on the font disk. After fonts are downloaded, the printer requires no additional setup to use them.

6.13 Using Mail in a Multilanguage Environment

Digital UNIX provides enhanced versions of the following commands and utilities to handle languages based on multibyte-character codesets:

- `sendmail`
- `mailx`
- MH (mail handler)

The following sections discuss enhancements to these components, along with a discussion of codeset conversion done by the `comsat` server. Refer to the `sendmail(8)`, `mailx(1)`, `mh(1)`, `comsat(8)` reference pages for more complete software descriptions.

6.13.1 The `sendmail` Utility

The `sendmail` utility, which is a back end to several user commands, can be configured to pass only 7-bit data in accordance with the Simple Mail Transfer Protocol (SMTP) or to pass 8-bit data as required for multibyte-character support. By default, `sendmail` supports only 7-bit data. You can configure `sendmail` to pass mail messages in 8-bit format by using the `/usr/sbin/wsetup` script or, in the Common Desktop Environment, by clicking on the Mail option of the I18N Configuration application. (The navigation path to the I18N Configuration application is Application Manager -> System Administration -> Configuration -> I18N.)

Note

Digital recommends that you not configure `sendmail` to use 8-bit data format because the SMTP protocol, which is widely used, does not support this format.

6.13.2 The `mailx` Command and MH Commands

Both the `mailx` command and all applicable commands in the MH system support the conversion of mail messages between the mail interchange codeset (used to transfer messages to some hosts) and a user's application

codeset. For example, if the mail interchange codeset is ISO-2022-JP and the application codeset is eucJP, the `mailx` or MH command converts incoming messages to the Japanese EUC codeset before displaying them.

To prevent data loss, when incoming messages are stored in mail folders, the messages are encoded in the codeset in which they are received. Codeset conversion takes place when users extract or display the messages.

To communicate mail interchange code information to other systems, outgoing messages include two additional header lines like the following:

```
Mime-Version: 1.0
```

```
Content-Type: TEXT/PLAIN; charset=ISO-2022-JP
```

The `charset` field in the preceding example specifies the mail interchange codeset, in this case, ISO-2022-JP. This codeset is an ISO 7-bit state-dependent codeset for Japanese characters. Codesets other than those that are part of the ISO standard, are identified by the prefix `x-` in the codeset name. For example, when DEC Hanyu is the codeset used for mail interchange, the following header lines are included in outgoing mail messages:

```
Mime-Version: 1.0
```

```
Content-Type: TEXT/PLAIN; charset=X-dechanyu
```

The `mailx` command or MH commands use the following values (listed in order of highest to lowest priority) to determine or set the mail interchange and application codesets for a particular message:

- The mail interchange codeset applied to incoming messages is determined from:
 1. If additional header lines are present in the message, the `charset` field in the mail header
 2. The codeset specified as the systemwide mail interchange default in the file `/usr/lib/mail-codesets`

If you create this file, it contains a single entry, which is the name of a locale.

If neither of the preceding values is available, codeset conversion does not occur.

- The mail interchange codeset applied to outgoing messages is determined from:
 1. The setting of the `EXCODE` environment variable
 2. The setting of the `excode` component as defined in the file `$HOME/.mailrc` (for `mailx` users) or the file `$HOME/.mh_profile` (for users of MH commands)
 3. The content of the file `/usr/lib/mail-codesets`

If a codeset is not determined for outgoing mail interchange, the mail is sent with no codeset identifier.

- The application codeset is determined from:
 1. The setting of the `LANG` environment variable
 2. The value of the `lang` component in the `$HOME/.mailrc` file (for the `mailx` command) or the `$HOME/.mh_profile` file (for MH commands)

6.13.3 The `comsat` Server

The `comsat` server, which notifies users of incoming mail messages, always attempts to convert incoming mail messages from the mail interchange codeset to the user's application codeset. The `comsat` server uses the following values (in order of highest to lowest priority) to determine the codesets that apply to a message:

- For the mail interchange codeset
 1. The `charset` field, if included in the mail message header
 2. The codeset specified as the systemwide mail interchange default in the file `/usr/lib/mail-codesets`

If neither of the preceding values is available, codeset conversion does not occur.

- For the application codeset
 1. The application codeset defined for the `atty` driver of the user's system

2. The codeset name in the file `$HOME/.codeset_device_name`, where `device_name` is the name of the terminal device for the current session

6.14 Applying Sort Orders to Non-English Characters

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. Most often, these variations exist to offer users the choice of more than one collating sequence.

When there is more than one locale 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 usually assign a locale name with an `@` suffix only to the appropriate locale category variable (or variables). For example:

```
% 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 user-defined characters, 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 user-defined characters 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 flags supported by the `sort` command. In addition, the `asort` command includes the following flags:

- `-C`

This flag indicates that the sort operation should use special system sort tables, along with sort tables produced by the `cgen` utility to support user-defined characters. This flag overrides the sort sequence defined in the locale specified by the `LC_COLLATE` variable.

- `-v`

This flag, which you can use only when you also specify the `-C` flag, implements breadth-first sorting.

Refer to the `asort(1)` reference page for more information about using this command.

6.15 Processing Reference Pages in Languages Other Than English

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. Enhanced versions of the `nroff`, `tbl`, and `man` commands are included in Digital UNIX to support this requirement.

6.15.1 The `nroff` Command

The `nroff` command includes the following capabilities 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, refer to the language-specific user guides that are available on the CD-ROM from which you install subsets for Asian-language support.

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 you should 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 guides that are available on line when you install language variant subsets of Digital UNIX. 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 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.

6.15.2 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 along 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 text of an Asian language, you must use printer hardware that supports the language.

6.15.3 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. Refer to the `man(1)` reference page for information about redefining the `man` command search path and for more details about codeset conversion.

6.16 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.

- The locale could be one of several 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.

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 file `accounts_local`, which is encoded in the `deckorean` codeset; converts the data to the `eucKR` codeset; and appends the results to the file `accounts_central`:

```
% iconv -f deckorean -t eucKR accounts_local \
>> accounts_central
```

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. 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 adheres to 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/deckorean_eucKR` converter.

Table 6–21 specifies the codeset conversions that Digital UNIX supports for English data. The user guides for the language variant subsets include tables with codeset conversions supported for Asian languages.

For detailed information about the `iconv` command, refer to the `iconv(1)` and `iconv_intro(5)` reference pages. For information on functions that programs can use to perform codeset conversion, refer to the `iconv_open(3)`, `iconv(3)`, and `iconv_close(3)` reference pages.

Table 6–21: 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

Codeset	ASCII-GR	ISO8859-1	ISO8859-1-GL	ISO8859-1-GR
ISO8859-1-GL	No	Yes	–	No
ISO8859-1-GR	No	Yes	No	–

6.17 Miscellaneous Information for Base System Commands

The following list includes information about features and restrictions that apply when using traditional UNIX commands in local-language environments:

- `rlogin`

When using the `rlogin` command to log on a Digital UNIX system from an ULTRIX system, be sure to specify the `-8` flag to pass 8-bit data without stripping. Otherwise, you will have problems entering non-ASCII characters from your terminal.

If you view a large data file while logged on the remote system, use a pager command, such as `pg`, and not the Hold Screen key to view a large data file. The `-8` option sets the terminal mode of the original host to RAW, disabling flow control. So, if data is sent to the terminal a rate faster than the terminal can handle it, some data is lost when you use the Hold Screen key.

This `rlogin` restriction applies not only when logging in from an ULTRIX system, but when logging in from any UNIX system whose software does not fully support 8-bit data format.

- Emacs editor

The operating system includes the multilingual Emacs software from the Free Software Foundation. Before using this editor, you must add the `/usr/i18n/mule/bin` directory to your process-specific search path. You can then invoke this editor by using the `mule` command.

- `vi` and `more`

The `vi` and `more` commands discard text that follows an invalid multibyte character. If you encounter this problem, it is likely that your locale setting is not correct for the text being viewed or edited. In this

case, reset your locale to one that matches the text and invoke the command again.

When used with Thai characters, `vi` may wrap lines before the right boundary of the screen. This happens because Thai text includes nonspacing characters, which contribute to the character count but not to display width. The editor wraps lines based on character count. For example, `vi` may wrap a line after entry of 80 characters, even though these characters do not occupy 80 columns on the screen.

- Using local-language user names and file names

It is a limitation of UNIX file systems that you cannot use a multibyte character whose second or subsequent byte is an ASCII slash (/) in names of files, users, or other objects. For portability reasons, Digital recommends that you avoid using multibyte characters in these names.

6.18 Using Language Support Enhancements for DECwindows Motif Applications

In the DECwindows Motif environment, you use versions of DECwindows Motif fonts, codesets, servers, and applications that support features discussed in earlier sections of this chapter. This section provides more detail on using DECwindows Motif with Asian languages. Topics include:

- Tuning the cache and unit size of the X Display Server for languages with ideographic characters
- Using font renderers for multibyte PostScript fonts
- Changing the language of the Start Session (login) and Pause windows
- Setting fonts in the Motif Window Manager for local and remote display
- Customizing a DECterm window for local languages
- Using the CDA viewer and converters with Asian-language text files

6.18.1 Tuning the X Server for Ideographic Languages

Asian languages have large ideographic character sets, so all characters needed for display are not loaded into memory at the same time. Instead, only as many characters as will fit in the memory cache are simultaneously loaded. When characters needed for display are not currently cached in memory, the least recently used font glyphs are removed from the cache to make room. The font-cache mechanism allows you to display ideographic

text in multiple typefaces, font sizes, and font styles without increasing the amount of memory that systems must have to support ideographic languages.

The X Server font-cache mechanism allows you to change the number of cache units and the size of these units to best accommodate the character sets used in displays. You will probably need to change the default values set for cache parameters to achieve the best performance from your system if it will display Asian-language text. Consider the following criteria when deciding on the optimal values for font caching:

- The number of ideographic languages that you want to display

If you intend to work with several ideographic languages during the same DECwindows Motif session, you need larger values for acceptable performance.

- The number of fonts that will be used simultaneously

Variation in font number and size depends partly on the kinds of applications you run. A desktop publishing application typically requires more fonts than other types of applications whereas a software development tool requires fewer.

- The number of frequently used characters in the languages you want to display

In Asian languages, only a subset of characters are used frequently. The size of this subset varies from one language to another. For example, around 20,000 standard characters are supported for Taiwan but only 5,000 of those characters are used frequently. Estimates for the number of frequently used characters for other Asian countries is as follows: People's Republic of China (3000), Korea (2000), and Japan (2000). Font-cache parameters are tuned to accommodate the subset of characters that are used frequently.

To change the cache size (which is the number of cache units) and the size of each cache unit, you must modify the X Server configuration file `/usr/lib/X11/xdm/Xservers`. This file contains a line, similar to the following one, that starts the X Server:

```
:0 local /usr/bin/X11/X
```

You can modify this line to add definitions for cache size and unit size. For example:

```
:0 local /usr/bin/X11/X -cs cache_size -cu unit_size
```

Table 6-22 describes the options that tune the font-cache mechanism.

Table 6-22: X Server Options for Tuning the Font-Cache Mechanism

stty Option	Description
<code>-cs <i>cache_size</i></code>	Defines the number of cache units. The minimum (and also default) value for this parameter is 1024. If you specify a cache size smaller than 1024, font caching is disabled. For one ideographic language, the recommended value is the lowest multiple of 1024 that accommodates the number of frequently used characters in that language. If a workstation displays multiple ideographic languages simultaneously, you have to add together the values required for each language. Specify an even larger value if you intend to run applications, such as desktop publishing software, that require multiple font styles and sizes for each ideographic character.
<code>-cu <i>unit_size</i></code>	Defines the size of each cache unit. The minimum value for unit size is 31 bytes and the default value is 128 bytes. If you specify a value smaller than 31 bytes, the value has no effect. If a particular font requires more memory space than 128 bytes, the font-cache mechanism automatically allocates one or more additional units to store its glyphs.

Note

Font caching applies only to uncompressed fonts in `pcf` format. Font caching is not applied to any compressed fonts or to fonts in `bdf` format. Because font caching cannot be used with compressed fonts, the 2-byte fonts for Asian languages are not installed in compressed format.

You can calculate cache unit size with the following formula:

```
unit_size =  
((floor(ceil((double)WIDTH / 8.0) / 4.0)) + 1.0) * 4.0 * (double)HEIGHT
```

Consider the following calculation for a typical font size of 24x24:

```
unit_size in bytes  
= ((floor(ceil((double) 24 / 8.0 / 4.0)) + 1.0) * 4.0 * (double) 24  
= 96
```

For 34x34 fonts, the unit size calculation would yield 272 bytes.

Given that 96 bytes are needed to cache a 24x24 font glyph and 272 bytes is needed to cache a 34x34 font glyph, the default unit size of 128 has the following implications:

- For 24x24 fonts, each character needs only one cache unit. If cache size is set at 4096, the cache can accommodate 4096 characters.
- For 34x34 fonts, each character needs three cache units. If cache size is set at 4096, the cache can accommodate 1365 characters.

Small fonts (whose characters require a single, 128-byte unit) are used more frequently for displaying ideographic characters. Therefore, you usually have to change only the cache size to achieve acceptable performance in text displays of languages with ideographic characters.

6.18.2 Using Font Renderers for Multibyte PostScript Fonts

The operating system 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 fonts
- UDC fonts

6.18.2.1 Setting Up the Font Renderer for Double-Byte PostScript Fonts

The font renderer for Chinese and Korean PostScript fonts can be set up 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.

6.18.2.2 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.

The UDC font renderer can be set up 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. The following table shows the XLFD entry that corresponds to different Asian codesets.

Table 6–23: 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.udec -system-decwin-normal-r--24-240-75-75-m-24-DEC.CNS11643.1986-UDC
zh_TW.big5.udec -system-decwin-normal-r--24-240-75-75-m-24-BIG5-UDC
```

6.18.3 Changing the Language of the Start Session Window

The language of the window used to resume your session when it is in pause state is determined by the current language setting for your session. However, you must set the language of the Start Session window where you log in to your workstation by modifying the X Display Manager configuration file `/usr/var/X11/xdm/xdm-config`. In this file, define the entry for the `DisplayManager*language` resource to be a locale for the language you want. The following example sets this resource to a locale for Japanese:

```
DisplayManager*language: ja_JP.sdeckanji
```

6.18.4 Setting Fonts for Display of Local Languages

The system where you install language variant subsets is automatically updated with fonts required for text display in the supported languages. Usually, the new fonts are also added to the font list in the systemwide resource file `/usr/lib/X11/app-defaults/Mwm` that the local Motif Window Manager uses. This automatic update procedure is sufficient, except when:

- A language-specific version of the systemwide `Mwm` resource file is installed as part of the local-language support
- The system where language variant subsets are installed is a client in a client-server display environment

The following sections explain how to work around the preceding restrictions for the DECwindows Motif environment.

6.18.4.1 Using MwmFontSetup to Update a Private Mwm File

Currently, the subsets that support Japanese and Hebrew install a language-specific version of the systemwide `Mwm` resource file. Therefore, if you need access to Japanese or Hebrew fonts along with access to fonts that support other languages, you cannot rely on the systemwide `Mwm` file and must update the font list in your private `Mwm` file. You can run the `/usr/i18n/usr/bin/X11/MwmFontSetup` script to add or remove language-specific fonts from the font list in `$HOME/Mwm`.

The `MwmFontSetup` script:

1. Creates a backup copy of your current `Mwm` file
2. Displays the fonts that are listed in the current file
3. Asks if you want to remove or add fonts
4. If you choose to add fonts, displays a list of languages for which support is installed and asks you to select the language whose fonts should be added to your `Mwm` file
5. Allows you to repeat the steps for adding or removing fonts until you are satisfied with the font list

At this point, you can select the EXIT option to exit from the procedure.

Note that the `MwmFontSetup` is useful only in the DECwindows Motif environment. In the Common Desktop Environment, applications access fonts through alias names that are mapped to the real names of the fonts. Font alias files must exist for each supported locale. For example, the font alias files for Japanese Extended UNIX Code are `/usr/dt/config/xfonts/ja_JP.eucJP/75dpi/fonts.alias`, `/usr/dt/config/xfonts/ja_JP.eucJP/100dpi/fonts.alias`, and `/usr/dt/config/ja_JP.eucJP/sys.font`. These alias files are installed when Digital UNIX software for Japanese language support is installed.

6.18.4.2 Accessing Local Language Fonts for Remote Displays

The information in this section is appropriate for the DECwindows Motif environment. In the Common Desktop Environment, fonts are mapped to generic alias names.

The system where Asian-language subsets are installed may function as a client in a client-server display environment. In this case, the local-language fonts must also be available to the Motif Window Managers for all the server systems where native language text is displayed. You need to install fonts for other locales either on individual systems used for remote login to the system where language variant subsets are installed or make the fonts known to the other systems through a font server. Table 6–24, Table 6–25, Table 6–27, Table 6–28, Table 6–29, and Table 6–30 describe the fonts used to display text in various local languages. You can use the `/usr/bin/X11/xlsfonts` command to determine which fonts are currently installed on a system.

Table 6–24: Bitmap Fonts for Asian Locales

Language	Typeface	Style	Sizes	75dpi	100dpi
Japanese	Gothic (ISO Latin-1)	Normal	8, 10, 12, 14, 18, 24	x	x
	Gothic (Kanji)	Normal	8, 10, 12, 14, 18, 24	x	x
	Gothic (Roman Kana)	Normal	8, 10, 12, 14, 18, 24	x	x
	kmenu (ISO Latin-1)	Normal	12	x	x
	kmenu (Roman Kana)	Normal	12	x	x
	Mincho (ISO Latin-1)	Normal	8, 10, 12, 14, 18, 24	x	x
	Mincho (Kanji)	Normal	8, 10, 12, 14, 18, 24	x	x
	Mincho (Roman Kana)	Normal	8, 10, 12, 14, 18, 24	x	x
	Screen (DECsuppl)	Normal	14, 18, 24	x	
	Screen (DECTech)	Normal	14, 18, 24	x	
	Screen (ISO Latin-1)	Normal	14, 18, 24	x	
	Screen (Kanji00)	Normal	10, 14, 16, 18, 24	x	
	Screen (Kanji11)	Normal	10, 14, 18, 24	x	
	Screen (Roman Kana)	Normal	10, 14, 18, 24	x	
Korean	Gotic	Normal	16, 24	x	

Table 6–24: Bitmap Fonts for Asian Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
	Myungcho	Normal	16, 24, 32	x	
	Screen	Normal	18, 24	x	
	KS Roman	Normal	18, 24	x	
Simplified Chinese	FangSongTi	Normal	24, 34	x	
	HeiTi	Normal	16, 24, 34	x	
	KaiTi	Normal	24, 34	x	
	Screen	Normal	18, 24	x	
	SongTi	Normal	16, 24, 34	x	
Traditional Chinese	Hei (CNS11643)	Normal	16, 24	x	
	Hei (DTSCS)	Normal	16, 24	x	
	Screen (CNS11643)	Normal	18, 24	x	
	Screen (DTSCS)	Normal	18, 24	x	
	Sung (CNS11643)	Normal	24, 32	x	
	Sung (DTSCS)	Normal	24, 32	x	
Thai	Screen	Normal	14, 18, 24	x	
Asia (Misc.)	Screen (DEC Ctrl)	Normal	14, 18, 24	x	
	Screen (DRCS)	Normal	18, 24	x	

Table 6–25: Bitmap Fonts for *.ISO8859-2 Locales

Language	Typeface	Style	Sizes	75dpi	100dpi
Czech, Hungarian, Polish, Slovak, Slovene	Arial	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x

Table 6–25: Bitmap Fonts for *.ISO8859-2 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Arial Narrow	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Book Antiqua	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Bookman Old Style	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Century Gothic	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x

Table 6–25: Bitmap Fonts for *.ISO8859-2 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Century Schoolbook	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Courier	Normal	8, 10, 12, 14, 18, 24, 36	x	x
		Italic	8, 10, 12, 14, 18, 24, 36	x	x
		Bold	8, 10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24, 36	x	x
	Monotype Corsiva	Normal	10, 12, 14, 18, 24, 36	x	x
	Times New Roman	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Terminal	Normal	14, 18	x	x
		Double-Width	14, 18	x	x

Table 6–25: Bitmap Fonts for *.ISO8859-2 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Double-Width, Double-Height	28, 36	x	x
		Narrow	14, 18	x	x
		Double-Width, Narrow	14, 18	x	x
		Double-Width, Double-Height, Narrow	28, 36	x	x
		Bold	14, 18	x	x
		Double-Width, Bold	14, 18	x	x
		Double-Width, Double-Height, Bold	28, 36	x	x
		Narrow, Bold	14, 18	x	x
		Double-Width, Narrow, Bold	14, 18	x	x
		Double-Width, Double-Height, Narrow, Bold	28, 36	x	x

Table 6–26: Bitmap Fonts for *.ISO8859-4 Locales

Language	Typeface	Style	Sizes	75dpi	100dpi
Lithuanian	Arial	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x

Table 6–26: Bitmap Fonts for *.ISO8859-4 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
	Arial Narrow	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Book Antiqua	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Bookman Old Style	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Century Gothic	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Century Schoolbook	Normal	10, 12, 14, 18, 24, 36	x	x

Table 6–26: Bitmap Fonts for *.ISO8859-4 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Courier	Normal	8, 10, 12, 14, 18, 24, 36	x	x
		Italic	8, 10, 12, 14, 18, 24, 36	x	x
		Bold	8, 10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24, 36	x	x
	Monotype Corsiva	Normal	10, 12, 14, 18, 24, 36	x	x
	Times New Roman	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Terminal	Normal	14, 18	x	x
		Double-Width	14, 18	x	x
		Double-Width, Double-Height	28, 36	x	x
		Narrow	14, 18	x	x

Table 6–26: Bitmap Fonts for *.ISO8859-4 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Double-Width, Narrow	14, 18	x	x
		Double-Width, Double- Height, Narrow	28, 36	x	x
		Bold	14, 18	x	x
		Double-Width, Bold	14, 18	x	x
		Double-Width, Double- Height, Bold	28, 36	x	x
		Narrow, Bold	14, 18	x	x
		Double-Width, Narrow, Bold	14, 18	x	x
		Double-Width, Double- Height, Narrow, Bold	28, 36	x	x

Table 6–27: Bitmap Fonts for *.ISO8859-5 Locales

Language	Typeface	Style	Sizes	75dpi	100dpi
Russian	Arial	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Courier	Normal	8, 10, 12, 14, 18, 24, 36	x	x

Table 6–27: Bitmap Fonts for *.ISO8859-5 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Italic	8, 10, 12, 14, 18, 24, 36	x	x
		Bold	8, 10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24, 36	x	x
	Nimrod	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Plantin	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Times New Roman	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Terminal	Normal	14, 18	x	x
		Double-Width	14, 18	x	x

Table 6–27: Bitmap Fonts for *.ISO8859-5 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Double-Width, Double-Height	28, 36	x	x
		Narrow	14, 18	x	x
		Double-Width, Narrow	14, 18	x	x
		Double-Width, Double-Height, Narrow	28, 36	x	x
		Bold	14, 18	x	x
		Double-Width, Bold	14, 18	x	x
		Double-Width, Double-Height, Bold	28, 36	x	x
		Narrow, Bold	14, 18	x	x
		Double-Width, Narrow, Bold	14, 18	x	x
		Double-Width, Double-Height, Narrow, Bold	28, 36	x	x

Table 6–28: Bitmap Fonts for *.ISO8859-7 Locales

Language	Typeface	Style	Sizes	75dpi	100dpi
Greek	Arial	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x

Table 6–28: Bitmap Fonts for *.ISO8859-7 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
	Courier	Normal	8, 10, 12, 14, 18, 24, 36	x	x
		Italic	8, 10, 12, 14, 18, 24, 36	x	x
		Bold	8, 10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24, 36	x	x
	Times New Roman	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Terminal	Normal	14, 18	x	x
		Double-Width	14, 18	x	x
		Double-Width, Double-Height	28, 36	x	x
		Narrow	14, 18	x	x
		Double-Width, Narrow	14, 18	x	x
		Double-Width, Double-Height, Narrow	28, 36	x	x
		Bold	14, 18	x	x
		Double-Width, Bold	14, 18	x	x
		Double-Width, Double-Height, Bold	28, 36	x	x

Table 6–28: Bitmap Fonts for *.ISO8859-7 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Narrow, Bold	14, 18	x	x
		Double-Width, Narrow, Bold	14, 18	x	x
		Double-Width, Double- Height, Narrow, Bold	28, 36	x	x

Table 6–29: Bitmap Fonts for *.ISO8859-8 Locales

Language	Typeface	Style	Sizes	75dpi	100dpi
Hebrew	David	Normal	8, 10, 12, 14, 18, 24	x	x
		Italic	8, 10, 12, 14, 18, 24	x	x
		Bold	8, 10, 12, 14, 18, 24	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24	x	x
	Frankruhl	Normal	8, 10, 12, 14, 18, 24	x	x
		Italic	8, 10, 12, 14, 18, 24	x	x
		Bold	8, 10, 12, 14, 18, 24	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24	x	x
	Gam	Normal	8, 10, 12, 14, 18, 24	x	x
		Italic	8, 10, 12, 14, 18, 24	x	x

Table 6–29: Bitmap Fonts for *.ISO8859-8 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Bold	8, 10, 12, 14, 18, 24	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24	x	x
	menu	Normal	10, 12	x	x
	Miriam	Normal	8, 10, 12, 14, 18, 24	x	x
		Italic	8, 10, 12, 14, 18, 24	x	x
		Bold	8, 10, 12, 14, 18, 24	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24	x	x
	Miriam Fixed	Normal	8, 10, 12, 14, 18, 24	x	x
		Italic	8, 10, 12, 14, 18, 24	x	x
		Bold	8, 10, 12, 14, 18, 24	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24	x	x
	Narkiss Tam	Normal	8, 10, 12, 14, 18, 24	x	x
		Italic	8, 10, 12, 14, 18, 24	x	x
		Bold	8, 10, 12, 14, 18, 24	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24	x	x
	Terminal	Normal	14, 18	x	x
		Double-Width	14, 18	x	x
		Double-Width, Double-Height	28, 36	x	x

Table 6–29: Bitmap Fonts for *.ISO8859-8 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Narrow	14, 18	x	x
		Double-Width, Narrow	14, 18	x	x
		Double-Width, Double-Height, Narrow	28, 36	x	x
		Bold	14, 18	x	x
		Double-Width, Bold	14, 18	x	x
		Double-Width, Double-Height, Bold	28, 36	x	x
		Narrow, Bold	14, 18	x	x
		Double-Width, Narrow, Bold	14, 18	x	x
		Double-Width, Double-Height, Narrow, Bold	28, 36	x	x

Table 6–30: Bitmap Fonts for *.ISO8859-9 Locales

Language	Typeface	Style	Sizes	75dpi	100dpi
Turkish	Arial	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Arial Narrow	Normal	10, 12, 14, 18, 24, 36	x	x

Table 6–30: Bitmap Fonts for *.ISO8859-9 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Book Antiqua	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Bookman Old Style	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Century Gothic	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Century Schoolbook	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x

Table 6–30: Bitmap Fonts for *.ISO8859-9 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Courier	Normal	8, 10, 12, 14, 18, 24, 36	x	x
		Italic	8, 10, 12, 14, 18, 24, 36	x	x
		Bold	8, 10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	8, 10, 12, 14, 18, 24, 36	x	x
	Monotype Corsiva	Normal	10, 12, 14, 18, 24, 36	x	x
	Times New Roman	Normal	10, 12, 14, 18, 24, 36	x	x
		Italic	10, 12, 14, 18, 24, 36	x	x
		Bold	10, 12, 14, 18, 24, 36	x	x
		Bold-Italic	10, 12, 14, 18, 24, 36	x	x
	Terminal	Normal	14, 18	x	x
		Double-Width	14, 18	x	x
		Double-Width, Double-Height	28, 36	x	x
		Narrow	14, 18	x	x
		Double-Width, Narrow	14, 18	x	x
		Double-Width, Double-Height, Narrow	28, 36	x	x

Table 6–30: Bitmap Fonts for *.ISO8859-9 Locales (cont.)

Language	Typeface	Style	Sizes	75dpi	100dpi
		Bold	14, 18	x	x
		Double-Width, Bold	14, 18	x	x
		Double-Width, Double-Height, Bold	28, 36	x	x
		Narrow, Bold	14, 18	x	x
		Double-Width, Narrow, Bold	14, 18	x	x
		Double-Width, Double-Height, Narrow, Bold	28, 36	x	x

In the DECwindows Motif environment, fonts used to display local-language text in window titles, menu bars, menus, and so forth must also be added to one or more of the Motif Window Manager resource files on the server systems. These resource files are:

- Systemwide

Color monitor: `/usr/lib/X11/app-defaults/Mwm`

Black-and-white monitor: `/usr/lib/X11/app-defaults/Mwm_bw`

Gray-scale monitor: `/usr/lib/X11/app-defaults/Mwm_gray`

- Private

Color monitor: `$HOME/Mwm`

Black-and-white monitor: `$HOME/Mwm_bw`

Gray-scale monitor: `$HOME/Mwm_gray`

If users remotely log on to their home systems, where language variant subsets are installed and where they have run the `MwmFontSetup` script, their private `Mwm` resource files may already list the fonts moved to the display systems. Refer to Section 6.18.4.1 for information about the `MwmFontSetup` script.

6.18.5 Customizing the DECterm Window for Local Languages

The following features and restrictions apply to DECterm windows that you create when an Asian language is specified for the language setting:

- You cannot customize the National Replacement Character Set (NRCS) when a DECterm window is emulating a terminal type that supports ideographic character sets.
- Depending on the language setting, additional menu items, push buttons, toggle switches, and text entry fields may be available to you for customizing DECterm features.
- Additional terminal identifiers may be available for terminal emulation. Note that terminal emulation always follows the selected language. For example, you cannot have a DECterm window emulate a Japanese terminal when the user interface language for the DECterm window is set to English.
- From the DECterm Window Options dialog box, you can select the following font sizes for ideographic character sets:
 - Big Font: 24-point font
 - Little Font: 18-point font
 - Fine Font: 14-point font

Fine Font is available for Japanese and Thai only.

German Standard Font is not supported when a DECterm window emulates a terminal that supports ideographic characters. For other ISO Latin character sets, the size options are the same as those offered for standard DECterm software.

- By default, the DECterm application saves its options in a file named `DXterm_%l_%t`, where `%l` is replaced by the language and `%t` is replaced by the territory (country) of the current locale.

Resource files should not be shared among locales. Therefore, do not save any application's resource file in one locale and attempt to use the same resource file when invoking the application in another locale.

- For a language supported by an input method server, you must be sure the input language server is connected to the DECterm window where you input characters in that language. Otherwise, you cannot use the

input method for character entry. The connection between a DECterm window and an input server does not exist if:

- The DECterm window was started before the input server started
- The input method server was killed for some reason

For example, an input method server is killed if it was being run on a remote system that shut down.

If the connection between a DECterm window and the input method server was broken, you can first try to reconnect to the server by selecting the Reset Terminal item from the window's Commands menu. Alternatively, you can start the input method server and then create another DECterm window where you can use the input method.

For information about terminal programming enhancements that applications can use to draw ruled lines on a DECterm window, see Section 4.2.

6.18.6 Using the CDA Viewer and Converters with Asian Language Text

The CDA viewer is a DECwindows Motif application that lets you display the contents of compound documents and graphics, image files, and text files that contain ideographic characters. The viewer also supports PostScript files; however, PostScript display is supported only for languages with single-byte characters.

The viewer works with converters that convert files from one data format to another. If you want to view or convert text files that contain Asian-language characters, you must specify an option file to the CDA viewer and converters. This file must contain an entry to identify the codeset that applies to the text file being viewed, converted, or both. An option entry for text files starts with the keyword `text`. For Asian-language text files, this entry line must specify the appropriate character encoding (`text_encoding`). The following example is appropriate for a Japanese text file encoded in DEC Kanji:

```
text text_encoding dec_kanji
```

By convention, option files use the file extension `.cda_options`, so an appropriate name for the options file with the preceding entry might be `japanese.cda_options`.

The following table lists the supported encodings for text files used with CDA viewers and converters.

Language	Codeset	text_encoding Keyword
Chinese (Simplified)	DEC Hanzi	dec_hanzi
Chinese (Traditional)	DEC Hanyu	dec_hanyu
Japanese	DEC Kanji	dec_kanji
Korean	DEC Hangul (Korean)	dec_hangul

You specify an options file to CDA commands with the `-O` flag. The following example shows how to invoke the CDA viewer for the DECwindows Motif environment to display a Japanese text file named `ja_document.txt`:

```
% dxvdoc -f text -O japanese.cda_options \
ja_document.txt
```

The following example shows how to invoke a CDA converter to convert the same Japanese text file to `ddif` format:

```
% cdoc -s text -O japanese.cda_options \
-o ja_document.ddif ja_document.txt
```

After the text file is converted to `ddif` format, you can convert the `ddif` file to a PostScript file, as follows:

```
% cdoc -d ps -o ja_document.ps ja_document.ddif
```

Your system should have the required fonts installed if you are using the CDA converter to produce a PostScript file with Asian-language characters. The following table lists the basic fonts that the CDA converter uses for different Asian languages.

Language	Basic Font
Korean	Munjo
Hanyu	Sung-Light-CNS11643 or Sung-Light-DTSCS
Hanzi	XiSong-GB2312-80
Japanese	Ryumin-Light-EUC-H or Ryumin-Light.Hankaku

If the preceding fonts do not exist on the system, the converter uses Courier font.

As an alternative to the option file mechanism for specifying the encoding of input text files, you can define the environment variables `DDIF_READ_TEXT_GL` and `DDIF_READ_TEXT_GR`. The following table lists the supported values and associated encoding for these variables:

<code>DDIF_READ_TEXT_GL</code>	<code>DDIF_READ_TEXT_GR</code>	Encoding
LATIN1	MCS	MCS
LATIN1	LATIN1	ISO Latin-1
LATIN1	KATAKANA	ASCII-Kana
LATIN1	KANJI	DEC Kanji
ROMAN	MCS	Roman-MCS
ROMAN	LATIN1	Roman
ROMAN	KANJI	Roman-Kanji
ROMAN	KATAKANA	Roman-Kana
LATIN1	HANZI	DEC Hanzi
LATIN1	HANGUL	DEC Hangul (Korean)
LATIN1	HANYU	DEC Hanyu

Note

The CDA converter does not support vertical writing. Therefore, vertical text prints horizontally in files produced by the converter.

For complete information about CDA viewers and converters, refer to the `cda(4)` reference page. The `cda(4)` reference page also lists additional reference pages that describe specific CDA commands. Only a few of those commands and their options have been described here.

7

Creating Locales

This chapter explains how to develop a locale, which provides information appropriate for a particular combination of language, territory, and codeset. You use the `localedef` command to create locales from the following files:

- A character map source file (`charmap`)

The `charmap(4)` reference page explains 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 that shows part of a `charmap` file for the SJIS codeset, which defines both single-byte and multibyte characters.

- A locale source file

The `locale(4)` reference page explains the rules and format for this file. This chapter develops a locale named `de_DE.ISO8859-1@example` that supports the language and customs of Germany.

- A methods file with associated shareable library

These files are required when the `charmap` file defines multibyte characters; otherwise, the files are optional. The methods file specifies the shareable library that contains redefinitions of the C Library interfaces that convert data to and from internal process (wide-character) encoding.

7.1 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 7-1 shows a fragment of the source file, `ISO8859-1.cmap`, used for `the_de_DE.ISO8859-1@example` locale being developed in this chapter. Appendix B contains this file in its entirety.

Example 7-1: The charmap File for a Sample Locale

```
# Map file providing symbols for characters whose binary
# encodings are specified in the ISO Latin-1 codeset.
<code_set_name> "ISO8859-1"
<mb_cur_max>1
<mb_cur_min> 1
<escape_char> \
<comment_char> #

CHARMAP
<NU>      \d000
<SH>      \d001
<SX>      \d002
<EX>      \d003
<ET>      \d004
<EQ>      \d005
<AK>      \d006
<BL>      \d007
<BS>      \d008
.
.
<0>      \d048
<1>      \d049
<2>      \d050
<3>      \d051
.
.
<A>      \d065
<B>      \d066
<C>      \d067
<D>      \d068
<E>      \d069
.
.
<X>      \d088
<Y>      \d089
<Z>      \d090
<<(>    \d091
</>     \d092
<)\>>   \d093
<'>>    \d094
<_>     \d095
<'!>    \d096
<a>      \d097
<b>      \d098
<c>      \d099
<d>      \d100
<e>      \d101
.
.
<x>\d120
<y>      \d121
<z>      \d122
<(!>    \d123
<!>     \d124
<!)>    \d125
<'?>    \d126
<DT>    \d127
```


Example 7–1: The charmap File for a Sample Locale (cont.)

```

:
<O:>    \d214    [4]
<U:>    \d220
:
<ss>    \d223    [4]
:
<o:>    \d246    [4]
:
<u:>    \d252    [4]
:
<backspace>    \d008    [5]
<tab>          \d009
<newline>      \d010
<vertical-tab> \d011
<form-feed>    \d012
<carriage-return> \d013
:
<space>        \d032    [5]
<exclamation-mark> \d033
<quotation-mark>  \d063
<number-sign>    \d035
<dollar-sign>    \d036
END CHARMAP    [6]
```

[1] Comment line

By default, the comment character is the number sign (#). You can override this default with a `<comment_char>` definition (see 2).

[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 `<comment_char>` and `<escape_char>` specify the default values for the comment character and escape character, respectively. The value for `<mb_cur_max>`, the maximum length (in bytes) of a character, is 1 for this particular locale. The value for `<mb_cur_min>`, the minimum length (in bytes) of a character, must be 1 in all locales. (All locales include characters in the Portable Character Set, which defines single-byte characters.)

The `<code_set_name>` value will be 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
`\dnnn` or `\dnn` , where *n* is a decimal digit
- hexadecimal
`\xnn`, where *n* is a hexadecimal digit
- octal
`\nnn` or `\nn`, where *n* is an octal digit

- 5 Additional maps for characters

You can create multiple symbolic names for the same character (encoding). In this source file, for example, the backspace character (value `\d008`) has two symbolic names, `<BS>` and `<backspace>`. When more than one symbolic name exists for a character, you can specify any of them in locale definition source files to refer to the character.

- 6 Trailer marking end of character maps

The source files for codesets with multibyte characters have more complex character maps. Example 7-2 shows 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 7-2: 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
#
# 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
```

Example 7–2: Fragment from a charmap File for a Multibyte Codeset (cont.)

❶ Codeset name

❷ Minimum number of bytes per character

This value must be 1.

❸ Maximum number of bytes per character

In SJIS, the largest multibyte character is 2 bytes in length.

❹ Symbols and encodings for ASCII characters

❺ 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.

❻ Maps for user-defined characters within the SJIS codeset

These maps establish ranges of encodings for which users can later define characters.

❼ Maps for the single-byte characters of the Hankaku-Kana character set

Refer to the `charmap(4)` reference page 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 shown in this example are not necessarily the names being proposed for adoption by any standards group.

7.2 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. Example 7-3 shows the structure of a locale definition source file in pseudocode. The sections for locale categories are discussed in more detail following the example.

Example 7-3: 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, by default the backslash (`\`), is used in decimal, hexadecimal, and octal constants and 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 the definition itself.

6 Trailer for locale category section

Section trailers start with the keyword `END`, 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 the same as defined for the POSIX, or C, locale.

7.2.1 Defining the LC_CTYPE Locale Category

The LC_CTYPE section defines character classes and character attributes used in operations such as case conversion. Example 7-4 shows the definition for this section.

Example 7-4: LC_CTYPE Category Definition

```
LC_CTYPE 1
upper <A>;<A:>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;<N>;<O>;\
<O:>;<P>;<Q>;<R>;<S>;<T>;<U>;<U:>;<V>;<W>;<X>;<Y>;<Z> 2
lower <a>;<a:>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;<n>;<o>;\
<o:>;<p>;<q>;<r>;<s>;<ss>;<t>;<u>;<u:>;<v>;<w>;<x>;<y>;<z> 2
alpha <A>;<A:>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;<N>;<O>;\
<O:>;<P>;<Q>;<R>;<S>;<T>;<U>;<U:>;<V>;<W>;<X>;<Y>;<Z>;<a>;<a:>;<b>;\
<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;<n>;<o>;<o:>;<p>;<q>;<r>;\
<s>;<ss>;<t>;<u>;<u:>;<v>;<w>;<x>;<y>;<z> 2
space <tab>;<newline>;<vertical-tab>;<form-feed>;<carriage-return>;<space>;\
<NS> 2
cntrl <NUL>;...;<IS1>;<DEL>;...;<AC> 2
:
:
toupper (<a>,<A>);(<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>);(<o:>,<O:>);(<p>,<P>);\
(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);(<u>,<U>);(<u:>,<U:>);\
(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);(<z>,<Z>) 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, 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.

As shown in the definition of the `cntrl` class, you can specify a horizontal ellipsis (...) to represent a range of characters. In the string `<NUL>;...;<IS1>`, for example, the ellipsis represents all characters whose encodings are between the character whose symbol is `<NUL>` and

the character whose symbol is `<IS1>`. The symbols and their encodings are specified in the `charmap` file for the locale.

The standard character classes are represented by the following keywords:

- `upper` (uppercase letter characters)
- `lower` (lowercase letter characters)
- `alpha` (all letter characters)
- `space` (white-space characters)
- `cntrl` (control characters)
- `punct` (punctuation characters)
- `digit` (numeric digits)
- `xdigit` (hexadecimal digits)
- `blank` (blank characters)
- `graph` (printable characters, excluding the space character)
- `print` (printable characters, including the space character)

From the application standpoint, there is also the class `alnum`. This class is not defined in a locale; it is by definition a combination of characters in the `alpha` and `digit` classes.

3 Definitions of case conversion for letter characters

These definitions, which begin with the keywords `toupper` and `tolower`, list symbols in pairs rather than individually. In the `toupper` definition shown 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 perform case conversion on text data.

4 Section trailer

The preceding example does not completely illustrate all the options you can use when defining the `LC_CTYPE` category. You can:

- 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

Character classification is language specific. Therefore, the standard character classes may not apply to all languages. Define for a locale 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 the class-specific functions `iswalpha`, `iswpunct`, and so forth to test the standard character classes.

Refer to the `locale(4)` reference page for additional rules and restrictions that apply to the `LC_CTYPE` category definition.

7.2.2 Defining the LC_COLLATE Locale Category

The `LC_COLLATE` section specifies how characters and strings are collated. Example 7-5 shows part of an `LC_COLLATE` section.

Example 7-5: LC_COLLATE Category Definition

```
LC_COLLATE      1
order_start    forward;forward;backward      2
:
:          <o>      <o>;<o>;<o>      3
:
:          <o:>     <o>;<o>;<o:>     3
:
:          <O>     <O>;<O>;<O>     3
:
:          <O:>     <O>;<O>;<O:>     3
:
:
```

Example 7–5: LC_COLLATE Category Definition (cont.)

```
    <Z>      <z>;<Z>;<Z>  [3]
:
    UNDEFINED      IGNORE; IGNORE; IGNORE  [4]
order_end  [5]
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 order. 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, ensures that nonignored characters that are the shortest distance from the start (`forward,position`) or end (`backward,position`) of the string collate first)

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 Digital 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, 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. For example, the lowercase character `o`, lowercase character `o` umlaut, uppercase character `O`, and uppercase character `O` umlaut, whose symbols are `<o>`, `<o: >`, `<O>`, and `<O: >`, respectively, are grouped together (have the same weight) at the first sort level. At the secondary sort level, lowercase `o` is grouped with lowercase `o` umlaut and uppercase `O` is grouped with uppercase `O` umlaut. The four characters have distinct weights at the tertiary sort level.

4 Collation order statement for undefined characters

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. This statement

indicates that such characters are to be ignored during collation for all weight comparisons.

You should include a collation order statement that begins with the `UNDEFINED` keyword. 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.

- 5] Trailer to indicate the end of collation order statements
- 6] Trailer to indicate the end of the `LC_COLLATE` section

The preceding example shows only a few of the options that you can specify when defining the `LC_COLLATE` category. You can also use:

- 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 symbols <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
:
```

Refer to the `locale(4)` reference page for more detailed information on the `LC_COLLATE` category definition.

7.2.3 Defining the `LC_MESSAGES` Locale Category

The `LC_MESSAGES` section defines strings that are valid for affirmative and negative responses from users. Example 7-6 shows an `LC_MESSAGES` section.

Example 7–6: LC_MESSAGES Category Definition

```
LC_MESSAGES      1
yesexpr          2  "^[<j><J>][[:alpha:]]*"
noexpr           3  "^[<n><N>][[:alpha:]]*"
yesstr           4  "<j>"
nostr            5  "<n>"
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.

In German, an affirmative responses is “ja.” The expression specified for `yesexpr` defines a valid response as being `j` or `J` or a string that begins with `j` or `J` and is followed by any number of letter characters. Note that 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.

In German, “nein” is the negative response. The definition of `noexpr` is similar to the one for `yesexpr`, except that the only or initial character of the user’s response must be the letter `n` or `N`.

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 string that is delimited by double quotation marks.

The `yesstr` entry is marked for removal from the X/Open UNIX standard; however, some applications and systems software might still use `yesstr` rather than `yesexpr`. To ensure that your locale works correctly with such software, it is a good idea to define `yesstr` in your locale.

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 string that is delimited by double quotation marks.

The `nostr` entry is marked for removal from the X/Open UNIX standard; however, some applications and systems software might still use `nostr` rather than `noexpr`. To ensure that your locale works correctly with such software, it is a good idea to define `nostr` in your locale.

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 `LC_MESSAGES`. The `copy` statement represents a complete definition of the category and cannot be used along with explicit symbol definitions.

7.2.4 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 7-7 shows an `LC_MONETARY` section.

Example 7-7: `LC_MONETARY` Category Definition

```
LC_MONETARY      1
int_curr_symbol  " <D><M> " 2
currency_symbol  " <D><M> " 2
mon_decimal_point "<,>"    2
mon_thousands_sep "<.>"    2
mon_grouping     3          2
positive_sign    " "        2
negative_sign    "<->"      2
:
END LC_MONETARY  3
```

1 Section header

2 Symbol definitions

The entries in the example specify the following:

- The international and local currency symbols are the string `DM` (for Deutsch Mark).
- The decimal point is the comma `,`.
- The separator grouping digits to the left of the decimal point is the period `.`.
- The number of digits in groups separated by periods is 3.
- The positive sign is null.
- The negative sign is the minus `-` character.

3 Section trailer

The following list describes all 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

- `positive_sign`

The string indicating that a monetary value is nonnegative

- `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 along with explicit symbol definitions.

Refer to the `locale(4)` reference page for complete information about specifying `LC_MONETARY` symbol definitions.

7.2.5 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 7–8 shows this section.

Example 7–8: LC_NUMERIC Category Definition

```
LC_NUMERIC      1
decimal_point   "<,>" 2
thousands_sep  "<.>" 3
grouping       3      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
- 4 The size of each group of digits to the left of the radix character
- 5 Category trailer

The preceding example shows 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.

Refer to the `locale(4)` reference page for detailed rules about symbol definitions.

7.2.6 Defining the LC_TIME Locale Category

The `LC_TIME` section defines the interpretation of field descriptors supported by the `date` command. This category section also affects the behavior of the `strftime()`, `wcsftime()`, `strptime()`, and `nl_langinfo()` functions. Example 7–9 shows some of the symbols defined for the sample German locale.

Example 7–9: LC_TIME Category Definition

```
LC_TIME      1

abday        "<S><o>" ; "<M><o>" ; "<D><i>" ; "<M><i>" ; "<D><o>" ; \
             "<F><r>" ; "<S><a>"      2

day          "<S><o><n><n><t><a><g>" ; "<M><o><n><t><a><g>" ; \
             "<D><i><e><n><s><t><a><g>" ; \
             "<M><i><t><t><w><o><c><h>" ; \
             "<D><o><n><n><e><r><s><t><a><g>" ; \
             "<F><r><e><i><t><a><g>" ; "<S><a><m><s><t><a><g>"      3

abmon        "<J><a><n>" ; "<F><e><b>" ; "<M><a><:><r>" ; \
             "<A><p><r>" ; "<M><a><i>" ; "<J><u><n>" ; \
             "<J><u><l>" ; "<A><u><g>" ; "<S><e><p>" ; \
             "<O><k><t>" ; "<N><o><v>" ; "<D><e><z>"      4

mon          "<J><a><n><u><a><r>" ; "<F><e><b><r><u><a><r>" ; \
             "<M><a><:><r><z>" ; "<A><p><r><i><l>" ; "<M><a><i>" ; \
             "<J><u><n><i>" ; "<J><u><l><i>" ; \
             "<A><u><g><u><s><t>" ; \
             "<S><e><p><t><e><m><b><e><r>" ; \
             "<O><k><t><o><b><e><r>" ; \
             "<N><o><v><e><m><b><e><r>" ; \
             "<D><e><z><e><m><b><e><r>"      5

d_t_fmt      "%d.%B %Y %H:%M:%S"      6
:
:
END LC_TIME  7
```

1 Section header

2 Abbreviated names for days of the week

Use the %a conversion specifier to include this string in formats.

3 Full names for days of the week

Use the %A conversion specifier to include this string in formats.

4 Abbreviated names for months of the year

Use the `%b` conversion specifier to include this string in formats.

- 5 Full names for months of the year

Use the `%B` conversion specifier to include this string in formats.

- 6 Format for combined date and time information

Use this format to combine field descriptors (whose first character is the percent sign (`%`)) and symbols for characters. You can specify characters from the Portable Character Set (PCS), such as the period (`.`) and ASCII space, explicitly as characters rather than implicitly through symbols; however, use symbols to specify all other characters.

The specified format includes the field descriptors for the day of the month (`%d`), the full name of the month (`%B`), the full representation of the year (`%Y`), the number of hours in a 24-hour period (`%H`), the number of minutes (`%M`), and the number of seconds (`%S`). If the date were December 12, 1993, and the time 29 seconds after 12 o'clock in the afternoon, the format specified in this example would cause the `date` command to display `12.Dezember 1993 12:00:29`.

- 7 Section trailer

The preceding example includes only some of the symbol definitions that are standard for the `LC_TIME` category. The following definitions are also standard:

- `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 English would be:

```
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 (an Asian date construct) in the locale

- 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 (used in Asian locales); corresponds to the %O field descriptor

As is true for other category sections, you can specify a `copy` statement to include all `LC_TIME` definitions from another locale. Note that Digital UNIX supports symbols and field descriptors in addition to those described here. Refer to the `locale(4)` reference page for more complete information.

7.3 Building Libraries to Convert Multibyte/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**.

Only locales with multibyte codesets must use methods. When a locale uses methods, there are some methods that the locale must supply and other methods that it can optionally supply. A method is required when the corresponding interface is converting characters between data formats and needs codeset-specific logic to do that operation correctly. A method is optional when the corresponding interface is working with data after it has

been converted to wide-character format and can apply logic that is valid for both single-byte and multibyte characters.

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, along with 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.

7.3.1 Required Methods

If your locale uses methods, it must supply the following methods; without these methods, it is impossible for C Library functions to convert data between multibyte and wide-character formats:

- `__mbstopcs`
- `__mbtopc`
- `__pcstombs`
- `__pctomb`
- `mblen`
- `mbstowcs`
- `mbtowc`
- `wcstombs`
- `wctomb`
- `wcswidth`
- `wcwidth`

7.3.1.1 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 7.3.1.6) 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 7–10 shows the file `__mbstopcs_sdeckanji.c` that defines the `__mbstopcs` method used with the `ja_JP.sdeckanji` locale.

Example 7–10: 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_ctype_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 = __mbstopc_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
    } 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 per 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 Initialize 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; breaks out of the loop and sets `endptr` to the first byte of the character that could not be converted if the `mbtopc` method fails to convert a character and returns an error.

The `err` variable contains the return status 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.

7.3.1.2 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 7.3.1.7) 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 7-11 shows the file `__mbtopc_sdeckanji.c` that defines the `__mbtopc` method used with the `ja_JP.sdeckanji` locale.

Example 7–11: 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 {

```

Example 7–11: The `__mbtopc_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```
        *err = 2;
        return(0);
    }
} 14
else if (s[0] == 0x8f) {
    if (maxlen >= 3) {
        if ((s[1] >=0xal && s[1] <=0xfe) &&
            (s[2] >=0xal && s[2] <= 0xfe)) {
            *pwc = (wchar_t) (((s[1] - 0xal) << 7) |
                (wchar_t) (s[2] - 0xal)) + 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] >= 0xal && s[0] <= 0xfe) {
    if (maxlen >= 2) {
        if (s[1] >=0xal && s[1] <= 0xfe) {
            *pwc = (wchar_t) (((s[0] - 0xal) << 7) |
                (wchar_t) (s[1] - 0xal)) + 0x15e;
            return(2);
        }
        else if (s[1] >=0x21 && s[1] <= 0x7e) {
            *pwc = (wchar_t) (((s[0] - 0xal) << 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 `int`, to a large signed data type, such as `char`. Sign extension means that 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 `0xffff8e`. In this case, a condition like the following one would be evaluated as true when you would 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; otherwise, an application could produce a segmentation fault by referring to this pointer when a wide character has not been stored in `pwc`.

- 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.

7.3.1.3 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 7-12 shows the file `__pcstombs_sdeckanji.c` that defines the `__pcstombs` method used with the `ja_JP.sdeckanji` locale.

Example 7-12: 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 shown 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 described in Section 7.3.1.3 performs. Because of the direction of the data conversion, the `__pcstombs` method:

- 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

7.3.1.4 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 7–13 shows the file `__pctomb_sdeckanji.c` that defines the `__pctomb` method used with the `ja_JP.sdeckanji` locale.

Example 7–13: 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.

7.3.1.5 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 7-14 shows the file `__mblen_sdeckanji.c` that defines the `mblen` method used with the `ja_JP.sdeckanji` locale.

Example 7-14: 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);
    } 8

    if (s[0] <= 0x8d)
        return(1); 9

    else if (s[0] == 0x8e) {
```

Example 7–14: The `__mbrlen_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```
        if (maxlen >= 2 && s[1] >=0xa1 && s[1] <=0xfe)
    } 10      return(2);

    else if (s[0] == 0x8f) {
        if(maxlen >=3 && (s[1] >=0xa1 && s[1] <=0xfe) &&
            (s[2] >=0xa1 && s[2] <= 0xfe))
    } 11      return(3);

    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) )
    } 13      return(2);

    _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 `mbrlen()` 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 `int`, to a large signed data type, such as `char`. Sign extension means that 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 `0xffff8e`. In this case, a condition like the following one would be evaluated as true when you would 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.

7.3.1.6 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 7-15 shows the file `__mbstowcs_sdeckanji.c` that defines the `mbstowcs` method used with the `ja_JP.sdeckanji` locale.

Example 7-15: 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);
```

Example 7–15: The `__mbstowcs_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```
        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
}
```

- 1** Include 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 locale'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.

The programmer 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()`. The programmer 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 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.

7.3.1.7 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 7-16 shows the file `__mbtowc_sdeckanji.c` that defines the `mbtowc` method used with the `ja_JP.sdeckanji` locale.

Example 7-16: 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] |
+-----+-----+-----+-----+
| 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
}
```

Example 7–16: The `__mbtowc_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```
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);
} 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 `int`, to a large signed data type, such as `char`. Sign extension means that 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 `0xffff8e`. In this case, a condition like the following one 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.

7.3.1.8 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 7-17 shows the file `__wcstombs_sdeckanji.c` that defines the `wcstombs` method used with the `ja_JP.sdeckanji` locale.

Example 7-17: 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

    while (1) { 12
```

Example 7–17: The `__wctombs_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```
if ((len = __wctomb_sdeckanji(tmp, *pwcs)) == -1)
    return(-1); 13

else if (cnt+len > n) {
    *s = '\0';
    break; 14
}

if (tmp[0] == '\0') {
    *s = '\0';
    break; 15
}

for (i=0; i<len; i++) {
    *s = tmp[i];
    s++;
    16
}

cnt += len; 17

if (cnt == n)
    break; 18

    pwcs++; 19
} 20

if (cnt == 0)
    cnt = len; 21
return (cnt); 22
}
```

- 1** Include 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 number of 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.

The programmer 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()`. The programmer 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.

7.3.1.9 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 7–18 shows the file `__wctomb_sdeckanji.c` that defines the `wctomb` method for the `ja_JP.sdeckanji` locale.

Example 7–18: 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	--	--	--

Example 7–18: The `__wctomb_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```

| 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;
        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 Include 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 and, 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 satisfies none of the preceding conditions.

7.3.1.10 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 7–19 shows the file `__wcswidth_sdeckanji.c` that defines the `wcswidth` method used for the `ja_JP.sdeckanji` locale.

Example 7–19: 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

+-----+-----+-----+-----+
| 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
```

Example 7–19: The `__wcswidth_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```
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 Include header files that contain constants and structures required for this method.

2 Describes the algorithm used to determine the required display width.

Note that 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.

7.3.1.11 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 7–20 shows the file `__wctype_sdeckanji.c` that defines the `wctype` method used with the `ja_JP.sdeckanji` locale.

Example 7–20: The `__wctype_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 __wctype_sdeckanji(
    wint_t wc, 3
    _LC_ctype_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))
```

Example 7–20: The `__wctype_sdeckanji` Method for the `ja_JP.sdeckanji` Locale (cont.)

```
return(2); 10  
return(-1); 11  
}
```

- 1** Include header files that contain constants and structures required for this method.
- 2** Describes the algorithm used to determine the required display width.

Note that 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.

7.3.2 Optional Methods

A locale can include methods in addition to those discussed in Section 7.3.1. If your locale uses methods but does not supply any 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. The following list names the optional methods:

- **LC_CTYPE category**
 - `towupper`
 - `towlower`
 - `wctype`
 - `iswctype`
- **LC_COLLATE category**
 - `fnmatch`
 - `strcoll`
 - `strxfrm`
 - `wscoll`
 - `wcsxfrm`
 - `regcomp`
 - `regexec`
 - `regfree`
 - `regerror`
- **LC_MONETARY, LC_NUMERIC, or both categories**
 - `localeconv`
 - `strfmon`
- **LC_TIME category**
 - `strftime`
 - `strptime`
 - `wcsftime`
- **LC_MESSAGES**
 - `rpmatch`

- Miscellaneous use

- nl_langinfo

Writing optional methods requires detailed information about the internal interfaces to C library routines. This information is proprietary to Digital and may be subject to change. In the rare cases where your locale must include an optional method, contact your Digital technical support representative to request information.

7.3.3 Building a Shareable Library to Use with a Locale

Example 7–21 shows 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 7–21: 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 \
  __wctype_sdeckanji.c

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 \
  __wctype_sdeckanji.o \
  -lc
```

Refer to the `cc(1)` and `ld(1)` reference pages for more information about the `cc` and `ld` commands and how you build shared libraries.

7.3.4 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 7-22 shows the section of a `methods` file for the methods used with the `ja_JP.sdeckanji` locale. Because there is a mandatory list of methods that you must define if you want to override any C library interfaces, your `methods` file must always specify an entry for each of the required methods as shown in this example. The `ja_JP.sdeckanji` locale relies on default implementations for all optional methods, so Example 7-22 does not contain entries for any of the optional methods.

Example 7-22: 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
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. Note that 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.

- ❑ Header to mark start of method entries
- ❑ Entries for required methods
- ❑ Trailer to mark end of method entries

Refer to the `localedef(1)` reference page for detailed information about `methods` file entries.

7.4 Building and Testing the Locale

Use the `localedef` command to build a locale from its source files. Example 7–23 shows the command line needed to build the German 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 7–23: Building the `de_DE.ISO8859-1@example` Locale

```
% localedef -f ISO8859-1.cmap \      ❶  
-i de_DE.ISO8859-1.lscr \          ❷  
de_DE.ISO8859-1@example           ❸
```

- ❶ The `-f` option specifies the character map source file.
- ❷ The `-i` option specifies the locale definition source file.
- ❸ 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, you should 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 directory `/usr/lib/nls/loc` where other locales reside.

Example 7–23 shows only one form and a few options for the `localedef` command. The `localedef(1)` reference page is 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 7–22.
- 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 7–24 uses the `date` command to test the date/time format.

Example 7–24: Setting the `LOCPATH` Variable and Testing a Locale

```
% setenv LOCPATH ~/harry/locales
% setenv LANG de_DE.ISO8859-1@example
% date
12.Dezember 1993 09:18:11
```

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 shows 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 de_DE.ISO8859-1
% setenv LC_CTYPE de_DE.ISO8859-1@example
% setenv LC_COLLATE de_DE.ISO8859-1@example
:
% setenv LC_TIME de_DE.ISO8859-1@example
```


A

Summary Tables of Worldwide Portability Interfaces

This appendix lists and summarizes worldwide portability interfaces (WPI) that are defined by current X/Open CAE specifications to support the wide-character data type. Tables in this appendix also list the ISO C equivalent, if any, for each WPI interface. The 1994 amendment to the ISO C standard specifies additional interfaces that are not included in Issue 4, Version 2 of the X/Open CAE specifications. These additional interfaces are also included in this appendix. The reference pages (manpages) provide detailed information for each interface. Refer to the `standards(5)` reference page for information about compiling a program to include the appropriate definition environments for interfaces in different standards.

A.1 Locale Announcement

Programs call the following function to use the appropriate locale (language, territory, and codeset) at run time:

WPI Function	Description
<code>setlocale()</code>	Establishes localization data at run time.

A.2 Character Classification

Character classification functions classify wide-character values according to the codeset defined in the locale category `LC_CTYPE`.

WPI Function	Equivalent in ISO C	Description
<code>iswalnum()</code>	<code>isalnum()</code>	Tests if a character is alphanumeric.
<code>iswalpha()</code>	<code>isalpha()</code>	Tests if a character is alphabetic.
<code>iswcntrl()</code>	<code>iscntrl()</code>	Tests if a character is a control character.

WPI Function	Equivalent in ISO C	Description
<code>iswdigit()</code>	<code>isdigit()</code>	Tests if a character is a decimal digit in the portable character set.
<code>iswgraph()</code>	<code>isgraph()</code>	Tests if a character is a graphic character.
<code>iswlower()</code>	<code>islower()</code>	Tests if a character is lowercase.
<code>iswprint()</code>	<code>isprint()</code>	Tests if a character is a printing character.
<code>iswpunct()</code>	<code>ispunct()</code>	Tests if a character is a punctuation mark.
<code>iswspace()</code>	<code>isspace()</code>	Tests if a character determines white space in displayed text.
<code>iswupper()</code>	<code>isupper()</code>	Tests if a character is uppercase.
<code>iswxdigit()</code>	<code>isxdigit()</code>	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 two more functions that provide a common interface to all the classification categories:

- `wctype()`
Returns a value that corresponds to a character classification.
- `iswctype()`
Tests if a wide character has a certain property.

The 11 WPI functions listed in the preceding table can therefore be replaced by calls to the `wctype()` and `iswctype()` functions as shown in the following table:

Call Using Classification Function	Equivalent Call Using <code>wctype()</code> and <code>iswctype()</code>
<code>iswalnum(<i>wc</i>)</code>	<code>iswctype(<i>wc</i>, wctype("alnum"))</code>
<code>iswalpha(<i>wc</i>)</code>	<code>iswctype(<i>wc</i>, wctype("alpha"))</code>
<code>iswcntrl(<i>wc</i>)</code>	<code>iswctype(<i>wc</i>, wctype("cntrl"))</code>
<code>iswdigit(<i>wc</i>)</code>	<code>iswctype(<i>wc</i>, wctype("digit"))</code>
<code>iswgraph(<i>wc</i>)</code>	<code>iswctype(<i>wc</i>, wctype("graph"))</code>
<code>iswlower(<i>wc</i>)</code>	<code>iswctype(<i>wc</i>, wctype("lower"))</code>

Call Using Classification Function	Equivalent Call Using <code>wctype()</code> and <code>iswctype()</code>
<code>iswprint(wc)</code>	<code>iswctype(wc , wctype("print"))</code>
<code>iswpunct(wc)</code>	<code>iswctype(wc , wctype("punct"))</code>
<code>iswspace(wc)</code>	<code>iswctype(wc , wctype("space"))</code>
<code>iswupper(wc)</code>	<code>iswctype(wc , wctype("upper"))</code>
<code>iswxdigit(wc)</code>	<code>iswctype(wc , wctype("xdigit"))</code>

In this table, the quoted literals in the call to `wctype` are the character classes commonly defined in locales for Western European and many Eastern European languages; however, a locale can define other character classes. For example, a locale for an Asian language might define additional character classes to distinguish ideographic from phonetic characters. The `wctype()` and `iswctype()` functions let you check for any character class defined in a locale, not just for classes that have associated classification functions, such as `iswalnum()`.

The 1994 amendment to the ISO C standard adds the following functions:

- `wctrans()`
Maps a wide character to a property defined in the current locale
- `towctrans()`
Converts a wide character according to a property defined in the current locale

A.3 Case Conversion

The following case conversion functions let you switch the case of a wide character according to the codeset defined in the locale category `LC_CTYPE`:

WPI Function	Equivalent in ISO C	Description
<code>towlower()</code>	<code>towlower()</code>	Converts a character to lowercase.
<code>towupper()</code>	<code>towupper()</code>	Converts a character to uppercase.

A.4 Character Collation

The following WPI function sorts wide-character strings according to rules specified in the locale defined for the `LC_COLLATE` category:

WPI Function	Equivalent in ISO C	Description
<code>wscoll()</code>	<code>strcoll()</code>	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 following WPI functions allow programs to retrieve, according to locale setting, data that is language specific or country specific:

WPI Function	Description
<code>nl_langinfo()</code>	Is a general-purpose function that retrieves language and cultural data according to the locale setting.
<code>strfmon()</code>	Formats a monetary value according to the locale setting.
<code>localeconv()</code>	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
<code>strftime()</code>	Formats a date and time string based on the specified format string and according to the locale setting.
<code>wcsftime()</code>	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.
<code>strptime()</code>	Converts a character string to a time value according to a specified format string; reverses the operation performed by <code>strftime()</code> .

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
<code>fprintf()</code>	Prints formatted output to a file by using a <code>vararg</code> parameter list.
<code>fscanf()</code>	Converts formatted input from a file.
<code>printf()</code>	Prints formatted output to the standard output stream by using a <code>vararg</code> parameter list.
<code>scanf()</code>	Converts formatted input from the standard input stream.
<code>sprintf()</code>	Formats one or more values and writes the output to a character string by using a <code>vararg</code> parameter list.
<code>sscanf()</code>	Converts formatted data from a character string.
<code>vfprintf()</code>	Prints formatted output to a file by using a <code>stdarg</code> parameter list.

WPI/ISO C Function	Description
<code>vprintf()</code>	Prints formatted output to the standard output stream by using a <code>stdarg</code> parameter list.
<code>vsprintf()</code>	Formats a <code>stdarg</code> parameter list and writes the output to a character string.

The WPI enhancements to the preceding functions include:

1. `%digit$` conversion specifier

This specifier allows variation in the ordinal position of the argument being printed; such variation is frequently necessary when text is translated into different languages.

2. Use of the decimal-point character as specified by the locale

This feature affects `e`, `E`, `f`, `g`, and `G` conversions.

3. Use of the thousands-grouping character specified by the locale

4. The `C` and `S` conversion characters

These conversion characters let you convert wide characters and wide-character strings, respectively.

The 1994 amendment to the ISO C standard adds the following functions:

- `fwprintf()`

Prints formatted wide characters to the specified output stream by using a `vararg` parameter list

- `fwsscanf()`

Converts formatted wide characters from the specified output stream

- `swprintf()`

Prints formatted wide characters to the specified address by using a `vararg` parameter list

- `swscanf()`

Converts formatted wide characters from the specified address

- `vfwprintf()`

Prints formatted wide characters to the specified output stream by using a `stdarg` parameter list

- `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

- `wscanf()`

Converts formatted wide characters from the standard input

A.8 Number Conversion

The WPI adds the following functions to convert wide-character strings to various numeric formats:

WPI Function	Equivalent in ISO C	Description
<code>wctod()</code>	<code>strtod()</code>	Converts the initial portion of a wide-character string to a double-precision floating-point number.
<code>wctol()</code>	<code>strtoul()</code>	Converts the initial portion of a wide-character string to a long integer number.
<code>wctoul()</code>	<code>strtoul()</code>	Converts the initial portion of a wide-character 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 various functions to convert between multibyte data and wide-character data.

WPI Function	Description
<code>mblen()</code>	<p>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, <code>cp</code>:</p> <pre>cp++;</pre> <p>The following example incorporates the <code>mblen()</code> function to ensure language-independent operation at run time; the <code>MB_CUR_MAX</code> variable is defined by the locale to be the maximum number of bytes that any character can occupy:</p> <pre>cp += mblen(cp, MB_CUR_MAX);</pre>
<code>mbstowcs()</code>	Converts a multibyte-character string to a wide-character string.
<code>mbtowlc()</code>	Converts a multibyte character to a wide character.
<code>wcstombs()</code>	Converts a wide-character string to a multibyte character string.
<code>wctomb()</code>	Converts a wide character to a multibyte character.

Note

You do not always need to explicitly handle the conversion to and from file code (multibyte data) to internal process code (wide-character 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 ISO C input/output functions (discussed in Section A.10) also perform multibyte/wide-character conversions automatically.

The 1994 amendment to the ISO C standard adds the following functions:

- `btowc()`

Converts a single byte from multibyte-character format to wide-character format

- `wctob()`

Converts a wide character to a single byte in multibyte-character format, if possible

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	Equivalent in ISO C	Description
<code>fgetwc()</code>	<code>fgetc()</code>	Gets a character from the input stream and converts it to a wide character.
<code>fgetws()</code>	<code>fgets()</code>	Gets a character string from the input stream and converts it to a wide-character string.
<code>fputwc()</code>	<code>fputc()</code>	Converts a wide character to a multibyte character and writes the result to an output stream.
<code>fputws()</code>	<code>fputs()</code>	Converts a wide-character string to a multibyte character string and writes the result to an output stream.
<code>getwc()</code>	<code>getc()</code>	Gets a character from the input stream, which is passed to the function as an argument, and converts it to a wide character.
<code>getwchar()</code>	<code>getchar()</code>	Gets a character from the standard input stream and converts it to a wide character.
None	<code>gets()</code>	Use <code>fgetws()</code> .
<code>putwc()</code>	<code>putc()</code>	Converts a wide character to a multibyte character and writes the result to an output stream, which is passed to the function as an argument.

WPI Function	Equivalent in ISO C	Description
<code>putwchar()</code>	<code>getchar()</code>	Converts a wide character to a multibyte character and writes the result to the standard output stream.
None	<code>puts()</code>	Use <code>fputws()</code> .
<code>ungetwc()</code>	<code>ungetc()</code>	Pushes a wide character back onto the input stream.

A.11 String Handling

The WPI defines alternatives and additions to ISO C string-handling functions to support manipulation of wide-character strings. The WPI functions support both single-byte and multibyte characters.

String Concatenation:

WPI Function	Equivalent in ISO C	Description
<code>wscat()</code>	<code>strcat()</code>	Appends a copy of a string to the end of another string.
<code>wcsncat()</code>	<code>strncat()</code>	Is similar to <code>wscat()</code> except that the number of characters to be appended is limited by the parameter <i>n</i> .

String Searching:

WPI Function	Equivalent in ISO C	Description
<code>wcschr()</code>	<code>strchr()</code>	Locates the first occurrence of a wide character in a wide-character string.
<code>wcsrchr()</code>	<code>strrchr()</code>	Locates the last occurrence of a wide character in a wide-character string.
<code>wcspbrk()</code>	<code>strpbrk()</code>	Locates the first occurrence of any wide characters from one wide-character string in another wide-character string.

WPI Function	Equivalent in ISO C	Description
<code>wcswcs()</code>	<code>strchr()</code>	Locates the first occurrence of one wide-character string in another wide-character string.
<code>wcscspn()</code>	<code>strcspn()</code>	Returns the number of initial elements of one wide-character string that are all wide characters not included in the second wide-character string.
<code>wcsspn()</code>	<code>strspn()</code>	Returns the number of initial elements of one wide-character string that are all characters included in the second wide-character string.

The 1994 amendment to the ISO C standards adds the following function:

- `wcsstr()`

Finds a wide-character substring

String Copying:

WPI Function	Equivalent in ISO C	Description
<code>wscpy()</code>	<code>strcpy()</code>	Copies a wide-character string.
<code>wcncpy()</code>	<code>strncpy()</code>	Is similar to <code>wscpy()</code> except that the number of wide characters to be copied is limited by the parameter <i>n</i> .

String Comparison:

WPI Function	Equivalent in ISO C	Description
<code>wscmp()</code>	<code>strcmp()</code>	Compares two wide-character strings.
<code>wcncmp()</code>	<code>strncmp()</code>	Is similar to <code>wscmp()</code> except that the number of wide characters to be compared is limited by a parameter <i>n</i> .

String Length Determination:

WPI Function	Equivalent in ISO C	Description
<code>wcslen()</code>	<code>strlen()</code>	Determines the number of wide characters in a wide-character string.

String Decomposition:

WPI Function	Equivalent in ISO C	Description
<code>wcstok()</code>	<code>strtok()</code>	Decomposes a wide-character string into a series of tokens, each delimited by a wide character from another wide-character string.

Printing Position Determination:

WPI Function	Equivalent in ISO C	Description
<code>wcswidth()</code>	None	Determines the number of printing positions required for a number of wide characters in a wide-character string.
<code>wcwidth()</code>	None	Determines the number of printing positions required for a wide character.

Performing Memory Operations on Wide-Character Strings:

The 1994 amendment to the ISO C standard adds the following functions:

- `wmemcpy()`
Copies wide characters from one buffer to another
- `wmemchr()`
Searches a buffer for the specified wide character
- `wmemcmp()`
Compares the specified number of wide characters in two buffers
- `wmemmove()`

Copies wide characters from one buffer to another in a nondestructive manner

- `wmemset()`

Copies the specified wide 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. You specify for these interfaces 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 library `libiconv.a`.

WPI Function	Equivalent in ISO C	Description
<code>iconv_open()</code>	None	Initializes a conversion stream by identifying the source and the target codesets.
<code>iconv_close()</code>	None	Closes the conversion stream.
<code>iconv()</code>	None	Converts an input string encoded in the source codeset to an output string encoded in the target codeset.

Refer to Section 6.16 for a description of the `iconv` command and the types of conversions that are supported.

A.13 Miscellaneous Functions

The 1994 amendment to the ISO C standard adds the following functions:

- `fwide()`

Sets stream orientation to byte or wide character

- `mbsinit()`

Determines whether a multibyte string is in the initial conversion state

B

Sample Locale Source Files

This appendix contains complete source files for the sample locale discussed in Chapter 7.

B.1 Character Map (charmap) Source File

```
# Map file providing symbols for characters whose binary
# encodings are specified in the ISO Latin-1 codeset.
```

```
CHARMAP
<NU>    \d000
<SH>    \d001
<SX>    \d002
<EX>    \d003
<ET>    \d004
<EQ>    \d005
<AK>    \d006
<BL>    \d007
<BS>    \d008
<HT>    \d009
<LF>    \d010
<VT>    \d011
<FF>    \d012
<CR>    \d013
<SO>    \d014
<SI>    \d015
<DL>    \d016
<D1>    \d017
<D2>    \d018
<D3>    \d019
<D4>    \d020
<NK>    \d021
<SY>    \d022
<EB>    \d023
<CN>    \d024
<EM>    \d025
<SB>    \d026
<EC>    \d027
```

<FS>	\d028
<GS>	\d029
<RS>	\d030
<US>	\d031
<SP>	\d032
<!>	\d033
<">	\d034
<Nb>	\d035
<DO>	\d036
<%>	\d037
<&>	\d038
<'>	\d039
<(>	\d040
<)>	\d041
<*>	\d042
<+>	\d043
<,>	\d044
<->	\d045
<.>	\d046
</>	\d047
<0>	\d048
<1>	\d049
<2>	\d050
<3>	\d051
<4>	\d052
<5>	\d053
<6>	\d054
<7>	\d055
<8>	\d056
<9>	\d057
<:>	\d058
<:>	\d059
<<>	\d060
<=>	\d061
<\>>	\d062
<?>	\d063
<At>	\d064
<A>	\d065
	\d066
<C>	\d067
<D>	\d068
<E>	\d069
<F>	\d070
<G>	\d071
<H>	\d072
<I>	\d073

<J>	\d074
<K>	\d075
<L>	\d076
<M>	\d077
<N>	\d078
<O>	\d079
<P>	\d080
<Q>	\d081
<R>	\d082
<S>	\d083
<T>	\d084
<U>	\d085
<V>	\d086
<W>	\d087
<X>	\d088
<Y>	\d089
<Z>	\d090
<<(>	\d091
</>	\d092
<)\>>	\d093
<'>>	\d094
<_>	\d095
<'!>	\d096
<a>	\d097
	\d098
<c>	\d099
<d>	\d100
<e>	\d101
<f>	\d102
<g>	\d103
<h>	\d104
<i>	\d105
<j>	\d106
<k>	\d107
<l>	\d108
<m>	\d109
<n>	\d110
<o>	\d111
<p>	\d112
<q>	\d113
<r>	\d114
<s>	\d115
<t>	\d116
<u>	\d117
<v>	\d118
<w>	\d119

<x>\d120
<y> \d121
<z> \d122
<(!> \d123
<!!> \d124
<!)> \d125
<'?> \d126
<DT> \d127
<PA> \d128
<HO> \d129
<BH> \d130
<NH> \d131
<IN> \d132
<NL> \d133
<SA> \d134
<ES> \d135
<HS> \d136
<HJ> \d137
<VS> \d138
<PD> \d139
<PU> \d140
<RI> \d141
<S2> \d142
<S3> \d143
<DC> \d144
<P1> \d145
<P2> \d146
<TS> \d147
<CC> \d148
<MW> \d149
<SG> \d150
<EG> \d151
<SS> \d152
<GC> \d153
<SC> \d154
<CI> \d155
<ST> \d156
<OC> \d157
<PM> \d158
<AC> \d159
<NS> \d160
<!I> \d161
<Ct> \d162
<Pd> \d163
<Cu> \d164
<Ye> \d165

<BB>	\d166
<SE>	\d167
<' :>	\d168
<Co>	\d169
<-a>	\d170
<<<>	\d171
<NO>	\d172
<-->	\d173
<Rg>	\d174
<' ->	\d175
<DG>	\d176
<+>	\d177
<2S>	\d178
<3S>	\d179
<' '>	\d180
<My>	\d181
<PI>	\d182
<.M>	\d183
<' ,>	\d184
<1S>	\d185
<-o>	\d186
<\>\>>	\d187
<14>	\d188
<12>	\d189
<34>	\d190
<?I>	\d191
<A!>	\d192
<A'>	\d193
<A\>>	\d194
<A?>	\d195
<A:>	\d196
<AA>	\d197
<AE>	\d198
<C ,>	\d199
<E!>	\d200
<E'>	\d201
<E\>>	\d202
<E:>	\d203
<I!>	\d204
<I'>	\d205
<I\>>	\d206
<I:>	\d207
<D->	\d208
<N?>	\d209
<O!>	\d210
<O'>	\d211

<O\>>	\d212	
<O?>	\d213	
<O:>	\d214	
<*X>	\d215	
<O/>	\d216	
<U!>	\d217	
<U'>	\d218	
<U\>>	\d219	
<U:>	\d220	
<Y'>	\d221	
<TH>	\d222	
<ss>	\d223	
<a!>	\d224	
<a'>	\d225	
<a\>>	\d226	
<a?>	\d227	
<a:>	\d228	
<aa>	\d229	
<ae>	\d230	
<c,>	\d231	
<e!>	\d232	
<e'>	\d233	
<e\>>	\d234	
<e:>	\d235	
<i!>	\d236	
<i'>	\d237	
<i\>>	\d238	
<i:>	\d239	
<d->	\d240	
<n?>	\d241	
<o!>	\d242	
<o'>	\d243	
<o\>>	\d244	
<o?>	\d245	
<o:>	\d246	
<-:>	\d247	
<o/>	\d248	
<u!>	\d249	
<u'>	\d250	
<u\>>	\d251	
<u:>	\d252	
<y'>	\d253	
<th>	\d254	
<y:>	\d255	
<NUL>		\d000
<SOH>		\d001

<STX>	\d002
<ETX>	\d003
<EOT>	\d004
<ENQ>	\d005
<ACK>	\d006
<alert>	\d007
<BEL>	\d007
<backspace>	\d008
<tab>	\d009
<newline>	\d010
<vertical-tab>	\d011
<form-feed>	\d012
<carriage-return>	\d013
<DLE>	\d016
<DC1>	\d017
<DC2>	\d018
<DC3>	\d019
<DC4>	\d020
<NAK>	\d021
<SYN>	\d022
<ETB>	\d023
<CAN>	\d024
<SUB>	\d026
<ESC>	\d027
<IS4>	\d028
<IS3>	\d029
<intro>	\d029
<IS2>	\d030
<IS1>	\d031
	\d127
<space>	\d032
<exclamation-mark>	\d033
<quotation-mark>	\d063
<number-sign>	\d035
<dollar-sign>	\d036
<percent-sign>	\d037
<ampersand>	\d038
<apostrophe>	\d039
<left-parenthesis>	\d040
<right-parenthesis>	\d041
<asterisk>	\d042
<plus-sign>	\d043
<comma>	\d044
<hyphen>	\d045
<period>	\d046
<full-stop>	\d046

```

<slash>                \d047
<solidus>              \d047
<zero>                 \d048
<one>                  \d049
<two>                  \d050
<three>                \d051
<four>                 \d052
<five>                 \d053
<six>\d054
<seven>                \d055
<eight>                \d056
<nine>                 \d057
<colon>                \d058
<semicolon>           \d059
<less-than-sign>      \d060
<equals-sign>         \d061
<greater-than-sign>   \d062
<question-mark>      \d063
<commercial-at>      \d064
<left-square-bracket> \d091
<reverse-solidus>     \d092
<right-square-bracket> \d093
<circumflex-accent>  \d094
<low-line>            \d095
<grave-accent>       \d096
<left-curly-bracket>  \d123
<vertical-line>       \d124
<right-curly-bracket> \d125
<tilde>               \d126
END CHARMAP

```

B.2 Locale Definition Source File

```

#
# de_DE_ISO8859-1.lsrc
#
# This is a locale definition source file for German in Germany.
# Character symbols in this file are defined in the charmap
# file ISO88591.cmap, which specifies character encodings
# according to the ISO Latin-1 codeset.
#
LC_CTYPE

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

lower  <a>;<a:;>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;<n>;<o>;\
<o:;>;<p>;<q>;<r>;<s>;<ss>;<t>;<u>;<u:;>;<v>;<w>;<x>;<y>;<z>

```

```

alpha <A>;<A:;>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;<N>;<O>;\
<O:;>;<P>;<Q>;<R>;<S>;<T>;<U>;<U:;>;<V>;<W>;<X>;<Y>;<Z>;<a>;<a:;>;<b>;\
<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;<n>;<o>;<o:;>;<p>;<q>;<r>;\
<s>;<ss>;<t>;<u>;<u:;>;<v>;<w>;<x>;<y>;<z>

space <tab>;<newline>;<vertical-tab>;<form-feed>;<carriage-return>;<space>;\
<NS>

cntrl <NUL>;...;<IS1>;<DEL>;...;<AC>

punct <exclamation-mark>;...;<slash>;<colon>;...;<commercial-at>;\
<left-square-bracket>;...;<grave-accent>;<left-curly-bracket>;...;\
<tilde>;<!I>;...;<?I>;<*X>;<-:;>

digit <0>;<1>;<2>;<3>;<4>;<5>;<6>;<7>;<8>;<9>

xdigit <0>;<1>;<2>;<3>;<4>;<5>;<6>;<7>;<8>;<9>;<A>;<B>;<C>;<D>;<E>;<F>;<a>;\
<b>;<c>;<d>;<e>;<f>

blank <space>;<tab>;<NS>

graph <A>;<A:;>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;<N>;<O>;\
<O:;>;<P>;<Q>;<R>;<S>;<T>;<U>;<U:;>;<V>;<W>;<X>;<Y>;<Z>;<a>;<a:;>;<b>;\
<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;<n>;<o>;<o:;>;<p>;<q>;<r>;\
<s>;<ss>;<t>;<u>;<u:;>;<v>;<w>;<x>;<y>;<z>;<0>;<1>;<2>;<3>;<4>;<5>;\
<6>;<7>;<8>;<9>;<exclamation-mark>;...;<slash>;<colon>;...;\
<commercial-at>;<left-square-bracket>;...;<grave-accent>;\
<left-curly-bracket>;...;<tilde>;<!I>;...;<?I>;<*X>;<-:;>

print <A>;<A:;>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;<N>;<O>;\
<O:;>;<P>;<Q>;<R>;<S>;<T>;<U>;<U:;>;<V>;<W>;<X>;<Y>;<Z>;<a>;<a:;>;<b>;\
<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;<n>;<o>;<o:;>;<p>;<q>;<r>;\
<s>;<ss>;<t>;<u>;<u:;>;<v>;<w>;<x>;<y>;<z>;<0>;<1>;<2>;<3>;<4>;<5>;\
<6>;<7>;<8>;<9>;<exclamation-mark>;...;<slash>;<colon>;...;\
<commercial-at>;<left-square-bracket>;...;<grave-accent>;\
<left-curly-bracket>;...;<tilde>;<!I>;...;<?I>;<*X>;<-:;>;<space>

toupper (<a>,<A>);(<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>);(<o:;>,<O:;>);(<p>,<P>);\
(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);(<u>,<U>);(<u:;>,<U:;>);\
(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);(<z>,<Z>)

tolower (<A>,<a>);(<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>);(<O:;>,<o:;>);(<P>,<p>);\
(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);(<U>,<u>);(<U:;>,<u:;>);\
(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);(<Z>,<z>)

END LC_CTYPE

#

LC_COLLATE

order_start forward;forward;backward
UNDEFINED IGNORE;IGNORE;IGNORE
<0> <0>;<0>;<0>
<1> <1>;<1>;<1>
<2> <2>;<2>;<2>

```

<3>	<3>i<3>i<3>
<4>	<4>i<4>i<4>
<5>	<5>i<5>i<5>
<6>	<6>i<6>i<6>
<7>	<7>i<7>i<7>
<8>	<8>i<8>i<8>
<9>	<9>i<9>i<9>
<a>	<a>i<a>i<a>
<a:>	<a>i<a>i<a:>
	ii
<c>	<c>i<c>i<c>
<d>	<d>i<d>i<d>
<e>	<e>i<e>i<e>
<f>	<f>i<f>i<f>
<g>	<g>i<g>i<g>
<h>	<h>i<h>i<h>
<i>	<i>i<i>i<i>
<j>	<j>i<j>i<j>
<k>	<k>i<k>i<k>
<l>	<l>i<l>i<l>
<m>	<m>i<m>i<m>
<n>	<n>i<n>i<n>
<o>	<o>i<o>i<o>
<o:>	<o>i<o>i<o:>
<p>	<p>i<p>i<p>
<q>	<q>i<q>i<q>
<r>	<r>i<r>i<r>
<s>	<s>i<s>i<s>
<ss>	<s><s>i<s><s>i<ss>
<t>	<t>i<t>i<t>
<u>	<u>i<u>i<u>
<u:>	<u>i<u>i<u:>
<v>	<v>i<v>i<v>
<w>	<w>i<w>i<w>
<x><x>i<x>i<x><y>	<y>i<y>i<y>
<z>	<z>i<z>i<z>
<A>	<a>i<A>i<A>
<A:>	<a>i<A>i<A:>
	ii
<C>	<c>i<C>i<C>
<D>	<d>i<D>i<D>
<E>	<e>i<E>i<E>
<F>	<f>i<F>i<F>
<G>	<g>i<G>i<G>
<H>	<h>i<H>i<H>
<I>	<i>i<I>i<I>
<J>	<j>i<J>i<J>
<K>	<k>i<K>i<K>
<L>	<l>i<L>i<L>
<M>	<m>i<M>i<M>
<N>	<n>i<N>i<N>
<O>	<o>i<O>i<O>
<O:>	<o>i<O>i<O:>
<P>	<p>i<P>i<P>
<Q>	<q>i<Q>i<Q>
<R>	<r>i<R>i<R>
<S>	<s>i<S>i<S>
<T>	<t>i<T>i<T>

```

        <U>      <u>i<U>i<U>
        <U:>    <u>i<U>i<U:>
        <V>      <v>i<V>i<V>
        <W>      <w>i<W>i<W>
        <X>      <x>i<X>i<X>
        <Y>      <y>i<Y>i<Y>
        <Z>      <z>i<Z>i<Z>
order_end

END LC_COLLATE

#

LC_MESSAGES

yesexpr      "[<j><J>][[:alpha:]]*"
noexpr      "[<n><N>][[:alpha:]]*"

END LC_MESSAGES

#

LC_MONETARY

int_curr_symbol      "<D><M>"
currency_symbol      "<D><M>"
mon_decimal_point    "<,>"
mon_thousands_sep    "<.>"
mon_grouping         3
positive_sign
negative_sign        "<->"
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      "<,>"
thousands_sep     "<.>"
grouping          3
END LC_NUMERIC

#
LC_TIME
abday              "<S><o>";" <M><o>";" <D><i>";" <M><i>";" <D><o>";\
                  "<F><r>";" <S><a>"
day                "<S><o><n><n><t><a><g>";" <M><o><n><t><a><g>";\
                  "<D><i><e><n><s><t><a><g>";\
                  "<M><i><t><t><w><o><c><h>";\
                  "<D><o><n><n><e><r><s><t><a><g>";\
                  "<F><r><e><i><t><a><g>";" <S><a><m><s><t><a><g>"
abmon              "<J><a><n>";" <F><e><b>";" <M><a><r>";\
                  "<A><p><r>";" <M><a><i>";" <J><u><n>";\
                  "<J><u><l>";" <A><u><g>";" <S><e><p>";\
                  "<O><k><t>";" <N><o><v>";" <D><e><z>"
mon                "<J><a><n><u><a><r>";" <F><e><b><r><u><a><r>";\
                  "<M><a><r><z>";" <A><p><r><i><l>";" <M><a><i>";\
                  "<J><u><n><i>";" <J><u><l><i>";\
                  "<A><u><g><u><s><t>";\
                  "<S><e><p><t><e><m><b><e><r>";\
                  "<O><k><t><o><b><e><r>";\
                  "<N><o><v><e><m><b><e><r>";\
                  "<D><e><z><e><m><b><e><r>"
d_t_fmt            "%d.%B %Y %H:%M:%S"
d_fmt              "%d.%m %y"
t_fmt              "%H:%M:%S"
am_pm              ;
t_fmt_ampm         ;
END LC_TIME

```

Glossary

ASCII

American Standard Code for Information Interchange. ASCII is the traditional UNIX codeset and defines 128 characters, including both control characters and graphic characters, represented by 7-bit binary values (see also ISO 646).

See also: character set, codeset, Portable Character Set

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 multibyte or single-byte value. The expression “multibyte character” is synonymous with the term “character;” that is, 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, codeset, Portable Character Set

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

coded character set

See: codeset

codeset

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.

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.

decomposed character

In Unicode, a character sequence that uses a base character, such as e, followed by a combining character, such as acute (´), to represent a single character in a native language.

See also: precomposed character

file code

The encoding format that applies to data outside the program.

See also: process code

graphic character

A character, other than a control character, that has a visual representation when hand-written, printed, or displayed.

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. The mnemonic I18N is frequently used as an abbreviation for internationalization.

See also: locale, localization

ISO 10646

The ISO Universal character set. 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.

ISO8859-1

ISO 8-bit single-byte codeset Part 1, Latin Alphabet No. 1. The ISO8859-1 character set comprises 191 graphic characters covering the requirements of most Western European languages.

L10N

See: localization

LANG

An environment variable that specifies the locale to use for all locale categories. The following environment variables can be set to override the LANG setting in specific locale categories:

- LC_COLLATE, for information on how to order characters and strings in sorting, or collation, operations
- LC_CTYPE, for definitions of classes and attributes of characters used operations such as case conversion
- LC_MESSAGES, for definitions of strings that are valid for affirmative and negative responses
- LC_MONETARY, for rules and symbols used to format monetary values

- `LC_NUMERIC`, for rules and symbols used to format numeric values
- `LC_TIME`, for information related to date and time

The `LC_ALL` environment variable also specifies locale. If set, this variable overrides all the preceding variables, including `LANG`.

See also: locale

langinfo database

See: locale

local language

See: native language

locale

A set of data, sometimes referred to as the “langinfo database,” that supports a particular combination of native (local) language, cultural data, and codeset.

See also: codeset, cultural data, `LANG`, localization

localization

The process of implementing for an application the requirements for local languages and customs. 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. The mnemonic L10N is frequently used as an abbreviation for localization.

See also: internationalization, locale

LOCPATH

An environment variable used to specify the search path for locales.

See also: locale

message catalog

A file or storage area containing 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, Italian, or Spanish.

NLSPATH

An environment variable used to indicate the search path for message catalogs.

Portable Character Set

A character set that is supported in both compile-time (source) and run-time (executable) environments for all locales and that contains:

- The 26 uppercase letters of the English 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 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 Portable Character Set contains control characters that represent alert, backspace, carriage return, and new line.

The Portable Character Set 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 set 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 Portable Character Set. An example of character substitution might be the British pound sign (£) for the number sign (#).

See also: character set, codeset, ISO 646

precomposed character

In Unicode, a discrete code point that represents a sequence of a base character, such as e, with a combining character, such as acute (').

See also: decomposed character, Unicode

process code

The wide-character encoding format used for manipulating data inside programs.

See also: file code

radix character

The character that separates the integer part of a number from the fractional part.

string

See: character string

UCS

See: ISO 10646

Unicode

A codeset (maintained by the Unicode consortium) that uses a generalized multibyte encoding format to accommodate characters in all native languages. Unicode is code-for-code identical with the UCS-2 form of ISO 10646.

See also: codeset, 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 `wchar_t`, which is defined in the header files `/usr/include/stddef.h` (for conformance to the *X/Open Portability Guide*) and `/usr/include/stdlib.h` (for conformance to the ANSI C standard). 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 (not the case for DEC OSF/1) might define `wchar_t` as a byte 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

Index

A

- acode option of stty command, 6-16
 - effect on codeset conversion of mail messages, 6-79
- add_wch function, 4-2
- add_wchnstr function, 4-4
- add_wchstr function, 4-4
- addnwstr macro, 4-5
- addwch macro, 4-2
- addwchnstr macro, 4-4
- addwchstr macro, 4-4
- addwstr macro, 4-5
- adec option of stty command, 6-14
- ASCII codeset, 2-2
- asctime function, A-5
- Asian language support, 2-4
- asort command, 6-79
 - collating user-defined characters, 6-42

B

- backslash character
 - coding in message strings, 3-4
- backspace character
 - coding in message strings, 3-4
- bdf entry in cp_dirs file, 6-59
- bdf font format
 - restrictions when tuning X server cache, 6-88n
- bdf option of cgen command, 6-43
- bit patterns
 - coding in message strings, 3-4

C

- C option of cedit command, 6-26
- can-space-after characters, 6-82
- can-space-before characters, 6-82
- carriage return
 - (*See return character*)
- case conversion, 2-11, A-3
- catclose function
 - argument, 3-34
 - using with nl_catd descriptor type, 2-18
- catgets function
 - arguments, 3-34
 - as argument to printf function, 2-19
 - detecting catalog open failures with, 3-33
 - in program-defined macro, 3-35
 - using with puts function, 2-18
- catopen function, 3-28
 - arguments, 3-29
 - behavior when effective user ID is root, 3-34n
 - failure to return error status, 3-33
 - performance overhead, 3-24
 - troubleshooting problems with, 3-32
 - use of NLSPATH environment variable, 3-29
 - using with nl_catd descriptor type, 2-18
- cc command

- for compiling locale method definitions, 7-57e
 - support for trigraph sequences, 2-10
- CDA commands, 6-112
 - Asian-language fonts used for PostScript files, 6-113
 - cdoc, 6-113
 - converting ddif files to PostScript files, 6-113
 - dxvdoc, 6-113
 - option files
 - specifying for Asian languages, 6-112
 - text_encoding keywords for, 6-113
 - support for vertical writing, 6-114n
 - using with Asian-language text files, 6-112
- cdb entry in cp_dirs file, 6-59
- cdoc command, 6-113
- cedit command, 6-25
 - changing language of user interface messages, 6-30
- font-editing screen, 6-31
 - bitmap data buffers, 6-35
 - editing function keys, 6-40
 - editing functions, 6-36
 - editing modes, 6-34
 - function keymap, 6-37
 - keys for drawing, 6-39
 - keys for mode switching, 6-38
 - keys for window areas, 6-39
 - keys to control cursor, 6-38
 - miscellaneous function keys, 6-37
- options and arguments for, 6-26
- user interface screen, 6-26
- cgen command, 6-42
 - bdf option
 - alternative to fontconverter command, 6-45
 - compared to font renderer, 6-91
 - options for, 6-43
 - pcf option
 - alternative to fontconverter command, 6-45
- character classes
 - defining in a locale, 7-9
- character collation functions, A-4
- character map files
 - (See *charmap files*)
- character sets
 - (See *codesets*)
- character string, 1-5
 - empty, 1-6
- character-attribute databases
 - (See *UDC databases*)
- characters
 - collation of, 2-12
 - compared with char data type elements, 1-5
 - converting case of, 2-11
 - deleting on command line, 6-16
 - encoding for locales, 7-4
 - identifying classes of, 2-10
 - ideographic
 - tuning X Server cache for, 6-88
 - multibyte, 1-5
 - writing methods to convert, 7-23
 - user-defined in Asian languages, 6-23
 - wide, 1-5
- charmap files, 7-1
 - character encoding in, 7-4
 - character symbols in, 7-4
 - standardization of, 7-7n
- charset field in mail headers, 6-77
 - use by comsat server, 6-78
- Chinese languages
 - phrase input method, 6-49
- class.dat file of phrase database, 6-50
- clause option of stty command, 6-20
- Clear Display option

- effect on ruled lines in
 - DECterm window, 4-22
- client-server display environment
 - font installation requirements, 6-92
- code option of stty command, 6-16
- codesets, 1-2
 - ASCII, 2-2
 - using the most significant bit of a byte, 2-5
 - conversion of
 - application codeset to terminal codeset, 6-15
 - by comsat server, 6-78
 - by mailx and MH, 6-76
 - for data files, 6-83
 - for print jobs, 6-63
 - converting files from one codeset to another, A-13
 - creating, 7-1
 - for mail interchange, 6-77
 - for support of Asian languages, 2-3
 - ISO, 2-2
 - use in locales, 2-2
 - null characters in, 2-9
 - problems when using
 - assuming single-byte characters, 2-5
 - case conversion, 2-11
 - character classification, 2-10
 - comparing strings, 2-12
 - data transparency, 2-5
 - handling multibyte characters, 2-7
 - in-code literals, 2-5
 - referring to octal values, 2-5
 - setting name of, 7-4
 - source and execution versions of, 2-9
 - state-dependent encoding of characters in, 2-9
 - used over networks and interchange media, 2-3
 - user application
 - defining for mailx and MH, 6-78
- col option of cgen command, 6-43
- COLL_WEIGHTS_MAX variable, 7-13
- collating sequence
 - character order in non-English languages, 1-3
- collating tables
 - creating for user-defined characters, 6-42
- collating value databases
 - setting default locations of, 6-58
- collation
 - algorithms, 2-12
 - defining in a locale, 7-11
 - functions used for, 2-12
 - maximum number of levels, 7-13
 - performance issues, 2-13
- collation order
 - defining in locale source file, 7-11
- command-line editing, 6-16
 - commands for, 6-17
 - history mode, 6-17
- comment character
 - in methods file, 7-58
- comments
 - in charmap file, 7-3
 - in locale definition source file, 7-7
 - in message set directives, 3-6
- Compose key, 6-5
- compound strings
 - in Motif applications, 5-5
 - creating, 5-6
- comsat server
 - codeset conversion done by, 6-78
- constants
 - using non-English characters as, 2-6
- cp_dirs file, 6-58
 - default entries in, 6-59e

- cs option in X Server
 - configuration file, 6–88
- ctime function, A–5
- cu option in X Server
 - configuration file, 6–88
- cultural data, 1–2
 - currency symbols, 2–13
 - date formats, 2–13
 - radix character, 2–13
 - stored in langinfo database, 2–13
 - thousands separator, 2–13
- currency symbols
 - defining international, 7–17e
 - defining local, 7–17e
 - determining with localeconv function, 2–16
 - variation for, 2–13
- curses library
 - enhancements for multibyte characters, 4–1
 - overwriting multicolumn characters, 4–2n
 - support for wide-character data, 4–1

D

- D_FMT constant
 - using with strftime function, 2–14
- D_T_FMT constant
 - using with nl_langinfo function, 2–14
- data files
 - converting
 - ddif format to PostScript format, 6–113
 - from one codeset to another, 6–83
 - text files with CDA commands, 6–112
 - database location configuration file (*See cp_dirs file*)
 - date format

- defining in locale source file, 7–21e
 - era construct, 7–23
- dates
 - differences in formats for, 2–13
 - formatting, 2–20, A–5
 - generating strings for, 2–14
- DCH escape sequence, 4–22
- ddif file format
 - converting to PostScript format, 6–113
- dec option of stty command, 6–14
- DECCOLM escape sequence, 4–22
- DECDHLB escape sequence, 4–22
- DECDHLT escape sequence, 4–22
- DECDRLBR escape sequence, 4–17
- DECDWL escape sequence, 4–22
- DECERLBRA escape sequence, 4–20
- DECERLBRP escape sequence, 4–19
- decimal point
 - (*See radix character*)
- DECclaser 1152
 - downloading fonts to, 6–74
 - PostScript filters for, 6–62
- DECclaser 1152 printer
 - printcap entries for, 6–63
- DECclaser 5100
 - downloading fonts to, 6–75
 - PostScript filters for, 6–62
- DECSTR escape sequence, 4–22
- DECterm software
 - drawing ruled lines in window, 4–16
 - determining support for, 4–24
 - erasing ruled lines in window
 - in specified area, 4–20
 - in specified pattern, 4–19
 - features and restrictions for Asian languages, 6–111
 - resource file, 6–111
 - ruled lines in window
 - effect of standard escape sequences, 4–22
- DECwindows

- creating UDC fonts for
 - editing glyphs, 6–31
 - in bdf and pcf format, 6–43
- delch macro, 4–8
- delset directive, 3–7
 - position in message source file, 3–8
- digit grouping size
 - determining with localeconv function, 2–16
- DL escape sequence, 4–22
- downline loadable characters, 4–24
- DRCS characters, 4–24
- dspcat command, 3–26
- dspmsg command, 3–26
- dxkeycaps utility, 6–6
- DXmCSText widget, 5–5
- dxvdoc command, 6–113

E

- ECH escape sequence, 4–22
- echo_wchar function, 4–2
- echowchar macro, 4–2
- ED escape sequence, 4–22
- EL escape sequence, 4–22
- Emacs editor
 - multilingual, 6–85
 - __mbstopcs method, 7–24
 - __mbtopc method, 7–27
 - __pcstombs method, 7–32
 - __pctomb method, 7–34
- erase option of stty command, 6–17
- errno
 - setting in threadsafe manner, 7–37
- esc.alw and -esc.alw options of stty command, 6–20
- escape character
 - coding in message strings, 3–4

- setting
 - for nroff command, 6–81
 - in charmap file, 7–3
 - in locale definition source file, 7–8
- examples, online
 - ximdemo application, 5–7
 - xpg4demo application, 2–1
- excode definition
 - in .mailrc file, 6–78
 - in .mh_profile file, 6–78
- EXCODE environment variable, 6–78
- exit function
 - effect on open message catalogs, 3–34
- extract command, 3–15

F

- f option of localedef command, 7–59
- ffd daemon, 6–75
- ffserver command, 6–75
- fgetc function, A–9
- fgets function, A–9
- fgetwc function, A–9
- fgetws function, A–9
 - writing a method for, 7–24
- file code, 2–8
- file names
 - multibyte characters in, 6–86
- font renderers, 6–89
 - Asian PostScript font renderer, 6–90
 - configuration file, 6–90
 - UDC font renderer, 6–91
- font sets, 5–10
 - converting encoding in Xt applications, 5–3
 - converting encoding of, 5–17
 - creating and using, 5–11
 - drawing text with, 5–13
 - obtaining metrics for, 5–13

- font option of fontconverter
 - command, 6-46
- font-faulting mechanism, 6-74
 - print queue management for, 6-74
 - server for, 6-75
- fontconverter command
 - options and arguments, 6-46
- fonts
 - (*See also cedit command, cgen command, fontconverter command*)
 - bitmap
 - CDE font alias files, 6-93
 - displaying installed, 6-94
 - for Chinese, 6-94
 - for Czech, 6-95
 - for Greek, 6-103
 - for Hebrew, 6-105
 - for Hungarian, 6-95
 - for Japanese, 6-94
 - for Korean, 6-94
 - for Lithuanian, 6-98
 - for Polish, 6-95
 - for Russian, 6-101
 - for Slovak, 6-95
 - for Slovene, 6-95
 - for Thai, 6-94
 - for Turkish, 6-107
 - compiled
 - making available to X applications, 6-45
 - compressed, 6-88n
 - creating UDC files for, 6-42
 - creating user-defined glyphs, 6-30
 - files for user-defined characters
 - creating for DECwindows Motif, 6-33
 - creating for system software, 6-33
 - setting default location of, 6-58
 - in DECterm windows
 - sizes supported for ideographic character sets, 6-111
 - listing local-language fonts in Mwm file, 6-92
 - on systems that use remote display, 6-92
 - PostScript
 - downloading to Asian printers, 6-70
 - other than built-in, 6-70
 - restrictions when tuning X
 - server cache, 6-88n
 - setting style and size for local-language printer, 6-63
 - tuning X Server cache for ideographic, 6-88
 - use by CDA commands for PostScript output, 6-113
- form-feed character
 - coding in message strings, 3-4
- format specifiers
 - in output text strings, 2-20
 - used with input text strings, 2-20
- formatting
 - date and time
 - D_T_FMT constant, 2-14
 - strftime function, 2-14
 - input text, 2-20
 - messages, 2-19
 - monetary values, 2-16
 - numeric values, 2-16
 - output text, 2-19
 - format specifiers for, 2-20
- fprintf function, A-5
- fprop option of cgen command, 6-43
- fputs function, A-9
- fputws function, A-9
 - writing a method for, 7-32
- fscanf function, A-5
- fwide function, A-13
- fwprintf function, A-6
- fwscanf function, A-6

G

gencat command, 3-22
 defined by X/Open, 2-17
 deleting a message set with, 3-8
 handling message source
 modifications for, 3-22
 handling of delset directive by,
 3-7
 interactive use of, 3-19
 lines ignored by, 3-11
 processing multiple source files
 with, 3-25
 use in makefile, 3-20
 using with -g option of dspcat
 command, 3-26
get_wch function, 4-13
get_wstr function, 4-12
getc function, A-9
 restricted use of, 2-7
getch function, 4-13
getchar function, A-9
 problems with multibyte
 characters, 2-5
getn_wstr function, 4-12
getnwstr macro, 4-12
gets function, A-9
 restricted use of, 2-7
gettxt function, 3-35n
getwc function, A-9
 writing a method for, 7-27
getwch function, 4-13
getwchar function, A-9
getwstr macro, 4-12
GSM escape sequence, 4-22

H

-h option of cedit command, 6-26
-h option of fontconverter
 command, 6-46
Hankaku characters in the
 Japanese language, 6-18

henkan option of stty command,
 6-20
Hiragana characters in the
 Japanese language, 6-18
history and -history options of stty
 command, 6-17
Hold Screen key, 6-85

I

-i option of localedef command, 7-59
I18N mnemonic, 1-1
ICH escape sequence, 4-22
iconv command, 6-83, A-13
iconv function, A-13
iconv_close function, A-13
iconv_open function, A-13
ignore file, 3-16
ikk and -ikk options of stty
 command, 6-20
iks entry in cp_dirs file, 6-59
-iks option of cgen command, 6-43
IL escape sequence, 4-22
imode option of stty command, 6-20
in_wch function, 4-9
in_wchnstr function, 4-10
in_wchstr function, 4-10
innwstr macro, 4-11
input
 handling in X applications, 5-21
input method servers
 handling failure of in client
 programs, 5-36
 when connection to DECterm
 window breaks, 6-111
input methods, 6-7
 (*See also XIC object, XIM
 object*)
 default, 5-8
 determining in X applications
 precedence order for, 5-22

- filtering events for
 - FocusIn and FocusOut, 5–35
 - in X applications, 5–33
 - KeyPress, 5–35
 - KeyRelease, 5–35
- interaction styles for, 5–23
 - on-the-spot, 5–29
 - supported by locales, 5–25
- off-the-spot
 - requires auto-resize be
 - enabled, 5–6n
 - opening and closing in X
 - application, 5–23e
 - preediting styles for, 6–7
 - selecting, 6–8
 - starting servers for, 6–9
- ins_nwstr function, 4–7
- ins_wch function, 4–3
- ins_wstr function, 4–7
- insnwstr macro, 4–7
- inswch macro, 4–3
- inswstr macro, 4–7
- internal process code, 2–8
- internationalization, 1–1
- inwch macro, 4–9
- inwchnstr macro, 4–10
- inwchstr macro, 4–10
- inwstr macro, 4–11
- IRM escape sequence, 4–22
- iscntrl function, A–1
- isdigit function, A–1
- isgraph function, A–1
- islower function, A–1
- ISO C standard
 - 1994 amendment, A–1
- ISO codesets, 2–2
- ISO-2022-JP, 6–77
- ISO/IEC 10646 standard, 2–4
- isprint function, A–1
- ispunct function, A–1
- isspace function, A–1
- isupper function, A–1
- iswalnum function, A–1
- iswalpha function, A–1

- iswcntrl function, A–1
- iswctype function, A–2
- iswdigit function, A–1
- iswgraph function, A–1
- iswlower function, A–1
- iswprint function, A–1
- iswpunct function, A–1
- iswspace function, A–1
- iswupper function, A–1

J

- jdec option of stty command, 6–14
- jinkey option of stty command, 6–20
- JSYKKSEQ environment variable,
 - 6–20
- justification of text by nroff, 6–82
- jx and -jx options of stty command,
 - 6–20

K

- Kana-Kanji conversion, 6–18
 - changing key map for, 6–20
 - dictionaries used with, 6–20
 - displaying key map for, 6–20
- Kanji characters in the Japanese
 - language, 6–18
- Katakana characters in the
 - Japanese language, 6–18
- kb_indicator command, 6–10
- keyboards
 - determining layout of selected,
 - 6–6
 - determining mode switch state
 - of, 6–10
 - entering basic characters not
 - supported on, 2–10
 - Keyboard Indicator utility, 6–10
 - Keyboard Options dialog box,
 - 6–6
 - obtaining composed strings
 - from, 5–34

- selecting for different languages, 6–5
- .ki command for nroff, 6–81
- kin option of stty command, 6–20
- kkcd daemon, 6–20
- kkmap option of stty command, 6–20
- kkseq option of stty command, 6–20
- knj.bsl and -knj.bsl options of stty command, 6–20
- knj.sp and -knj.sp options of stty command, 6–20
- .ko command for nroff, 6–81
- kout option of stty command, 6–20

L

- L10N mnemonic, 1–3
- LANG environment variable
 - effect on man command's search path, 6–83
 - effect on setlocale function, 2–22
 - including locale file name suffix, 6–3n
 - interaction with %L in search paths, 6–3n
 - interaction with NLSPATH setting, 3–31
 - use by mailx and MH, 6–78
 - use when generating message catalogs, 3–19
- .lang_device_name file, 6–79
- langinfo database, 1–4
 - compared to message catalogs, 3–1
 - information contained in, 2–13
 - querying, 2–14
- language
 - announcement, 1–5
 - implications for internationalized software, 1–1
 - syntax constructions, 2–19
- language variant subsets
 - documentation for, 6–2
- LC_COLLATE category
 - defining in locale source file, 7–11
- LC_CTYPE category
 - defining in locale source file, 7–9
- LC_CTYPE locale category
 - classes defined for, A–3
- LC_MESSAGES category
 - defining in locale source file, 7–15
- LC_MESSAGES variable
 - interaction with NLSPATH setting, 3–31
 - use by setlocale function, 2–18
- LC_MONETARY locale category
 - defining in locale source file, 7–17
- LC_NUMERIC locale category
 - defining in locale source file, 7–20
- LC_TIME locale category
 - defining in locale source file, 7–20
- ld command
 - for building a locale methods library, 7–57e
- libiconv library, A–13
- line wrapping by nroff, 6–81
- locale command, 3–27
- localeconv function, A–4
 - advantages of, 2–16
- localedef command, 7–59e
 - building shareable library with, 7–60
 - compiling methods files with, 7–60
 - cv options, 7–57
 - f option, 7–59
 - i option, 7–59
 - m option, 7–60
 - running in verbose mode, 7–60
 - w option, 7–60
- locales
 - binding program to locales set by users, 2–22
 - building, 7–59
 - categories in, 2–21
 - defining, 7–7

- changing setting for specific
 - category of, 2-24
- changing within program, 2-22
- checking for duplicate
 - definitions, 7-60
- compared to message catalogs,
 - 3-1
- creating, 7-1
- default setting for, 6-3
- default system location of, 7-59
- displaying information about,
 - 3-27
- initializing at run time, 2-21
- location of
 - LOCPATH variable, 7-60
- objects in X applications
 - affected by, 5-8
- provided with localized
 - systems, 2-2
- provided with standard system,
 - 2-2
- retrieving locale data from
 - scripts, 3-26
- setting, 6-3
 - in Motif applications, 5-4
 - in X applications, 5-7, 5-9e
 - in Xt applications, 5-2
- source files for
 - charmap file, 7-1
 - locale definition file, 7-7
- specifying to setlocale function,
 - 2-22
- testing, 7-60
- using font sets with in X
 - applications, 5-10
- using locale name extensions in
 - variable settings, 6-3n
- using name extensions for, 7-59
- localization, 1-3
- localtime function
 - using with strftime function, 2-14
- LOCPATH environment variable,
 - 7-60
- effect on iconv command, 6-84

- lowercase characters
 - testing for, 2-10
- lp command
 - local-language printer support,
 - 6-69
- lpc command
 - local-language printer support,
 - 6-69
- lpd printer daemon
 - local-language printer support,
 - 6-69
- lpq command
 - local-language printer support,
 - 6-69
- lpr command
 - local-language printer support,
 - 6-69
- lprm command
 - local-language printer support,
 - 6-69
- lprsetup command
 - Asian printers supported by, 6-69
 - setting up Asian printers with,
 - 6-67
- lpstat command
 - local-language printer support,
 - 6-69

M

- m option of localedef command,
 - 7-60
- mail messages
 - codeset conversion of, 6-76
 - notification of incoming, 6-78
- mail-codesets file
 - use by comsat server, 6-78
 - use by mailx and MH, 6-77, 6-78
- .mailrc file
 - defining application codeset in,
 - 6-78
 - defining mail interchange
 - codeset in, 6-78
- mailx command, 6-76

- man command, 6–83
 - and reference page
 - translations, 6–83
- manpages
 - (*See reference pages*)
- mblen function, A–7
 - writing a method for, 7–35
- mbsinit function, A–13
- mbstowcs function, 2–8, A–7
 - writing a method for, 7–38
- mbtowc function, 2–6, A–7
 - writing a method for, 7–40
- merge option of cgen command, 6–43
- merge option of fontconverter command, 6–46
- message catalogs, 1–4
 - (*See also messages, message sets*)
 - closing, 3–34
 - compared to locales, 3–1
 - converting existing program to use, 3–15
 - converting to source format, 3–26
 - design and maintenance considerations, 3–22
 - detecting file open failures, 3–33
 - determining locale to use with, 2–18
 - displaying contents of, 3–26
 - file name extension of, 3–22
 - finding when effective user ID is root, 3–34n
 - generating for different locales, 3–19
 - generating from message source files, 3–18
 - how interfaces determine location of, 2–19
 - installing in nondefault locations, 3–30
 - order of message sets in, 3–6
 - portability of, 3–22
 - program access to, 3–28
 - retrieving messages from, 2–18
 - script access to, 3–26
 - source files for, 3–2
 - blank lines in, 3–5
 - comment lines in, 3–11
 - editing, 3–18
 - file name extension of, 3–22
 - general syntax rules, 3–4
 - quoting strings in, 3–4
 - set directives in, 3–6
 - translating, 3–18
 - translating
 - date formats, 2–21
 - passive-verb constructions, 2–19
 - programmer comments to help translator, 3–11
 - trans command, 3–18
 - word order changes, 2–20
 - using to define non-English constants, 2–6
- message sets, 3–6
 - (*See also message catalogs, messages*)
 - default message set, 3–6
 - deleting, 3–7
 - specifying identifiers for, 3–6
 - symbolic identifiers for, 3–20
- messages
 - (*See also message catalogs, message sets*)
 - changing to empty string, 3–9
 - coding special characters in, 3–4
 - construction of strings in, 2–19
 - deleting, 3–9
 - deleting individual, 3–10
 - displaying from message catalog, 3–27
 - enabling and disabling quotation delimiter for, 3–10
 - format in message source file, 3–8
 - identifiers for, 3–8
 - language constraints for, 2–17

- line continuation in source entries, 3-5
- maximum length of, 3-9
- order within sets, 3-8
- ordering of elements in, 2-20
- preceding and trailing spaces in, 3-4
- reading into program, 3-34
- style guidelines for, 3-12
- symbolic identifiers for, 3-20
- methods, 7-23
 - arguments expected for, 7-57
 - building shareable libraries for, 7-57e
 - __mbstopcs, 7-24
 - __mbtopc, 7-27
 - __pcstombs, 7-32
 - __pctomb, 7-34
 - file to specify library to
 - localedef command, 7-57
 - mblen, 7-35
 - mbstowcs, 7-38
 - mbtowc, 7-40
 - optional, 7-55
 - required, 7-24
 - specifying to localedef command, 7-60
 - wcstombs, 7-45
 - wcswidth, 7-51
 - wctomb, 7-48
 - wcwidth, 7-53
- MH (mail handler) system, 6-76
- .mh_profile file
 - defining application codeset in, 6-78
 - defining mail interchange codeset in, 6-78
- mkcatdefs command, 3-20
 - automatic delset directives by, 3-9
 - automatic insertion of delset directives by, 3-10
 - deleting messages with, 3-9
 - handling of delset directive by, 3-7
 - header file produced by, 3-19
 - interactive use of, 3-19
 - lines ignored by, 3-11
 - processing multiple source files with, 3-25
 - use in makefile, 3-20
 - use restrictions and guidelines, 3-21
- mkfontdir command, 6-45
- MNLS
 - (*See System V Multi-National Language Supplement*)
- monetary values
 - (*See also currency symbols, numeric values*)
 - formatting, 2-16
- month names
 - defining in locale source file, 7-21e
- more command, 6-85
- Motif applications, 5-3
 - handling messages in, 3-1
 - setting language in, 5-4
 - support for bidirectional text display, 5-6n
 - text translation issues, 3-12
 - using font sets in, 5-4
 - using text widgets in, 5-4
- mule command, 6-85
- multibyte characters, 1-5
 - command-line editing, 6-16
 - compared to wide characters, 2-8
 - converting to wide characters, 2-6
 - converting to wide-character format, 7-23
 - in file and user names, 6-86
 - interfaces for manipulating, 2-7
 - setting terminal line discipline for, 6-13
 - testing for, 2-6

- multithreaded applications
 - setting errno for, 7-37
- mvadd_wch function, 4-2
- mvadd_wchstr function, 4-4
- mvaddnwstr macro, 4-5
- mvaddw_wnstr function, 4-4
- mvaddwch macro, 4-2
- mvaddwchnstr macro, 4-4
- mvaddwchstr macro, 4-4
- mvaddwstr macro, 4-5
- mvdelch macro, 4-8
- mvget_wch function, 4-13
- mvget_wstr function, 4-12
- mvgetch function, 4-13
- mvgetn_wstr function, 4-12
- mvgetnwstr macro, 4-12
- mvgetwch function, 4-13
- mvgetwstr macro, 4-12
- mvin_wch function, 4-9
- mvin_wnstr function, 4-10
- mvin_wchstr function, 4-10
- mvinnwstr macro, 4-11
- mvins_nwstr function, 4-7
- mvins_wch function, 4-3
- mvins_wstr function, 4-7
- mvinsnwstr macro, 4-7
- mvinswch macro, 4-3
- mvinswstr macro, 4-7
- mvinwch macro, 4-9
- mvinwchnstr macro, 4-10
- mvinwchstr macro, 4-10
- mvinwstr macro, 4-11
- mvprintw function, 4-15
- mvscanw function, 4-14
- mvw_getwch function, 4-13
- mvwadd_wch function, 4-2
- mvwadd_wnstr function, 4-4
- mvwadd_wchstr function, 4-4
- mvwaddnwstr macro, 4-5
- mvwaddwch macro, 4-2
- mvwaddwchnstr macro, 4-4
- mvwaddwchstr macro, 4-4
- mvwaddwstr macro, 4-5
- mvwdelch function, 4-8

- mvwdelch macro, 4-8
- mvwget_wstr function, 4-12
- mvwgetch function, 4-13
- mvwgetn_wstr function, 4-12
- mvwgetnwstr macro, 4-12
- mvwgetwch function, 4-13
- mvwgetwstr macro, 4-12
- mvwin_wch function, 4-9
- mvwin_wnstr function, 4-10
- mvwin_wchstr function, 4-10
- mvwinnwstr macro, 4-11
- mvwins_nwstr function, 4-7
- mvwins_wch function, 4-3
- mvwins_wstr function, 4-7
- mvwinsnwstr macro, 4-7
- mvwinswch macro, 4-3
- mvwinswstr macro, 4-7
- mvwinwch macro, 4-9
- mvwinwchnstr macro, 4-10
- mvwinwchstr macro, 4-10
- mvwinwstr macro, 4-11
- mvwprintw function, 4-15
- mvwscanw function, 4-14
- Mwm resource files
 - listing local-language fonts in, 6-92
- MwmFontSetup script, 6-93

N

- National Replacement Character Set
 - customization restriction, 6-111
- negative sign
 - defining for monetary values, 7-17e
 - determining with localeconv function, 2-16
- neqn command
 - using with tbl and nroff, 6-82
- newline character
 - coding in message strings, 3-4
- NL_CAT_LOCALE constant, 2-19
- nl_catd type, 3-29

- declaring and using in program, 2-18
- nl_langinfo function, A-4
 - and langinfo database, 2-14
 - using as argument to strftime function, 2-15
 - value returned for CODESET parameter, 7-4
- NL_MSGMAX constant, 3-8
- NL_SETD constant, 3-6
- NL_SETMAX constant, 3-6
- NL_TEXTMAX constant, 3-9
- NLSPATH environment variable, 3-29
 - interaction with
 - LC_MESSAGES setting, 3-31
 - substitution fields in setting of, 3-30
 - use by catclose function, 2-19
 - use by catopen function, 2-19, 3-29
 - when setting is ignored by catopen function, 3-34n
- no responses
 - defining in locale, 7-15
- no-first characters, 6-81
 - defining private set of, 6-81
- no-last characters, 6-81
 - defining private set of, 6-81
- noexpr keyword, 7-16
- nostr keyword, 7-17
- NRCS
 - (See *National Replacement Character Set*)
- nroff command, 6-80
 - handling of ideographic characters, 6-82
 - output for Asian languages
 - support in print filters for, 6-61
 - right-justification rules, 6-82
 - rules for wrapping lines, 6-81
- null characters, 2-9

- numeric conversion, A-7
- numeric values
 - customized formatting by program, 2-16

O

- O option of CDA commands, 6-113
- octal values
 - coding in message strings, 3-4
- odl and -odl options of stty command, 6-24
- odl entry in cp_dirs file, 6-59
- odl option of cgen command, 6-43
- odlall option of stty command, 6-24
- odldb option of stty command, 6-24
- odlreset option of stty command, 6-24
- odlsize option of stty command, 6-24
- odltype option of stty command, 6-24
- off-the-spot input style
 - requires auto-resize be enabled, 5-6n
 - text widget that supports, 5-5
- off-the-spot style for input
 - methods, 6-7
- on-demand loading of UDC databases, 6-23
- on-the-spot input style, 5-29
 - text widget that supports, 5-5
- on-the-spot style for input
 - methods, 6-7
- osiz option of cgen command, 6-43
- output contexts, 5-16
- output methods, 5-16
- over-the-spot input style
 - text widget that supports, 5-5
- over-the-spot style for input
 - methods, 6-7

P

- parentheses characters

- inappropriate line wrapping of, 6–81
- patterns file, 3–16
- Pause window
 - changing language of, 6–92
- pcf entry in cp_dirs file, 6–59
- pcf font format
 - restrictions when tuning X server cache, 6–88n
- pcf option of cgen command, 6–43
- performance tradeoffs
 - collation, 2–13
- pfsetup command, 6–74
- phrase databases, 6–49
 - classes in, 6–50
 - defining
 - classes for, 6–55
 - phrases for, 6–54
 - effect of locale setting on, 6–55
 - files in, 6–50
 - guidelines and restrictions for, 6–54
 - maintaining, 6–51
 - setting default locations of, 6–58
 - using, 6–56
- phrase input method, 6–49
 - customizing for Asian terminals, 6–50
 - in DECterm windows, 6–49
 - on other VT382 series terminals, 6–49
 - on VT382-D terminal, 6–49
- phrase utility
 - command-line options, 6–52
 - invoking, 6–51
 - screen interface, 6–52
 - menus, 6–53
- phrase.dat file of phrase database, 6–50
- Portable Character Set, 1–6
 - substituting characters in, 1–7
 - using characters as literals and constants, 2–6
- positive sign
 - defining for monetary values, 7–17e
 - determining with localeconv function, 2–16
- PostScript fonts
 - downloading to printers, 6–70
- pre entry in cp_dirs file, 6–59
- pre option of cgen command, 6–43
- preedit strings
 - attributes for, 5–27
- preediting strings
 - handling in X application, 5–29
- preload option of fontconverter command, 6–46
- print filters
 - for Asian-language text files, 6–61
 - for local-language printers, 6–61
 - for PostScript files, 6–62
- printcap file
 - symbols for local-language printers, 6–63
- printers
 - (*See also print filters, printcap file*)
 - downloading PostScript fonts to, 6–70
 - setting up, 6–67
 - setting up queues for Asian PostScript, 6–74
 - subsystem enhancements for Asian languages, 6–60
 - supported for different Asian languages, 6–69
- printf function, A–5
 - international format specifiers for, 2–20
 - restricted use of, 2–7
 - using catgets function as argument to, 2–19
- printw function, 4–15
- profile component of the .mailrc file, 6–78
- profile component of the .mh_profile file, 6–78

- punctuation characters
 - inappropriate line wrapping of, 6–81
- putc function, A–9
 - restricted use of, 2–7
- puts function, A–9
 - restricted use of, 2–7
- putwc function, A–9

Q

- quote directive, 3–10

R

- r option of cedit command, 6–26
- radicals, 1–3
- radix character
 - defining
 - for monetary values, 7–17e
 - for numeric values, 7–20e
 - determining with localeconv function, 2–16
 - functions that extract from langinfo database, 2–17
 - variation for, 2–13
- reference character attribute
 - databases, 6–26
- reference pages
 - location of translated files, 6–83
 - printing, 6–82
 - processing non-English, 6–80
- remote display
 - font requirements for Asian languages, 6–92
- Reset Terminal option
 - effect on ruled lines in DECterm window, 4–22
- resource databases
 - decxim.preeditdraw.encoding resource, 5–32
 - decxim.statusdraw.encoding resource, 5–32

- handling localized, 5–20
- resource files
 - sharing among locales, 6–111
- response strings
 - defining in locale, 7–15
- return character
 - coding in message strings, 3–4
- RIS escape sequence, 4–22
- rlogin command, 6–85
- root-window input style
 - text widget that supports, 5–5
- root-window style for input
 - methods, 6–7
- ruled lines in DECterm window
 - determining if supported, 4–24
 - drawing, 4–16, 4–17
 - specifying length of, 4–19
 - specifying start point, 4–19
 - effect of Clear Display on, 4–22
 - effect of Reset Terminal on, 4–22
 - effect of standard escape sequences on, 4–22
 - erasing
 - in specified area, 4–20
 - in specified pattern, 4–19
 - specifying bitmask pattern for, 4–19

S

- scanf function, A–5
 - international format specifiers for, 2–20
 - restricted use of, 2–7
- scanw function, 4–14
- screen handling
 - for character-cell terminals, 4–1
- scripts
 - retrieving locale data from, 3–26
 - using message catalogs from, 3–26
- sd symbol in /etc/printcap file, 6–63
- sendmail utility
 - configuring for 8-bit data, 6–76

- set directive, 3–6
 - (*See also message sets*)
- setlocale function, A–1
 - binding to preset locales, 2–22
 - changing locale setting with, 2–22
 - changing specific locale
 - category, 2–24
 - initializing locale, 2–21
- shareable libraries
 - for locale methods, 7–57
 - specifying in methods file, 7–58e
- shared libraries
 - to support locale methods, 7–23
- shell scripts
 - (*See scripts*)
- shift states, 2–9
- sim and -sim options of stty
 - command, 6–51
- sim entry in cp_dirs file, 6–59
- sim option of stty command, 6–56
- SIM service, 6–49
 - enabling, 6–50
- simall option of stty command, 6–51
- simclass option of stty command,
 - 6–51, 6–56
- simdb option of stty command,
 - 6–51, 6–56
- simkey option of stty command,
 - 6–51, 6–56
- SoftODL service, 6–24, 6–42
 - settings used by
 - character replacement
 - method for printer, 6–63
 - size of cache for printer, 6–63
- sort command
 - (*See asort command*)
- sort directives
 - maximum number of, 7–13
- sort rules
 - defining in locale source file, 7–11
- sorting characters in different
 - languages, 6–79
- sprintf function, A–5
- sscanf function, A–5
- Start Session window
 - changing language of, 6–92
- strcat function, A–10
- strchr function, A–10
- strcmp function, A–11
 - restrictions of, 2–12
- strcoll function, A–4
- strcpy function, A–11
- strcspn function, A–10
- strextract command, 3–15
- strfmon function, A–4
 - advantages of, 2–16
- strftime function, A–5
 - and langinfo database, 2–14
 - using nl_langinfo function as
 - argument, 2–15
 - using with time and localtime
 - functions, 2–14
- string
 - (*See character string*)
- string comparison, 2–12
- string-handling functions, A–10
- strings file, 3–16
- strlen function, A–12
- strmerge command, 3–15
- strncat function, A–10
- strncmp function, A–11
- strncpy function, A–11
- strpbrk function, A–10
- strptime function, A–5
- strrchr function, A–10
- strstr function, A–10
- strtod function, A–7
- strtok function, A–12
- strtol function, A–7
- strtoul function, A–7
- stty command
 - enhancements for Asian
 - languages, 6–13
 - options for phrase databases,
 - 6–51, 6–56
- stty utility
 - enabling command-line editing,
 - 6–17

- enabling the Japanese input method, 6-19
- enhancements for local languages, 6-13
- local-language enhancements for
 - codeset conversion, 6-15
 - terminal line discipline, 6-13
- substitution fields for NLSPATH setting, 3-30
- swprintf function, A-6
- swscanf function, A-6
- System V Multi-National Language Supplement
 - courses library enhancements for the, 4-1
- system option of phrase command, 6-52

T

- tab character
 - coding in message strings, 3-4
- tbl command, 6-82
- tcode option of stty command, 6-16
- tdec option of stty command, 6-14
- terminal drivers
 - user-defined character recognition, 6-42
- terminal emulation
 - escape sequences used in programs, 4-16
 - features and restrictions for Asian languages, 6-111
- terminal interface features, 6-13
- terminal line discipline
 - displaying current setting for, 6-15
 - restoring default, 6-14
 - setting, 6-13
- terminals
 - converting application codeset to one supported by terminal, 6-15

- text drawing
 - with font sets in X applications, 5-13
- text files
 - printing Asian language print filters that support, 6-61
 - viewing and converting with CDA commands, 6-112
- thousands separator
 - defining
 - for monetary values, 7-17e
 - for numeric values, 7-20e
 - determining with localeconv function, 2-16
 - variation for, 2-13
- time format
 - defining in locale source file, 7-21e
- time function
 - using with strftime function, 2-14
- time values, formatting, 2-14, A-5
- tolower function, A-3
- toupper function, A-3
- towctrans function, A-3
- towlower function, A-3
 - advantages of, 2-11
- towupper function, A-3
 - advantages of, 2-11
- trans command, 3-18
- translation
 - (*See also message catalogs*)
 - requirements for messages, 3-12
- trigraph sequences supported by C language compiler, 2-10
- tty driver
 - enhancements for local languages, 6-13

U

- UCS-4 codeset, 2-4
- UDC databases, 6-23
 - default path to, 6-26
 - font files for, 6-42

- font renderer for, 6–91
- setting default locations of, 6–58
- support files for, 6–42
- udc entry in cp_dirs file, 6–59
- udc option of fontconverter
 - command, 6–46
- ungetc function, A–9
- ungetch function, 4–13
- ungetwc function, A–9
- ungetwch function, 4–13
- Unicode standard, 2–4
- Universal Character Set (UCS), 2–4
- uppercase characters
 - testing for, 2–10
- user names
 - multibyte characters in, 6–86
- user option of phrase command, 6–52
- user-defined characters
 - (See also *cedit* command, *UDC* databases)
 - attributes of, 6–25
 - conversion from ULTRIX, 6–26
 - creating, 6–25
 - classes for, 6–29
 - codeset values for, 6–29
 - font glyphs for, 6–30
 - input key sequences for, 6–29
 - names for, 6–29
 - selecting font size for, 6–31
 - deleting, 6–29
 - languages supported for, 6–26
 - printer settings that SoftODL uses, 6–63
 - scaling fonts for, 6–30

V

- vertical writing
 - restriction for CDA converter
 - output, 6–114n
- vfprintf function, A–5
- vwprintf function, A–6
- vi command, 6–85

- vprintf function, A–5
- vsprintf function, A–5
- vswprintf function, A–6
- vw_printw function, 4–15
- vw_scanw function, 4–14
- vwprintf function, A–6
- vwprintw function, 4–15
- vwscanw function, 4–14

W

- w option of fontconverter
 - command, 6–46
- wadd_wch function, 4–2
- wadd_wchnstr function, 4–4
- wadd_wchstr function, 4–4
- waddnwstr function, 4–5
- waddwch function, 4–2
- waddwchnstr function, 4–4
- waddwchstr macro, 4–4
- waddwstr macro, 4–5
- wscat function, A–10
- wcschr function, A–10
- wscmp function, A–11
 - restrictions of, 2–12
- wscoll function, A–4
 - advantages of, 2–12
- wscpy function, A–11
- wscspn function, A–10
- wcsftime function, A–5
- wcslen function, A–12
- wcsncat function, A–10
- wcsncmp function, A–11
- wcsncpy function, A–11
- wcsprbk function, A–10
- wcsrchr function, A–10
- wcsstr function, A–11
- wctod function, A–7
- wctok function, A–12
- wctol function, A–7
- wctombs function, A–7
 - writing a method for, 7–45
- wctoul function, A–7
- wcswcs function, A–10

wcswidth function, A-12
 writing a method for, 7-51

wcsxfrm function
 advantages of, 2-12

wctomb function, A-7
 writing a method for, 7-48

wctrans function, A-3

wctype function, A-2

wcwidth function, A-12
 writing a method for, 7-53

wecho_wchar function, 4-2

wechowchar macro, 4-2

weekday names
 defining in locale source file,
 7-21e

werase option of stty command, 6-17

wget_wch function, 4-13

wget_wstr function, 4-12

wgetch function, 4-13

wgetn_wstr function, 4-12

wgetnwstr function, 4-12

wgetwch function, 4-13

wgetwstr macro, 4-12

wide characters, 1-5
 compared to multibyte
 characters, 2-8
 default size of, 2-8

wide-character string, 1-6

win_wch function, 4-9

win_wchnstr function, 4-10

win_wchstr function, 4-10

winnwstr function, 4-11

wins_nwstr function, 4-7

wins_wch function, 4-3

wins_wstr function, 4-7

winsnwstr function, 4-7

winswch function, 4-3

winswstr macro, 4-7

winwch function, 4-9

winwchnstr function, 4-10

winwchstr macro, 4-10

winwstr macro, 4-11

wmemchr function, A-12

wmemcmp function, A-12

wmemcpy function, A-12

wmemmove function, A-12

wmemset function, A-12

words
 deleting on command line, 6-16

Worldwide Portability Interfaces
 (WPI), A-1
 case conversion functions, A-3
 character classification
 functions, A-1
 character collation functions,
 A-4
 functions for codeset
 conversion, A-13
 functions that format date and
 time values, A-5
 functions that retrieve langinfo
 data, A-4
 input/output functions, A-9
 locale announcement function,
 A-1
 numeric conversion functions,
 A-7
 printing functions, A-5
 scanning functions, A-5
 string-handling functions, A-10
 wchar/multibyte conversion, A-7

WPI
 (*See Worldwide Portability
 Interfaces (WPI)*)

wprintf function, A-6

wprintw function, 4-15

wscanf function, A-6

wscanw function, 4-14

X

X applications
 creating UDC fonts for, 6-43
 handling messages in, 3-1
 text translation issues, 3-12
 use of multibyte PostScript
 fonts, 6-89

X display manager

- (*See xdm*)
- X libraries
 - handling text for interclient communication, 5-18
 - using internationalization features in, 5-1
 - using with input methods, 5-37
- X server
 - tuning cache parameters in configuration file, 6-87
 - formula for, 6-89
- X Toolkit Intrinsics
(*See Xt Library*)
- X11R6
(*See X libraries*)
- XBaseFontNameListOfFontSet function, 5-11
- XCloseIM function, 5-22, 5-23
- XCloseOM function, 5-16t
- XCreateFontSet function, 5-11
- XCreateIC function, 5-26
 - conditions for failure of, 5-28
- XCreateOC function, 5-16t
- .Xdefaults file
 - changing defaults for language and keyboard, 6-11
- XDefaultString function, 5-18
- XDestroyOC function, 5-16t
- XDisplayOfIM function, 5-22
- XDisplayOfOM function, 5-16t
- xdm
 - changing language for login prompts, 6-92
- XDm library, 5-3
- XDrawImageString function, 5-14
- XDrawImageString16 function, 5-14
- XDrawString function, 5-14
- XDrawString16 function, 5-14
- XDrawText function, 5-14
- XDrawText16 function, 5-14
- XExtentsOfFontSet function, 5-13
- XFillRectangle function, 5-15
- XFilterEvent function, 5-33
 - called by XtDispatchEvent function, 5-34
- XFontSet object, 5-8
- XFontSet structure, 5-10
- XFontSetExtents structure, 5-13
- XFontsOfFontSet function, 5-11
- XFontStruct structure, 5-10
- XFreeFontSet function, 5-11
- XGetICValues function, 5-29
 - XNFilterEvents argument, 5-33
- XGetIMValues function, 5-24, 5-25
- XGetOCValues function, 5-16t
- XGetOMValues function, 5-16t
- XIC object, 5-8, 5-21
 - creating and using, 5-26
 - destroying, 5-26
 - managing, 5-29
 - registering preediting callbacks for, 5-28
 - specifying attributes for, 5-27
 - XNClientWindow, 5-28
 - XNInputStyle, 5-28
- XIM object, 5-8, 5-21
 - closing if IM server fails, 5-36
 - opening and closing, 5-21
- ximdemo online application, 5-7
- XIMOfIC function, 5-29
- XLocaleOfFontSet function, 5-11
- XLocaleOfIM function, 5-22
- XLocaleOfOM function, 5-16t
- XLookupString function, 5-36
- xlsfonts command, 6-94
- Xm library, 5-3
- XmbDrawImageString function, 5-14, 5-15
- XmbDrawString function, 5-14
- XmbDrawText function, 5-14
- XmbLookupString function, 5-34, 5-36
- XmbResetIC function, 5-29
- XmbSetWMProperties function, 5-18, 5-20
- XmbTextEscapement function, 5-13, 5-16

XmbTextExtents function, 5-13
 XmbTextListToTextProperty
 function, 5-18
 XmbTextPerCharExtents function,
 5-13
 XmbTextPropertyToTextList
 function, 5-18
 XMODIFIERS environment
 variable, 5-8
 XmStringCreate function, 5-6
 XmStringCreateLocalized
 function, 5-6
 XmText widget, 5-5
 XmTextField widget, 5-5
 XNDestroyCallback resource, 5-36
 XNQueryInputStyle function, 5-24
 XOC object, 5-8, 5-16
 XOM object, 5-8, 5-16
 XOMOfOC function, 5-16t
 XOpenIM function, 5-21, 5-23
 conditions for failure of, 5-22
 XOpenOM function, 5-16t
 xpg4demo online application, 2-1
 XResourceManagerString
 function, 5-20
 XrmDatabase component, 5-9
 XrmGetFileDatabase function, 5-20
 XrmGetStringDatabase function,
 5-20
 XrmLocaleOfDatabase function,
 5-20
 XrmPutFileDatabase function, 5-20
 XrmPutLineResource function, 5-20
 XSelectInput function, 5-33
 Xservers configuration file
 tuning font cache in, 6-88
 xset command, 6-45
 XSetICFocus function, 5-29, 5-35
 XSetICValues function, 5-29
 XSetIMValues function, 5-36
 XSetLocaleModifiers function,
 5-2, 5-8
 XSetOCValues function, 5-16t
 XSetOMValues function, 5-16t

XSupportsLocale function, 5-2, 5-8
 Xt Library
 internationalization features,
 5-2
 codesets, 5-3
 font sets, 5-3
 input methods, 5-3
 setting locale, 5-2
 XtAppInitialize function, 5-2
 XtDispatchEvent function, 5-3, 5-34
 XtDisplayInitialize function, 5-2
 xterm windows
 support for Asian-language
 input methods, 6-9n
 XtInitialize function, 5-2
 XtOpenDisplay function, 5-2
 XtSetLanguageProc function, 5-2
 XUnsetICFocus function, 5-29, 5-35
 XVaCreateNestedList function, 5-27
 XwcDrawImageString function,
 5-14
 XwcDrawString function, 5-14
 XwcDrawText function, 5-14
 XwcFreeStringList function, 5-18
 XwcLookupString function, 5-34
 XwcResetIC function, 5-29
 XwcTextEscapement function, 5-13
 XwcTextExtents function, 5-13
 XwcTextListToTextProperty
 function, 5-18
 XwcTextPerCharExtents function,
 5-13
 XwcTextPropertyToTextList
 function, 5-18

Y

ya symbol in /etc/printcap file, 6-63
 yd symbol in /etc/printcap file, 6-63
 yes responses
 defining in locale, 7-15
 yesexpr keyword, 7-16
 yesstr keyword, 7-16
 yj symbol in /etc/printcap file, 6-63

yp symbol in /etc/printcap file, 6-63
ys symbol in /etc/printcap file, 6-63

yt symbol in /etc/printcap file, 6-63

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—	Local Digital subsidiary or approved distributor
Internal (submit an Internal Software Order Form, EN-01740-07)	—	SSB Order Processing – NQO/V19 <i>or</i> U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

Reader's Comments

Digital UNIX

Writing Software for the International Market
AA-Q0R4C-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: readers_comment@zk3.dec.com
- Fax: (603) 881-0120, Attn: UEG Publications, ZK03-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Usability (ability to access information quickly)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name, title, department _____

Mailing address _____

Electronic mail _____

Telephone _____

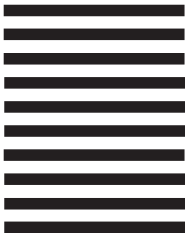
Date _____

----- Do Not Cut or Tear – Fold Here and Tape -----

digitalTM



NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UEG PUBLICATIONS MANAGER
ZK03-3/Y32
110 SPIT BROOK RD
NASHUA NH 03062-9987



----- Do Not Cut or Tear – Fold Here -----

Cut on
Dotted
Line