HEWLETT® PACKARD

HP 9000
Computers

Native Language Support: User's Guide

Native Language Support: User's Guide

HP 9000 Computers

HP Part No. B2355-90036
Printed in USA August 1992

Third Edition
E0892

Legal Notices

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Restricted Rights Legend.. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in sub-paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

UNIX® is a registered trademark of UNIX System Laboratories, Inc. in the USA and other countries.

Printing History

New editions of this manual will incorporate all material updated since the previous edition.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

May 1990. First Edition

January 1991. Second Edition

- This edition updates the previous document called *Native Language Support Concepts and Tutorials*. This edition includes corrections and additions for the HP-UX 8.0 release. Major changes and additions include: An overview of Native Language Support; improved organization to improve access to information in a more logical sequence; addition of information on supported codesets; expanded information on localizing international software; added illustrations to simplify complex topics; and added depth and coverage on may topics.

August 1992. Third Edition

- This edition documents the addition to NLS of World Portability Interface (WPI), and its wide-character encoding. This is a standardized approach (XPG4) and is the preferred approach.

Contents

6. Developing International Software

Figures

Tables

1

Overview

UNIX Background

The UNIX operating system was originally designed to be used with terminals that generated seven bit ASCII characters. The primary users were English-speaking Americans, so internationalizing the operating system was not even considered. A number of years have passed since the creation of the UNIX operating system; it grew to be used industry-wide and world-wide as an application programming environment.

There became a need to handle characters from coded character sets other than the standard ASCII codeset, some of which contain thousands of characters. There became a need to provide interfaces that allow users to interact with applications in their own language, rather than forcing them to use English error messages and instructions. Native Language Support (NLS) was created to do this in a manner which preserves the traditional UNIX operating system, but is language independent.

The NLS Concept

NLS enhances the traditional UNIX operating system in a way that benefits both the users and the application developers. It enables the design of *flexible* applications that can support a variety of language environments. NLS consists of a number of tools that help preserve the integrity of data across different codesets, provide interfaces in the local language, and represent local customs in software. Much existing software is designed for use only in the country or locale for which it was originally written. This means the programmer must redesign and recompile the software for every local language and local environment.

However, the features of Hewlett-Packard's **NLS** enable the application designer or programmer to create applications for an end-user's needs regardless of the local language. NLS addresses an application's internal functions (such as sorting and character handling) and the user interface (which includes displayed messages, user inputs, and currency formats). An internationalized program is really many programs in one, in that it uses and is supported by tools that separate language-dependent features from the main program logic.

The NLS Conceptual Model

The NLS concept is a simple one. NLS allows one discrete program to "speak" in a variety of languages. NLS consists of an extensive set of tools and routines. All of its components support an "extended multi-language application" consisting of three parts:

- *A Language-Independent Program*— The program displays messages in the user's native language by using language-dependent features that are not a fixed part of the program. The program does this in either of two ways:
  - it processes the language-dependent data in a *codeset-sensitive* way; we could call this the traditional NLS approach. Or,
  - the program uses the Worldwide Portability Interface (WPI) to process the language-dependent data. WPI's wide character encoding enables the program to deal with data in a way that is *codeset non-sensitive*.
- *Message Catalogs* — There are no hard-coded messages in the source code. Instead messages to the user (such as prompts and error messages) are stored in external message catalogs with a version for each supported language. For instance, instead of the source code containing the statement `printf ("display this string")`, the program calls a routine that opens a specific message catalog containing the language-specific equivalent.
- *Language Tables*— This component of NLS contains language-specific information and conventions unique to a particular locale (such as collation sequence definitions and monetary conventions). Programs consult a specific language table at run time according to the setting of the user's environmental variables.

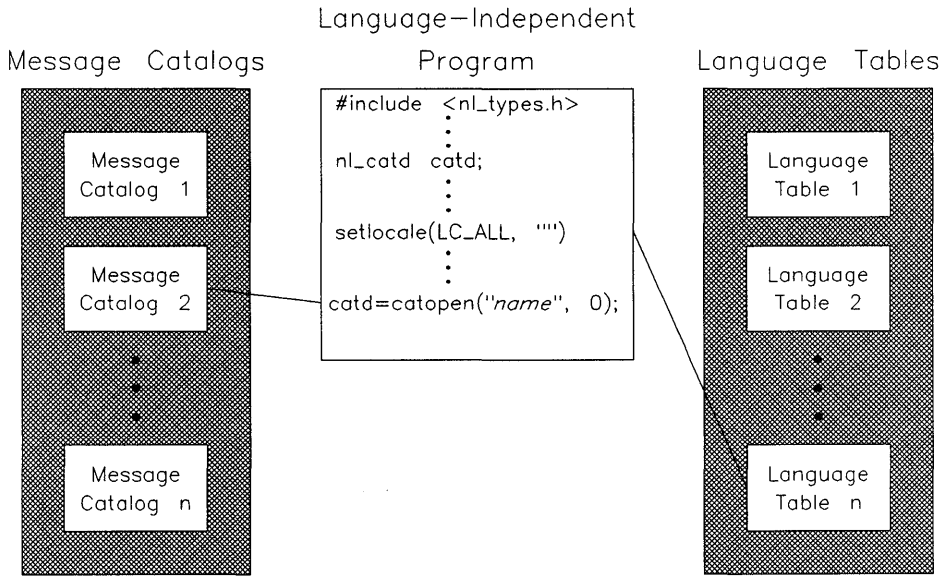

Figure 1-1. NLS Conceptual Model

Using this Manual

This manual is for people who are using, writing, or localizing programs for international use and who will need to make use of the various elements of Native Language Support (NLS).

Specific sections of this manual were written at a technical level appropriate to general users, system administrators, NLS coordinators, or applications programmers.

- **General Users** should read Chapters 2 and 3.
- **System Administrators** should read Chapter 4 and 5.
- **Applications Programmers** should read Chapter 5, 6, 7 and 8.

For further details, refer to the Appendices. For example, Appendix B provides an example of internationalized programming. All of the NLS commands and subroutines discussed in this manual are referenced in Appendix C and in the Index.

Table 1-1. Finding Information in the NLS User's Guide

To find this information ...	Please see ...
Overview: This chapter provides a conceptual overview of NLS and the internationalized application. This chapter also explains how the manual is structured. This chapter explains the typographical conventions used in this manual and identifies other related manuals.	Chapter 1
Introduction to NLS(for the general user): This chapter presents a basic description of the scope of Native Language Support, localization, and internationalization, including general aspects of character set handling, local conventions and messages.	Chapter 2
Using International Software(for the general user): This chapter shows how to run a localized application including terminal configuration, environment setup, and selection of the language.	Chapter 3
Administering International Software (for the system administrator): This chapter identifies the HP-UX directories and files, and explains how to set up the user environment, to install message catalogs and optional locales, and to configure terminals and peripherals.	Chapter 4
Localizing International Software (for the system administrator and the localizer): This chapter explains the details of localization for special user requirements, localizing message catalogs, and creating specialized locales.	Chapter 5
Developing International Software(for the programmer): This chapter describes the initialization process, character and string processing, and gives a brief introduction to setting up the message interface. This chapter focuses on the Worldwide Portability Interface (WPI) approach to internationalizing software.	Chapter 6
The Message Catalog System(for the programmer): This chapter explains how to call message routines, create the message text source file, and generate the message catalog.	Chapter 7
Advanced NLS Topics (for the programmer): This chapter explains about the NLS character- and string-processing tools, about processing non-Latin character input/output, and about special treatment of locales and message catalogs.	Chapter 8

Table 1-1.
Finding Information in the NLS User's Guide (continued)

To find this information ...	Please see ...
Special Topics for HP's 16-bit Interfaces: This appendix provides the information necessary to maintain existing software that was internationalized with HP's 16-bit Interface.	Appendix A
Examples of Internationalized Software: Character processing, collation, monetary formatting, messaging, and date/time format.	Appendix B
NLS References: An alphabetic listing of *HP-UX Reference* locations for all NLS commands and routines. Also lists the current NLS library routines in alphabetic order, along with their associated entry in the *HP-UX Reference*, and a description of the routine's purpose.	Appendix C
Previous Usage: Tables of current and obsolete NLS commands and routines.	Appendix D
Languages and Codesets: A listing of the native languages that are supported by HP codesets.	Appendix E
LC_COLLATE Example: An example set-up of the `LC_COLLATE` category.	Appendix F
Definitions: Major words and concepts used in this manual.	Glossary

Typographical Conventions in This Manual

Italics

This typography indicates manual names and references to manual pages in the *HP-UX Reference*. Italics are also used for symbolic items either typed by the user or displayed by the system, as discussed below under *Variable name*. Italics are occasionally used to *emphasize* or *stress* words.

New Terms

This typography is used when an important new term is introduced.

`Computer literal`

This typography indicates literal input to, or output from, the computer. Type the characters in this font exactly as they appear on the page. For example:

```
findstr prog.c > prog.str
```

Variable name

This typography indicates that you need to "fill in the blank" in a command line with your own word or data. This font is used for names of variables and symbolic names. For example:

`cat` *file_name*

means you type `cat` and substitute the appropriate *file_name* to complete the command line.

keycap

This typography indicates a key on your keyboard. For example, Return means to press the "Return" key. When prefixed by Shift, CTRL, or Extend char, press both keys simultaneously. For example:

CTRL-C

means you press the CTRL key and continue to hold it while you press the C key.

Related HP-UX Manuals

This manual may be used in conjunction with other HP-UX documentation. References to these manuals are included, where appropriate, in the text. Also refer to Appendix C for *HP-UX Reference* entries to NLS functionality.

- The *HP-UX Reference* contains the syntactic and semantic details of all commands and application programs, system calls, subroutines, special files, file formats, miscellaneous facilities, and maintenance procedures available on the HP 9000 HP-UX Operating System. Unless otherwise stated, all references in this manual such as "see *langinfo*(3C) for more details", refer to entries in the *HP-UX Reference* manual.
- The *HP-UX Portability Guide* presents guidelines for writing portable code, code that is easy to transfer between HP-UX systems or from HP-UX to another system or vice versa. It also describes techniques for porting programs in the HP-UX environment. Related to portability, this manual describes how to call routines written in other programming languages across HP-UX, for example, calling C routines from FORTRAN.
- The *HP-UX System Administration Tasks* manual provides step-by-step instructions for installing and updating the HP-UX Operating System software and for installing the NLS languages, if they are optional for your system. It also explains procedures for system boot and login, and contains guidance for implementing administrative tasks.
- The *Terminal Control User's Guide* contains valuable guidance for setting up your terminal and configuring the softkey definitions.
- *Finding HP-UX Information* provides descriptions, and part numbers for the Series 300, 400, 700, and 800 HP-UX manuals. (With the 9.0 release of HP-UX, this reference is available online in HP-VUE's Helpview Help under "HP-UX 9.0 Operating System Help.")
- *HP Visual Users Environment User's Guide* contains a task reference for general users, as well as advanced users and system administrators. This manual replaces both the `HP VUE User's Guide` and `HP VUE System Administration Manual` for previous versions of HP VUE.
- Native Language Input/Output

- The *Japanese NLIO Manual* explains the overall usage of Japanese Native Language I/O (NLIO). It includes details on both system administration and operational tasks. This manual is a combined manual of the former NLIO Access User's Guide, NLIO System Administrator's Guide and the Japanese Input Method Guide. This manual is written in Japanese. (B2200-90019)
- The *Native Language I/O Access User's Guide* explains how to use Native Language I/O (NLIO). An overview of the NLIO theory and structure as well as code mapping introduces the user to NLIO. The manual explains NLIO commands in detail and provides information for programmers. Separate sections lead the user through NLIO input methods for each of the Asian languages. This manual is written in each local language.
  B2204-90011 (Korean version)
  B2212-90011 (Simplified Chinese version)
  B2208-90011 (Traditional Chinese version)
- The *Native Language I/O System Administrator's Guide* tells system administrators how to install the Native Language I/O (NLIO) subsystem on the HP 9000 Series 400, Series 700 and Series 800 computers. Section 1 explains how to install Asian language-dependent printers and terminals. Host computer hardware and software requirements are also covered. Simplified Chinese, Traditional Chinese and Korean peripherals are covered in separate sections. The following additional fonts are covered in this manual: Traditional Chinese and Korean simplex fonts and Korean designer fonts. This manual is written in each local language.
  B2204-90002 (Korean version)
  B2212-90002 (Simplified Chinese version)
  B2208-90022 (Traditional Chinese version)
- The *NLIO Input Method Guides* explains details on using Native Language I/O (NLIO) to input Asian language on to HP 9000 computers.
  B2212-90003 (Simplified Chinese version)
  92553-90001 (Traditional Chinese version)

2

Introduction to NLS

Overview of Software Internationalization

HP-UX users speak a variety of languages and observe many different cultural practices. Local-language, processing capability is becoming a high priority with the kinds of software products in use throughout the world. For this reason, users need software which will easily accommodate local conventions.

To do so effectively, software products must preserve the integrity of data, correctly handle the written conventions of a variety of languages, and provide a message interface in the user's language. In addition, they must be versatile in handling a variety of local data formatting conventions.

The two processes in the NLS approach that enhance software for international use are: internationalizing, localizing.

- **internationalizing** software includes supporting the letters and symbols required to read and write the user's language, processing characters and text according to the rules of the user's language, providing the scheme to process translated messages and prompts, and changing functions and conventions to comply with local requirements. Such internationalization must be accomplished with a minimum of change to existing program code.
- **localizing** adapts the software to a particular locale, including the translation of messages and the use of appropriate language tables on the system.

In general, then, the main requirements which Hewlett-Packard has addressed to facilitate the international use of software are:

- Consistent preservation of data integrity
- Proper handling of characters
- Effective communication interface in the user's language
- Proper representation of local customs in the software

HP Native Language Support provides an extensive set of tools and routines for implementing language-independent software. Software can, with relatively minor modifications, use language-dependent processing information which is stored externally to the program code. At run time, the application accesses the processing information appropriate for the language currently specified. There are some unique advantages to this NLS strategy:

- Software is not duplicated in different versions for different languages. This makes it easier to update and maintain the program.
- Because all language-dependent processing information is kept external to the program source, programmers and localizers need not modify the program source when modifying messages. The chance of "bugs" being introduced into the software as a result of this process is eliminated.
- With the addition of the Worldwide Portability Interface (WPI) capability, which is a standardized approach (XPG4) and the *preferred* approach, especially for new development, the program no longer needs to deal with data in a language-sensitive way. This makes the programming easier, and provides for consistent treatment across languages because of the wide character encoding.
- Since software can be localized more easily, the time and expense required by localization is relatively low.
- Many users could simultaneously share the same copy of a program, with each one potentially using a different language or set of language conventions.

HP has an ongoing test process to ensure compliance with applicable standards of XPG3, POSIX, and ANSI-C.

Components of Internationalizing and Localizing with NLS

NLS provides a number of features to aid international users and programmers:

- It permits users to specify the desired language at run time.
- It allows different users to use different languages on the same system.
- It provides the programmer with the ability to internationalize software.

Internationalization

NLS supports these features by providing language-dependent tables for various locales and by program internationalization and localization. Internationalization involves:

- Designing an application so that it uses a set of NLS routines that retrieve and set the program's linguistic environment according to the user's needs.
- Using NLS tools for copying all hard-coded messages into external message catalogs and for updating the message catalogs.
- Referencing language tables for language and locale sensitive information.
- Replacing some original HP-UX routines in an application with NLS versions of the routines. For example, the routine `ctime` would be replaced with the NLS-enhanced version `strftime` or the WPI version `wcsftime`.
- Using the WPI to enable the application to handle both single and multi-byte character sets.
- Revising algorithms if necessary to preserve data integrity and provide language-sensitive processing.

Localization

The internationalized application then can be adapted for use in a specific linguistic environment through the process of localization. Localization includes:

- Translating the text in the message catalogs into the local language.
- Installing message catalogs and language tables on the system.
- Creating or adapting language tables for locales with special requirements.

- Providing the way to set up the user's environment in the local language, such as special fonts for Asian languages.

Localization activities are supported by translators and system administrators. Through the process of localization all aspects of the user interface are "customized" to the users' requirements.

Internationalization is usually done by the software developer as part of the application development or modification process. Localization can often be facilitated by Localization Centers operated by various Hewlett-Packard Country Product Organizations.

Aspects of NLS Support

There are three aspects of Native Language Support included in HP-UX software:

- Character and text handling
- Local customs and conventions
- Messages and menus

Character and Text Handling

NLS provides the ability to identify and manipulate characters in a variety of ways and to handle language-specific text processing.

Data Integrity

To say that software preserves data integrity means that the software must allow users to input characters required for their native language without corrupting those characters. For example, if a program prompts Müller to enter his name, the program should neither convert the "ü" to another character (such as "O", with the name appearing as MOller) nor should it map the character to an undefined value (such that "M ller" appears).

Character Sets and Encoding

In an HP-UX environment, the default local language character set is 7-bit ASCII (or USASCII). All programs which are not internationalized, or those

that are internationalized but in which the user has not enabled NLS, use this character set. Note, however, that 7-bit ASCII is not sufficient even to span the Latin-based alphabet used in many European languages.

For many Asian languages, character sets can contain several thousand characters. This is more than can be encoded in the single 8-bit number which is the conventional value used to represent character data. For this and other reasons, NLS character handling has the following characteristics:

- The 8th bit of a character byte is never stripped or modified.
- The extra bit provides support for languages that have additional characters, accented vowels, consonants with special forms, and special symbols.
- Multi-byte coded character sets may be used for character sets that contain more than 256 members.

There are many implementations of non-ASCII character sets currently in use. NLS permits users to define their own character sets and character properties. However, Hewlett-Packard has already supported coded character sets that permit the processing of many Eastern and Western European, Middle Eastern, and Asian languages.

Every HP-supported 8-bit coded character set is a superset of ASCII. The HP-supported 8-bit coded character sets for Western European languages are ROMAN8 and the standard ISO 8859-1. Hewlett-Packard also supports ISO 8859-2 and ISO 8859-5 for Eastern European languages, including Cyrillic. Other 8-bit coded character sets are defined for other locales. For a listing, please refer to Appendix E, "Languages and Codesets".

For alphabets of more than 256 characters, such as **Kanji** (Japanese ideographic characters), multi-byte character codes are required. Hewlett-Packard has defined a multi-byte character encoding scheme, HP-15, which uses two bytes (16-bits) to represent a character. Four sets are defined under this scheme, which are used to represent Traditional Chinese, Simplified Chinese, Korean, and Japanese.

In addition, Hewlett-Packard provides support for the EUC character encoding scheme for up to 2 bytes. This scheme is used for data processing and storage. For input and output, Hewlett-Packard uses a multi-byte character encoding scheme called HP-16.

Users may provide support for some non-HP-defined code and character sets by using the `localedef` command. For more information on the `localedef` command, see Chapter 5.

Character Type and Conversion

All sorting, case shifting, and type analysis of characters is done according to the local conventions for the native language selected. While the ROMAN8 and ISO 8859 coded character sets have uppercase and lowercase for most alphabetic characters, some languages discard accents when characters are shifted to uppercase. European French commonly discards accents in uppercase, while Canadian-French does not. If there is no representation of case in the user's language, as is the case in ideographic languages such as Japanese, characters are not shifted at all.

Local Customs and Conventions

Certain aspects of NLS relate to the local customs or conventions of a particular geographic area. These aspects, even when supported by a common character set, can change from region to region. Consequently, number format, currency information, date and time format, case shifting, and collation are set according to the user's local conventions. In NLS, these environmental characteristics collectively designate a "locale".

For instance, although Great Britain, the United States, Canada, Australia, and New Zealand share the English language, aspects of data representation differ according to local customs. Variations are encountered in the following everyday matters:

- Representation of numbers (numeric formatting)
- Representation of currency units (monetary formatting)
- Display of time
- Display of days, weeks, months

Chapter 8 describes the library routines used to handle these local customs.

Numeric Formatting

In the representation of numbers, all the following depend on local customs:

- The "radix" symbol which performs the decimal-indicating function (the period in America).
- The thousands separator (the comma in America) which serves to separate groups of digits.
- The convention for grouping digits (by three's in the `american` locale).

In the United States, a number is represented as follows:

```
2,345.678
```

But when representing the same number in France, the thousands separator is blank, and the radix symbol is a comma:

```
2 345,678
```

Monetary Formatting

Currency units and how they are subdivided vary with region and country. The symbol for a currency unit can change as well as the placement of the symbol. It can precede the numeric value, follow it, or appear within it.

Between the currency conventions used by America and France, the currency symbol string is transposed.

```
US$2,345.77
```

versus

```
2 345,77 FF
```

Display of Time

Computation and proper display of time, including 24-hour vs. 12-hour clocks, must be considered. The HP-UX system clock runs on Coordinated Universal Time (UTC). Corrections to local time zones consist of adding or subtracting whole or fractional hours from UTC. Some regions, instead of using the Western Gregorian calendar system, designate the years by seasonal, astronomical, or historical events. One system which HP supports is the Imperial system used in Japan for numbering years based on the reign of the current emperor.

Display of Days, Weeks, Months

Names for days of the week and months of the year may vary with language, as do rules for abbreviating these. The order of the year, month, and day, as well as the separating delimiters also differ. For example, October 7, 1991 would be represented in America as:

```
10/7/91
```

in Germany, it would be represented as:

```
7.10.1991
```

and in the U.K. as:

```
7/10/91
```

Chapter 6 describes the library routines used to handle these local customs.

Collation

Each language may use its own distinct "collating sequence"—the sequence in which characters or words are ordered by the computer. Some language may even have more than one set of collation rules. The ASCII collation order, which is the default setting for HP-UX, while it is fast, is inadequate even for the accuracy requirements of **american** locale dictionary sorting. Each language may order the characters differently, and certain languages have multiple acceptable orderings.

Chinese is an example in which the ideographic characters can be sorted in order of:

- The numeric value of the character as represented in a computer character set. (HP has implemented this method only.)
- The number of strokes required to represent the character.
- The radical (root) of the character.
- The pronunciation of the character.

Messages

The ability to customize messages for different countries is an important aspect of using NLS. NLS enables you to choose the language for prompts, responses to prompts, and error messages. All of this can be done at run time. And, since messages are kept in catalogs separate from the program code, it is not necessary to recompile the source code when you are using the program in another language.

It is, however, necessary to work closely with your translator to ensure that the semantics of system or program messages are correctly conveyed in the translation. In practice, the syntax of another language may force a change in the sentence structure of a translated message.

For example, an English message for a given command might be interpreted two ways in German.

The original in English is:

```
cannot read at directory
```

("`at`" is an HP-UX directory)

In German, this message could be interpreted as:

```
Kann das Verzeichnis nicht lesen.
```

(Literally: "cannot read the directory", with "`at`" misinterpreted as an untranslatable preposition)

If the meaning of "`at`" is pointed out to the translator in a "cookbook" accompanying the message catalog, the message would be correctly translated as:

```
at Verzeichnis nicht lesbar.
```

(Literally: "'`at`' directory not readable."—the intended meaning.)

Handling messages in message catalogs helps ensure that the messages are accessible for editing, updating, and translating into other languages, as required.

For details on the use of message catalogs, see the section "Localizing Message Catalogs" in Chapter 5.

Language Variations

Comparing Strings and Comparing Characters

The order in which character strings are sorted is language-dependent. In addition, there may be a discrepancy between a character's order within a character set and true lexicographical (dictionary) order. Sorting based on character code does not provide true lexicographical order even in the case of the ASCII character set.

Lexicographical order sorts "a" after "A" and before "B", whereas ASCII-based order sorts "a" after the entire set of uppercase letters. This situation becomes more complex in an internationalized program that is structured to handle many different, coded-character sets.

The following is an example of sorting the same list based on the C sorting method, and based on a German sorting method. The table shows that while "ä" *follows* "b" in the coded character set ordering, it is sorted *before* "b" when sorted according to lexicographical order. This situation makes sorting based on the code of each character inappropriate when internationalizing software. Even *within* the subset of lowercase values, character set order does not coincide with lexicographical order. In addition, characters can no longer be up-shifted or down-shifted as in ASCII by adding or subtracting a fixed offset.

Table 2-1. Sorting Example: C vs. German

Sorted by C rules	Sorted by German rules
Airplane	Airplane
Zebra	äpfel
bird	bird
car	car
äpfel	Zebra

Beyond the ordering of individual characters, some languages designate that certain characters be treated in a special way. For example, in some languages groups of characters are clustered and treated as a single character.

In Spanish "ll" is treated as a single character, and it is sorted after "l" and before "m". Similarly, the "ch" in Spanish is treated as a single character, and it is sorted after "c" but before "d":

Table 2-2. Sorting Example: C vs. Spanish

Sorted by C rules	Sorted by Spanish rules
chaleco	cuna
cuna	chaleco
día	día
llava	loro
loro	llava
maíz	maíz

When sorting strings in some languages, a single character is expanded and treated as if it were really two characters. For example, when sorting strings in German, ß (the "sharp s"), is treated as if it were "ss".

Table 2-3. Sorting Example: C vs German

Sorted by C rules	Sorted by German rules
Rosselenker	Rosselenker
Rostbratwurst	Roßhaar
Roßhaar	Rostbratwurst

In some languages, certain characters such as "-" are ignored when collating strings, and these also need to be taken into account.

- **Data directionality.** This is the spatial order in which data is displayed vs. the order in which it is entered. Data directionality is not the same for all languages. For example, some Middle Eastern languages are read from right to left and may be mixed with insertions in left-to-right European languages. NLS allows for processing of this type of character data. Currently, no special provisions are made for top-to-bottom languages, such as Chinese, which are handled in a left-to-right orientation.

- **Multi-byte characters.** Finally, character handling also involves the correct parsing of multi-byte character streams and the interpretation of multi-byte characters. Multi-byte character streams may contain both single-byte and multi-byte characters. To process this data, each byte must be identified as either a single-byte character or as part of a multi-byte character. The details of these and other aspects of character handling are discussed in Appendix A.

Regular Expressions

HP-UX allows the specification of arbitrary character strings through the use of regular expressions. For further details on their use, see the section, "Regular Expressions", in *The Ultimate Guide to the vi and ex Text Editors*. The syntax of regular expressions has been extended in HP-UX to allow use with other character sets.

Here is one example of an internationalized regular expression:

```
h[[=e=]]lp
```

This matches the word "help" spelled with any variation of the letter "e" (for example, e, é, ë, è).

The existing syntax of a range expression (e.g., "[a-z]") is not changed. However, its meaning has been extended to mean "match any collating element which falls between the two given collating elements based on the current locale's `LC_COLLATE` collation sequence."

For multi-byte languages, the support in regular expressions is not as extensive. For example, multi-byte characters are allowed as single character elements in expressions, and they can be used in character ranges. However, the inverse of a range ("`[^a..z]`") is not allowed with multi-byte characters in general. This is due to restrictions in the way the codesets are implemented. Moreover, some new features are not allowed with multi-byte codesets simply because they have no application to Asian languages.

3

Using International Software

Read this chapter if you are:
- ▷ A general user of internationalized commands and software.

This chapter covers information and tasks you will need to deal with in order to use internationalized software successfully. The information and the tasks are minimal because, in most situations, you will be receiving help from your system administrator for the following tasks:

- Advising on optimal use of NLS features
- Ordering NLS software
- Installing and updating the operating system
- Configuring the system
- Installing and initializing additional software
- Maintaining system software

NLS Environmental Variables

As a user, you interact with NLS through the selection of a specific linguistic *environment*. While in a non-internationalized program this environment is static (defined by the programmer who developed the software), in an internationalized program the linguistic environment is a variable (or a set of variables) that you, the user, can select or change. Your environment is defined by the environment variables shown in Table 3-1. By setting these variables (or using the system default values), you indicate to the system your unique requirements for various aspects of NLS capabilities.

Table 3-1. NLS Environment Variables

`LANG`	Specifies native language, local customs, and coded character set and messages. `LANG` provides default values for the below variables if they are not explicitly defined.
3 `LC_ALL`	Specifies the behavior of all categories including: ■ regular expressions ■ NLS string collation functions ■ character classification and conversion functions ■ all routines which process multibyte characters ■ functions which handle monetary values, ■ the radix character in formatted input/output functions and the string conversion functions ■ numeric values found in the `localeconv` structure ■ time conversion functions
`LC_COLLATE`	Specifies string collation.
`LC_CTYPE`	Specifies character classification and case conversion.
`LC_MESSAGES`	Specifies affirmative and negative response expressions.
`LC_MONETARY`	Specifies currency symbol and monetary value format.
`LC_NUMERIC`	Specifies decimal number format.
`LC_TIME`	Specifies date and time format and the names of days and months.
`NLSPATH`	Specifies search path for message catalogs.
`LANGOPTS`	Specifies data directionality for right-to-left languages.

For additional information on the NLS environment variables and their use, see *environ*(5) in the *HP-UX Reference*.

Default Values

Local system default values for the NLS environment variables are ordinarily determined and set by your system administrator. When you log in, your environment will be set to the system default values unless you specifically set your environmental variables to something different in your `.profile` or `.login` file.

To see the setting of your NLS environment variables type:

```
env
```

If none of the NLS environment variables is set or if you are not sure what NLS environment you need, consult with your system administrator to determine the appropriate settings for your locale.

Setting your Environment

If the local system default values are not satisfactory, you can use the `locale` you need by setting the environment variables appropriately.

3

For example, if you need a French locale, run the Bourne or Korn shell commands:

```
LANG=french ; export LANG
```

This is equivalent to the C shell command:

```
setenv LANG french
```

Add these commands to your `.profile` or `.login` file so that your preferred environment will be set when you log in.

If you will be running applications that need an NLS environment different from the system default and different from your individual environment, create a shell script that sets the environment variables as needed for the application.

For example, to run the command `prog` in a special NLS environment, the following `sh` script could be used:

```
: # run prog
# set special NLS environment for prog
LANG=english ; export LANG
LC_TIME=italian ; export LC_TIME
LC_MONETARY=german ; export LC_MONETARY
LC_NUMERIC=french ; export LC_NUMERIC
# run prog
prog file1 file2
```

Such a script could be installed by your system administrator in `/usr/bin` and used to invoke your program as well as saving time in setting a special NLS environment.

Setting Your Terminal

First, check your terminal to ensure that it is configured for transmitting and receiving 8-bit data. This involves setting your terminal to 8 data bits and no parity. Additionally, if you are using terminal emulator software on a workstation or PC, it may be necessary to choose the proper font for the character set you wish to work in.

To use international software, your terminal should also be set so that data is not corrupted by system software that might otherwise attempt to interpret the eighth bit of a byte. This bit is needed as part of the character code. To disable such interpretation, run:

```
stty -istrip -parity
```

It is generally convenient to add this command to your `.profile` or `.login` file.

Reference Information for Internationalized Commands

For any command you intend to use, consult the online man pages or the appropriate page in the *HP-UX Reference* to determine the extent to which it has been internationalized. The section "EXTERNAL INFLUENCES, Environment Variables" in each man page indicates NLS environment variables that affect the behavior of a command. For example, to see how `LC_TIME` affects the `date` command, run:

```
man date
```

Internationalized Messages

A command that has been internationalized for messages will have, in the *HP-UX Reference* section "EXTERNAL INFLUENCES, Environment Variables," a comment such as "LANG determines the language in which messages are displayed." Such a command, however, will not necessarily have message catalogs installed on the system for all languages or even for any language other than for a default locale.

When such a command is run, current locale messages will be displayed if they are available. Otherwise, default locale messages will be displayed. The command will, however, perform correctly for the current locale.

For example, `sort` will correctly sort data in all supported locales. Messages issued by `sort` will be in the `C` locale (the default locale for HP-UX commands) unless localized message catalogs have been provided and installed on the system.

See Chapter 5 for more information on localizing message catalogs.

Using Internationalized Commands

To see what locales are installed on your system run `nlsinfo`. Then set `LANG` to one of the installed locales and run `date`.

You should get a result with the format and naming conventions of the locale specified by `LANG`.

To test this further, try:

`cat` *file*

where *file* is non-existent. If there is a localized message catalog for `cat` you should get the `cannot open` message in the locale specified by `LANG`. If not, you will get the message in the `C` locale.

If you do not get the expected results, check with your system administrator to verify that the required language-specific files are properly installed on the system. Otherwise, you should now be able to use internationalized commands without further special action.

Note

If you find that `date` works according to the current setting of `LANG`, but `cat` does not, this means a message catalog for `cat` has not been installed on your system for the selected language. The `date` command relies on language tables and library routines, rather than message catalogs, so not having a message catalog installed for the selected language does not affect it.

4

Administering International Software

Read this chapter if you are:

- A System Administrator who supports the use or development of NLS software.

This chapter covers information you will need to know and tasks you will need to perform in order to ensure that users on your systems are able to use NLS features successfully.

Finding NLS Files

The NLS information used by HP-UX commands and libraries is located in the following directories and files:

Table 4-1. NLS Directories and Files

Directory/Files	Type of NLS Information
`/usr/lib/nls`	This is the directory under which NLS information is located.
`/usr/lib/nls/config`	This readable ASCII file identifies currently installed locales, including user-defined locales created by **localedef**. It contains locale names and their corresponding locale-ID numbers.
`/usr/lib/nls/`*locale*	This directory is present for each installed locale.
`/usr/lib/nls/`*locale*`/`*locale.inf*	These files contain locale-dependent processing information.
`/usr/lib/nls/locale/locale.ext`	These files contain locale-dependent processing information used by the WPI routines.
`/usr/lib/nls/`*locale*`/*.cat`	These are the localized message catalog files.

In the most general case, *locale* can be of the form: *language_territory.codeset*. Either of the extensions *_territory* or *.codeset* may be omitted if not applicable, and in general, both are omitted. If a locale has *_territory* or *.codeset* extensions, there is a corresponding subdirectory for each extension. For example, if `/usr/lib/nls/config` has entries:

```
    ⋮
german.iso88591
german_swiss
german_swiss.iso88591
    ⋮
japanese
japanese.euc
    ⋮
```

Then, you should expect to find the following directories:

```
/usr/lib/nls/german/iso88591
/usr/lib/nls/german/swiss
/usr/lib/nls/german/swiss/iso88591
/usr/lib/nls/japanese
/usr/lib/nls/japanese/euc
```


The Default User Environment

The NLS environment variables should have system default values appropriate to the local user community. These values would ordinarily be determined by the system administrator. You should include commands in `/etc/profile` and `/etc/csh.login` that will set the user's environment variables to these default values. Note that HP-UX does not set these variables, by default.

Installing Message Catalogs

Localized message catalogs should be installed in the appropriate location. Message catalogs for HP-UX commands and libraries should be located in `/usr/lib/nls/`*locale*. If your system has territory or codeset specific locales you will need to check additional directories. See discussion in "Finding NLS Files" above.

Message catalogs for other applications can be put in any location that can be referenced by the conventions of `catopen` and `NLSPATH`. The location and naming of local message catalogs will generally be made by you in consultation with the system administrator. This location and naming may require a change to the system default value of `NLSPATH`. If it does, the system administrator will determine the new value and make the required change to the `NLSPATH` setting in `/etc/profile` and `/etc/csh.login`. All users should be notified of this change.

Installing Optional Locales

The procedure for installing additional software such as an NLS locale is explained in detail in the section "Updating HP-UX" of *HP-UX System Administration Tasks*.

Normally, HP-UX is shipped with all supported locales. Not all coded character sets are supported on all peripherals, however, so peripherals which support the desired character set must also be obtained. After a locale is installed, the NLS locale-specific information can be used by any application program requesting it.

Peripheral Configuration

When you purchase peripherals for use in a non-ASCII or multiple language environment, you should consider the coded character sets that your peripherals will need to support. Hewlett-Packard provides printers, plotters, keyboards and terminals which support HP single- and multi-byte coded character sets, as well other standards (such as the ISO 8859-1 coded character set for Europe). In some cases, you may need special software in order to operate these peripherals, such as the NLIO system for Asian peripherals.

5

Localizing International Software

Read this chapter if you are:

- ▷ A System Administrator who supports the use or development of NLS software.
- ▷ A programmer of internationalized software
- ▷ A localizer who translates message catalog entries into the native language.

This chapter:

- Covers information and tasks for localizing commands that have been internationalized.
- Helps you determine local NLS needs.
- Describes how to create a language definition for a new locale through the creation or modification of a `localedef` script.
- Explains how to install a new language definition on your system.

Localizing the User Environment

HP-UX does not automatically set NLS environment variables. HP-UX commands, when run without NLS environment variables set, default to the "C" locale. If this is the desired system default locale, no changes for the user environment are needed.

To provide a different system default locale, specify the desired default values for the NLS environment variables:

- `LANG`
- `LC_`*categories*
- `NLSPATH`
- `LANGOPTS`

The chosen values should be those most commonly used on your system. The default values should be set in `/etc/profile` and `/etc/csh.login`. You should arrange to do this and advise users of any change to the system default.

Users who need an environment different from the system default can set their own environment as needed in their `.profile` or `.login` file.

Localizing Message Catalogs

For applications that have message catalog support, you can provide a local language interface. This involves:

- Obtaining a copy of the "C" locale messages.
- Translating the messages into a local language.
- Installing a message catalog containing the translated messages.

The C Locale Messages

To determine what HP-UX commands have message catalogs, run:

```
ls /usr/lib/nls/C/*.cat
```

For each HP-UX command that has message catalog support, there will be a file `/usr/lib/nls/C/`*command.cat* listed.

To localize a message catalog, you need to first get a readable version of the "C" locale messages. This is done with the `dumpmsg` command. For example, to get a message text source file of the "C" locale messages for `date`, run:

```
dumpmsg /usr/lib/nls/C/date.cat >date.msg
```

The file `date.msg` is a copy of the messages and is ready for translation to a native language.

Translating Messages

First you must set the `LANG` environmental variable to the locale you will be translating in. You are now ready to translate the messages to the target language:

```
vi date.msg
```

Note that `date.msg` is a message text source file in a format suitable for input to `gencat`. You must preserve the format and you must leave the message numbers and the set numbers unchanged.

If the message translated is longer than 80 bytes, don't start a new line or just hit carriage return. Instead you can either:

- continue typing allowing the word to "wrap-around" to the new line, or
- type a \ (backslash) before the carriage return and continue typing at the first column of the next line.

If you want to put a carriage return in the message when it is output to the user, type `\n`. When the message is output to the user, the text following the `\n` will begin on a new line.

Translation Problems

The developer should have provided a translator's "cookbook". Without this, here are some possible translation problems you might encounter:

- The meaning of a message, or a substitution parameter, may be unclear or ambiguous so that the desired translation is not apparent.
- There may be unspecified size constraints on the message. For example, it may be displayed in a space with a fixed length.
- There may be parts of a message that should not be translated. For example, messages for a command may contain the command name.
- Some messages might be used together and need to be translated as a pair.

Some possible solutions you might try:

- Experiment with the program to see if you can determine the intended behavior.
- Ask the developer of the program.
- Ask someone who has localized the program.

Installing Localized Messages

Once the message text source file has been translated to the target language you can generate a message catalog containing the newly translated messages. First set the `LANG` environment variable to the locale you will be translating in. To create a message catalog from the translated `date.msg` message text source file, run:

```
gencat date.cat date.msg
```

The new message catalog `date.cat` can now be used for installation in the appropriate locale.

Note that a message catalog contains no information to indicate the locale for which it is intended. To help ensure that the message catalog is installed in the proper directory, we recommend you deliver the catalog with a script that will install the catalog in the correct locale. Once the new message catalog is installed, be sure to verify the correct installation.

Creating a Locale

The standard locales cover most languages. If none of the existing locales are appropriate, you can create a locale that meets your specific requirements. This is most easily done if there is an existing locale that is similar to the one you need. If there is, you can get a copy of the locale description in `localedef` format, modify the description so that it conforms to your needs, then install it as a new locale.

For example, suppose you need a locale that is the same as the `american` locale except that it has a different date format.

For the `american` locale, `date` produces output of the form:

```
Fri, May 5, 1989 04:37:33 PM
```

Suppose the desired format is:

```
Fri, 5 May 1989, 04:37:33 PM
```

The format for `date` is controlled by the `d_t_fmt` and `d_fmt` items of the `LC_TIME` category. You can change these to give the desired format.

To create the new locale, get a `localedef` script of the `american` locale by executing:

```
localedef -d american > new_locale
```

You can now modify the `localedef` script to define the desired locale:

```
vi new_locale
```

The script will contain the following entries:

```
    :
langname        "american"
langid  1
    :
LC_TIME
d_t_fmt "%a, %b %.1d, %Y %I:%M:%S %p"
d_fmt   "%a, %b %.1d, %Y"
t_fmt   "%I:%M:%S %p"
day_1   "Sunday"
    :
END LC_TIME
    :
```

5 To get the desired formatting, change `d_t_fmt` and `d_fmt` in the script to:

```
    :
langname        "locale_name"
langid  locale_id
    :
LC_TIME
d_t_fmt "%a, %.1d %b %Y %I:%M:%S %p"
d_fmt   "%a, %.1d %b %Y"
t_fmt   "%I:%M:%S %p"
    :
END LC_TIME
    :
```

Script Requirements for `localedef`

The above example shows how to modify an existing language script to create a language definition for a new locale. This section describes the structure and syntax of `localedef` in detail, making explicit the "rules" you must adhere to when defining a new locale.

In general, the categories in Table 5-1 make up a language definition:

Table 5-1. Primary Subdivisions of a Language Table

Category	Function
`LC_COLLATE`	Affects the behavior of the regular expressions and the NLS string collation functions.
`LC_CTYPE`	Affects the behavior of character classification and conversion functions.
`LC_MESSAGES`	Affects affirmative and negative response expressions.
`LC_MONETARY`	Affects the behavior of functions which handle monetary values.
`LC_NUMERIC`	Affects the handling of the radix character in the formatted input/output functions and the string conversion functions.
`LC_TIME`	Affects the behavior of the time conversion functions.
`LC_ALL`	Contains language-specific information which does not belong to any of the above categories.

In creating or adapting a new language script, use the following general principles:

- All information in a `localedef` script (except the language name, language id, revision number, comment character, and escape character) belongs to one of the above categories.
- The beginning of a category is identified by a "category tag" which has the form `LC_`*`category`*. All the values specified in the left side of the above table constitute legal category tags.
- The end of each category is identified by an `END LC_`*`category`* category tag.

- Categories can be listed in any order in a `localedef` script.
- All category specifications are optional. If a category is not specified, "C" locale is set up as the default for that category.

`localedef` Syntax

Table 5-1 outlined the basic structure of a `localedef` script. In addition to the category, the syntax of your `localedef` script provides for the specification of subcategories within each category. Introduce these subcategories through a pre-defined keyword followed by one or more expressions that specify applicable characteristics of the language. (These subcategories are described in later sections). `localedef` also recognizes three keywords that do not belong to any category. These keywords form a header for the language definition script, uniquely identifying the language:

5

Table 5-2. Header Keywords

keyword	Function
`langname`	String identifying the name of the language. This keyword is required by `localedef` if the command line invoking `localedef` does not contain the *locale_name*. (See *localedef*(1M).)
`langid`	Decimal number identifying the language ID. This keyword is required by `localedef` if the command line invoking `localedef` does not contain the *locale_name* (see *localedef*(1M)). The language ID specified should be in the range of 1 to 999, and any user-defined language should assign its language ID in the range of 901 to 999.
`revision`	String identifying the revision number of the `locale.inf` file. The string is restricted to contain at most 6 characters, all digits and one optional decimal point (.) character.

Table 5-2. Header Keywords (continued)

keyword	Function
`comment_char`	Single character indicating the character to be interpreted as starting a comment within the script. The default `comment_char` is `#`. All characters from a `comment_char` to the next newline are ignored.
`escape_char`	A single character indicating the character to be interpreted as an escape character within the script. The default escape_char is \. `escape_char` is used to escape localedef metacharacters to remove special meaning and in the character constant decimal, octal, and hexadecimal formats.

The following example shows the header of a `localedef` script for the `american` locale:

```
langname      "american"
langid        1
revision      "nnn.nn"
comment_char  '#'
escape_char   '\'
```

LC_ALL Subcategories

The following key words belong to the LC_ALL category:

Table 5-3. LC_ALL Subcategories

Subcategory Keyword	Function
direction	String indicating a text direction. If the null string or 0 is specified, text direction is left-to-right; if 1 is specified, text direction is right-to-left. In all cases, a top-to-bottom format is assumed.
context	String indicating if character context analysis is required. If the null string or 0 is specified no context analysis is required. If 1 is specified, Arabic context analysis is required.

5

Below is an example of LC_ALL. Note that no context analysis is required and text direction is from left to right. Comment lines begin with #. The script follows C-language conventions; strings are enclosed in double quotes.

```
##################################################

# LC_ALL category

LC_ALL
direction  ""
context ""
END LC_ALL
```

LC_COLLATE Subcategories

The following key words belong to the LC_COLLATE category:

Table 5-4. LC_COLLATE Subcategories

Subcategory Keyword	Function
`collating-element`	Defines a multi-character, collating-element symbol composed of two characters.
`order-start`	Denotes the start of the list of collating-element entries that define the collating sequence.
`order-end`	Marks the end of the list of collating-element entries that define the collating sequence.

A collation sequence defines the relative order between collating elements in the locale. This order is expressed in terms of collation values that are assigned in the order they occur (ascending), to each of the collating elements.

The collating element may be a single character or a multi-character (limited to 2 bytes) element. The collating order is achieved by explicitly ordering the collating elements between the `order_start` and `order_end` keywords. The collating element entry is of the form

collating-identifier [*primary-weight*; [*secondary-weight*]]

where, *collating-identifier* is one of the following:

- a single character element
- a multi character element (two-to-one character code pair)
- ellipsis
- the special keyword "UNDEFINED"

(Note that *primary-weight* and *secondary-weight* are optional.)

Single-Character Element

The single character collating element may consist of a character within single quotes ('), or it may consist of a hexadecimal, octal or decimal value. For example:

```
'7'      '7';'7'      character collating element
\x37 \x37;\x37        hexadecimal representation
\d55 \d55;\d55        decimal representation
\067 \067;\067        octal representation
```

Multi-character Element

Multi-character collating elements are defined using the `collating-element` keyword. This must be defined prior to the `order_start` keyword and takes the form

5 `collating-element` <*symbol*> `from` "*string*"

this defines the symbol *symbol* for the multi-character string *string*, which will collate as a single entity at the point where *symbol* appears in the collating sequence.

The following is an example of `LC_COLLATE` code that defines a collating sequence where the string `C1` collates between the characters `C` and `D`.

```
collating-element   <xx> from "C1"

order_start
 ...
'C'      'C';'C'
<xx>     <xx>;<xx>
'D'      'D';'D'
 ...
order_end
```

A one-to-two character code pair, which consists of single characters that occupy two adjacent positions in the collating sequence, may also be defined. In the following example, the *sharp s* is collated as SS and occurs between S and s in the collation sequence.

```
order_start
 ...
'S'      'S';'S'
\xde     'S';"SS"
's'      'S';'s'
 ...
```

```
order_end
```

Ellipsis Symbol

The ellipsis symbol (...) specifies that a sequence of characters collates according to its encoded character values. This means that when an ellipsis is used, all characters with a coded-character-set value greater than the value of the character in the preceding line and less than the coded-character-set value in the following line, will collate in between the two in ascending order, according to their coded-character-set values.

In the following example, the ellipsis indicates that the characters B, C, D and E (in the ASCII character set) will collate between A and F in the order according to their values in the character set.

```
order_start
'A' 'A';'A'
 ...
'F' 'F';'F'
order_end
```

In this case the order is B, C, D and E. Please note that the use of the ellipsis symbol ties the definition to a specific, coded-character set such as ASCII or EBCDIC.

`UNDEFINED` Symbol

The symbol `UNDEFINED` is interpreted as "including all coded-character-set values not specified explicitly or via the ellipsis symbol". Such characters that are not specified, but belong to the character set, are inserted in the character collation order at the point indicated by the symbol `UNDEFINED`, and in ascending order according to their coded-character-set values.

In the following example, A to Z will collate before 1 to 9 and all characters not defined (not between A-Z or between 1-9) will collate, according to their coded-character-set values, between `Z` and `1`.

```
order_start
'A' 'A';'A'
 ...
'Z' 'Z';'Z'
UNDEFINED
'1' '1';'1'
 ...
'9' '9';'9'
```

```
order_end
```

The optional operands for each collation element are used to define the primary or secondary weights for the collating element. Two or more elements can be assigned the same primary weight. If this is done they are said to be in the same equivalence class. Note that the following two collating-sequence definitions are equivalent:

```
order_start
'A'
'B'
'C'
order_end
```

and

```
order_start
'A' 'A';'A'
'B' 'B';'B'
'C' 'C';'C'
order_end
```

In the following example definition, the letters `A` and `a` both belong to the equivalence class `A` and collate before the character `B`. Similarly, the characters `B` and `b` belong to the same equivalence class. However, `a` and `b` collate after `A` and `B` respectively because their secondary weights are different.

```
order_start
'A' 'A';'A'
'a' 'A';'a'
'B' 'B';'B'
'b' 'B';'b'
order_end
```

If the keyword `IGNORE` is used as the secondary weight, it causes the collating element to be ignored during comparisons, as if the string did not contain the collating element.

Note An example of an `LC_COLLATE` sequence is in Appendix F.

LC_CTYPE Subcategories

- When using expressions to describe character traits, character-code ranges can be specified by listing two constants defining the range, separated by an ellipsis (...). The constant preceding the ellipsis must have a smaller code value than the constant following the ellipsis. A range represents a set of consecutive character codes.
- When using shift expressions a pair of character codes are enclosed by left and right parentheses. For `tolower`, the first constant represents an uppercase character and the second the corresponding lowercase character. For `toupper`, the first constant represents a lowercase character, and the second represents the corresponding uppercase character.

The `LC_CTYPE` category consists of the following keywords:

Table 5-5. LC_CTYPE Categories

Subcategory Keyword	Character Code
`upper`	Uppercase letters.
`lower`	Lowercase letters.
`digit`	Numeric characters.
`space`	Spacing (delimiter) characters.
`punct`	Punctuation characters.
`cntrl`	Control characters.
`blank`	Printable space characters. These must also be defined in `space`.
`xdigit`	Hexadecimal digits.

Table 5-5. LC_CTYPE Categories (continued)

Subcategory Keyword	Character Code
`alpha`	Character codes classified as alphabetic characters. If omitted, this class is the concatenation of the **upper** and **lower** classes.
`print`	Character codes classified as printable characters. If omitted this class is the concatenation of the **upper**, **lower**, **alpha**, **digit**, **xdigit**, and **punct** classes and the **space** character.
`graph`	Character codes classified as graphic characters. If omitted, this class is all characters included in the **print** class except the **space** character.
`first`	First bytes of two-byte characters.
`second`	Second bytes of two-byte characters.
`toupper`	Lowercase to uppercase character relationships.
`tolower`	Uppercase to lowercase character relationships.
`bytes_char`	String containing the maximum number of bytes per character for the character set used for a specific language.
`alt_punct`	String mapped into the ASCII equivalent string *b*`!"#$%&'()*+,-./:;<=>?@[\]^_‘{|}~`, where *b* is a blank.
`code_scheme`	Specifies the multi-byte character encoding scheme used. The operand should be a string. Currently, **HP-15** and **EUC** are recognized. If this keyword is not specified, or the operand is a null string (""), the encoding scheme is single-byte, or HP-15 if **bytes_char** is 2.
`cswidth`	Defines the number of bytes contained in a character, and the number of columns per character displayed on the output devices. This keyword should be specified if the encoding scheme is "EUC".

The following example displays the LC_CTYPE category for the american locale:

```
##################################################
# Set up the LC_CTYPE category

LC_CTYPE

upper 'A';'B';'C';'D';'E';'F';'G';'H';'I';'J';'K';'L'; \
 'M';'N';'O';'P';'Q';'R';'S';'T';'U';'V';'W';'X'; \
 'Y';'Z';\xa1;\xa2;\xa3;\xa4;\xa5;\xa6;\xa7;\xad;\xae;\xb1; \
 \xb4;\xb6;\xd0;\xd2;\xd3;\xd8;\xda;\xdb;\xdc;\xde;\xdf;\xe0; \
 \xe1;\xe3;\xe5;\xe6;\xe7;\xe8;\xe9;\xeb;\xed;\xee;\xf0

lower 'a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l'; \
 'm';'n';'o';'p';'q';'r';'s';'t';'u';'v';'w';'x'; \
 'y';'z';\xb2;\xb5;\xb7;\xc0;\xc1;\xc2;\xc3;\xc4;\xc5;\xc6; \
 \xc7;\xc8;\xc9;\xca;\xcb;\xcc;\xcd;\xce;\xcf;\xd1;\xd4;\xd5; \
 \xd6;\xd7;\xd9;\xdd;\xde;\xe2;\xe4;\xea;\xec;\xef;\xf1

alpha 'A';'B';'C';'D';'E';'F';'G';'H';'I';'J';'K';'L'; \
 'M';'N';'O';'P';'Q';'R';'S';'T';'U';'V';'W';'X'; \
 'Y';'Z';'a';'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';\xa1;\xa2;\xa3;\xa4;\xa5;\xa6;\xa7;\xad; \
 \xae;\xb1;\xb2;\xb4;\xb5;\xb6;\xb7;\xc0;\xc1;\xc2;\xc3;\xc4; \
 \xc5;\xc6;\xc7;\xc8;\xc9;\xca;\xcb;\xcc;\xcd;\xce;\xcf;\xd0; \
 \xd1;\xd2;\xd3;\xd4;\xd5;\xd6;\xd7;\xd8;\xd9;\xda;\xdb;\xdc; \
 \xdd;\xde;\xdf;\xe0;\xe1;\xe2;\xe3;\xe4;\xe5;\xe6;\xe7;\xe8; \
 \xe9;\xea;\xeb;\xec;\xed;\xee;\xef;\xf0;\xf1

graph '!';'"';'#';'$';'%';'&';''';'(';')';'*';'+';',';  \
 '-';'.';'/';'0';'1';'2';'3';'4';'5';'6';'7';'8'; \
 '9';':';';';'<';'=';'>';'?';'@';'A';'B';'C';'D'; \
 'E';'F';'G';'H';'I';'J';'K';'L';'M';'N';'O';'P'; \
 'Q';'R';'S';'T';'U';'V';'W';'X';'Y';'Z';'[';'\'; \
 ']';'^';'_';'`';'a';'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';'{';'|';'}';'~';\xa1;\xa2; \
 \xa3;\xa4;\xa5;\xa6;\xa7;\xa8;\xa9;\xaa;\xab;\xac;\xad;\xae; \
 \xaf;\xb0;\xb1;\xb2;\xb3;\xb4;\xb5;\xb6;\xb7;\xb8;\xb9;\xba; \
 \xbb;\xbc;\xbd;\xbe;\xbf;\xc0;\xc1;\xc2;\xc3;\xc4;\xc5;\xc6; \
 \xc7;\xc8;\xc9;\xca;\xcb;\xcc;\xcd;\xce;\xcf;\xd0;\xd1;\xd2; \
 \xd3;\xd4;\xd5;\xd6;\xd7;\xd8;\xd9;\xda;\xdb;\xdc;\xdd;\xde; \
 \xdf;\xe0;\xe1;\xe2;\xe3;\xe4;\xe5;\xe6;\xe7;\xe8;\xe9;\xea; \
 \xeb;\xec;\xed;\xee;\xef;\xf0;\xf1;\xf2;\xf3;\xf4;\xf5;\xf6; \
 \xf7;\xf8;\xf9;\xfa;\xfb;\xfc;\xfd;\xfe

print ' ';'!';'"';'#';'$';'%';'&';''';'(';')';'*';'+'; \
 ',';'-';'.';'/';'0';'1';'2';'3';'4';'5';'6';'7'; \
 '8';'9';':';';';'<';'=';'>';'?';'@';'A';'B';'C'; \
```

```
 'D';'E';'F';'G';'H';'I';'J';'K';'L';'M';'N';'O'; \
 'P';'Q';'R';'S';'T';'U';'V';'W';'X';'Y';'Z';'['; \
 '\';']';'^';'_';'`';'a';'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';'{';'|';'}';'~';\xa1; \
 \xa2;\xa3;\xa4;\xa5;\xa6;\xa7;\xa8;\xa9;\xaa;\xab;\xac;\xad; \
 \xae;\xaf;\xb0;\xb1;\xb2;\xb3;\xb4;\xb5;\xb6;\xb7;\xb8;\xb9; \
 \xba;\xbb;\xbc;\xbd;\xbe;\xbf;\xc0;\xc1;\xc2;\xc3;\xc4;\xc5; \
 \xc6;\xc7;\xc8;\xc9;\xca;\xcb;\xcc;\xcd;\xce;\xcf;\xd0;\xd1; \
 \xd2;\xd3;\xd4;\xd5;\xd6;\xd7;\xd8;\xd9;\xda;\xdb;\xdc;\xdd; \
 \xde;\xdf;\xe0;\xe1;\xe2;\xe3;\xe4;\xe5;\xe6;\xe7;\xe8;\xe9; \
 \xea;\xeb;\xec;\xed;\xee;\xef;\xf0;\xf1;\xf2;\xf3;\xf4;\xf5; \
 \xf6;\xf7;\xf8;\xf9;\xfa;\xfb;\xfc;\xfd;\xfe

digit '0';'1';'2';'3';'4';'5';'6';'7';'8';'9'

space \x9;\xa;\xb;\xc;\xd;' '

punct '!';'"';'#';'$';'%';'&';''';'(';')';'*';'+';','; \
 '-';'.';'/';':';';';'<';'=';'>';'?';'@';'[';'\'; \
 ']';'^';'_';'`';'{';'|';'}';'~';\xa8;\xa9;\xaa;\xab; \
 \xac;\xaf;\xb0;\xb3;\xb8;\xb9;\xba;\xbb;\xbc;\xbd;\xbe;\xbf; \
 \xf2;\xf3;\xf4;\xf5;\xf6;\xf7;\xf8;\xf9;\xfa;\xfb;\xfc;\xfd; \
 \xfe

cntrl \x0;\x1;\x2;\x3;\x4;\x5;\x6;\x7;\x8;\x9;\xa;\xb; \
 \xc;\xd;\xe;\xf;\x10;\x11;\x12;\x13;\x14;\x15;\x16;\x17; \
 \x18;\x19;\x1a;\x1b;\x1c;\x1d;\x1e;\x1f;\x7f;\x80;\x81;\x82; \
 \x83;\x84;\x85;\x86;\x87;\x88;\x89;\x8a;\x8b;\x8c;\x8d;\x8e; \
 \x8f;\x90;\x91;\x92;\x93;\x94;\x95;\x96;\x97;\x98;\x99;\x9a; \
 \x9b;\x9c;\x9d;\x9e;\x9f;\xff

blank \x9;' '

xdigit '0';'1';'2';'3';'4';'5';'6';'7';'8';'9';'A';'B'; \
 'C';'D';'E';'F';'a';'b';'c';'d';'e';'f'

toupper ('a','A');('b','B');('c','C');('d','D'); \
 ('e','E');('f','F');('g','G');('h','H'); \
 ('i','I');('j','J');('k','K');('l','L'); \
 ('m','M');('n','N');('o','O');('p','P'); \
 ('q','Q');('r','R');('s','S');('t','T'); \
 ('u','U');('v','V');('w','W');('x','X'); \
 ('y','Y');('z','Z');(\xb2,\xb1);(\xb5,\xb4); \
 (\xb7,\xb6);(\xc0,\xa2);(\xc1,\xa4);(\xc2,\xdf); \
 (\xc3,\xae);(\xc4,\xe0);(\xc5,\xdc);(\xc6,\xe7); \
 (\xc7,\xed);(\xc8,\xa1);(\xc9,\xa3);(\xca,\xe8); \
 (\xcb,\xad);(\xcc,\xd8);(\xcd,\xa5);(\xce,\xda); \
 (\xcf,\xdb);(\xd1,\xa6);(\xd4,\xd0);(\xd5,\xe5); \
```

5

```
 (\xd6,\xd2);(\xd7,\xd3);(\xd9,\xe6);(\xdd,\xa7); \
 (\xe2,\xe1);(\xe4,\xe3);(\xea,\xe9);(\xec,\xeb); \
 (\xef,\xee);(\xf1,\xf0)

tolower ('A','a');('B','b');('C','c');('D','d'); \
 ('E','e');('F','f');('G','g');('H','h'); \
 ('I','i');('J','j');('K','k');('L','l'); \
 ('M','m');('N','n');('O','o');('P','p'); \
 ('Q','q');('R','r');('S','s');('T','t'); \
 ('U','u');('V','v');('W','w');('X','x'); \
 ('Y','y');('Z','z');(\xa1,\xc8);(\xa2,\xc0); \
 (\xa3,\xc9);(\xa4,\xc1);(\xa5,\xcd);(\xa6,\xd1); \
 (\xa7,\xdd);(\xad,\xcb);(\xae,\xc3);(\xb1,\xb2); \
 (\xb4,\xb5);(\xb6,\xb7);(\xd0,\xd4);(\xd2,\xd6); \
 (\xd3,\xd7);(\xd8,\xcc);(\xda,\xce);(\xdb,\xcf); \
 (\xdc,\xc5);(\xdf,\xc2);(\xe0,\xc4);(\xe1,\xe2); \
 (\xe3,\xe4);(\xe5,\xd5);(\xe6,\xd9);(\xe7,\xc6); \
 (\xe8,\xca);(\xe9,\xea);(\xeb,\xec);(\xed,\xc7); \
 (\xee,\xef);(\xf0,\xf1)

bytes_char "1"
alt_punct ""
code_scheme ""
cswidth ""
END LC_CTYPE
```

LC_MESSAGES Subcategories

The following key words belong to the LC_MESSAGES category:

Table 5-6. LC_MESSAGES Subcategories

Subcategory Keyword	Function
yesexpr	Expression identifying the affirmative response for yes/no questions.
noexpr	Expression identifying the negative response for yes/no questions.
yesstr	String identifying the affirmative response for yes/no questions. This keyword is now obsolete and **yesexpr** should be used instead.
nostr	String identifying the negative response for yes/no questions. This keyword is now obsolete and **noexpr** should be used instead.

5

Following is an example for the LC_MESSAGES category:

```
##################################################

# LC_MESSAGES category

LC_MESSAGES
yesexpr  "^[yY]"
noexpr   "^[nN]"
yesstr   "yes"
nostr    "no"
END LC_MESSAGES
```

LC_MONETARY Subcategories

The following keywords belong to the LC_MONETARY category and should be placed between the category tags LC_MONETARY and END LC_MONETARY:

Table 5-7. LC_MONETARY Subcategories

Subcategory Keyword	Description
int_curr_symbol	Four-character string specifying the international currency symbol used. The first three characters are alphabetic, specifying the international currency symbol. The fourth character is the character used to separate the international currency symbol from the monetary quantity.
currency_symbol	Specifies the currency symbol applicable to the current locale.
mon_decimal_point	Specifies the decimal point used to format monetary quantities.
mon_thousands_sep	Specifies the separator used to group digits to the left of the decimal point in monetary quantities.
mon_grouping	A semicolon-separated list of integers. The initial integer defines the size of the group immediately preceding the decimal delimiter, and the following integers define the preceding groups (lconv item).
positive_sign	Specifies the character used to indicate positive monetary quantities.
negative_sign	Specifies the character used to indicate negative monetary quantities.
int_frac_digits	Specifies the number of fractional digits displayed in an internationally formatted quantity.
frac_digits	Specifies the number of fractional digits displayed in a locally formatted monetary quantity.

Table 5-7. LC_MONETARY Subcategories (continued)

Subcategory Keyword	Description
`p_cs_precedes`	Specifies if `currency_symbol` precedes or follows a nonnegative formatted monetary quantity. `1` indicates it precedes the value; `0` that it follows it.
`n_cs_precedes`	Specifies if `currency_symbol` precedes or follows a negative formatted monetary quantity. `1` indicates it precedes the value; `0` that it follows it.
`n_sep_by_space`	A value of `1` indicates that the `currency symbol` is separated by a space from a nonnegative formatted monetary value; a value of `0` indicates that it is not.
`p_sign_posn`	Specifies the string position of the `positive_sign` for a non-negative formatted monetary quantity. **Value:** **Indicates:** `0` Parentheses surround the quantity and currency symbol. `1` The sign string precedes the quantity and currency symbol. `2` The sign string succeeds the quantity and currency symbol. `3` The sign string immediately precedes the currency symbol. `4` The sign string immediately succeeds the currency symbol.
`n_sign_posn`	Specifies the position of the `negative_sign` in a negative formatted monetary quantity. Values indicate the same as for `p_sign_posn` above.
`crncystr`	Symbol for currency precede by : `-` if it precedes the monetary value, `+` if it follows the monetary value, and `.` if it replaces the radix symbol in the monetary value.

The following example displays the LC_MONETARY category for the american locale:

```
##################################################


# LC_MONETARY category

LC_MONETARY
int_curr_symbol "USD "
currency_symbol "$"
mon_decimal_point "."
mon_thousands_sep ","
mon_grouping 3
positive_sign ""
negative_sign "-"
int_frac_digits 2
frac_digits 2
p_cs_precedes 1
p_sep_by_space 0
n_cs_precedes 1
n_sep_by_space 0
p_sign_posn 1
n_sign_posn 1
crncystr "-US$"
END LC_MONETARY
```

LC_NUMERIC Subcategories

The following keywords belong to the LC_NUMERIC category and should be placed between the category tags LC_NUMERIC and END LC_NUMERIC.

Table 5-8. LC_NUMERIC Keywords

grouping	Specifies the number of digits that are grouped together as a unit in formatted non-monetary quantities.
decimal_point	Specifies the radix character used to format non-monetary quantities.
thousands_sep	Specifies the character used to separate groups of digits to the left of the decimal point character in formatted non-monetary quantities.
alt_digit	Specifies the string mapped into the ASCII equivalent string 0123456789*b*+-.,eE, where *b* is a blank.

5

The following example displays the LC_NUMERIC category for the american locale:

```
##################################################
# LC_NUMERIC category


LC_NUMERIC
decimal_point   "."
thousands_sep   ","
grouping        3
alt_digit       ""
END LC_NUMERIC
```

LC_TIME Subcategories

The following keywords are defined for the LC_TIME category and should be placed between the category tags LC_TIME and END LC_TIME:

Table 5-9. LC_TIME Subcategories

Keyword	Function
d_t_fmt	String specifying date and time format.
d_fmt	String specifying date format.
t_fmt	String specifying time format.
t_fmt_ampm	Time representation in the 12-hour clock format with am_pm.
day	Seven semicolon-separated strings giving names for the days of the week beginning with Sunday; they correspond to langinfo items day_1 through day_7.
abday	Seven semicolon-separated strings giving abbreviated names for the days of the week beginning with Sunday; they correspond to langinfo items abday_1 through abday_7.
mon	Twelve semicolon_separated strings giving names for the months, beginning with January; they correspond to langinfo items mon_1 through mon_12.
abmon	Twelve semicolon_separated strings giving abbreviated names for the months, beginning with January; they correspond to langinfo items abmon_1 through abmon_12.
am_pm	Two semicolon-separated strings giving the representations for AM and PM.
year_unit	Symbol for years.
mon_unit	Symbol for month.
day_unit	Symbol for day.
hour_unit	Symbol for hour.

Table 5-9. LC_TIME Subcategories (continued)

Keyword	Function
`min_unit`	Symbol for minute.
`sec_unit`	Symbol for second.
`era_d_fmt`	Default string for formatting the **%E** (Emperor/Era name and year) directive of *date*(1) and *strftime*(3C) if an individual era format is not specified for an era.

This example displays the `LC_TIME` category for the `american` locale:

```
##################################################
# LC_TIME category

LC_TIME
d_t_fmt  "%a, %b %.1d, %Y %I:%M:%S %p"
d_fmt "%a, %b %.1d, %Y"
t_fmt "%I:%M:%S"
t_fmt_ampm "%I:%M:%S %p"

day "Sunday";"Monday";"Tuesday";"Wednesday";"Thursday";"Friday";"Saturday"

abday "Sun";"Mon";"Tue";"Wed";"Thu";"Fri";"Sat"

mon "January";"February";"March";"April";"May";"June"; \
"July";"August";"September";"October";"November";"December"

abmon "Jan";"Feb";"Mar";"Apr";"May";"Jun"; "Jul";"Aug";"Sep"; \
"Oct";"Nov";"Dec"

am_pm "AM";"PM"

year_unit   ""
mon_unit    ""
day_unit    ""
hour_unit   ""
min_unit    ""
sec_unit    ""
era_d_fmt   ""

END LC_TIME
```

Installing a Language Definition Table

Before you install a language definition table, you need to determine *locale_name* and *locale_id*. If you want to create a new locale, these must not conflict with existing locales and the *locale_id* must be in the range 901-999.

After you have changed `new_locale`, you can install it in the system by executing:

```
localedef  -i new_locale
```

You may need to be root to do this or you can deliver `new_locale` to your system administrator for installation.

To verify correct installation of the new locale:

- Run `nlsinfo` to see that the new locale is displayed.
- Examine `/usr/lib/nls/config` to see that *locale_name* is listed with *locale_id*.
- Verify that a directory `/usr/lib/nls/`*locale_name* exists.
- Verify that a file `/usr/lib/nls/`*locale_name*`/locale.inf` exists.
- Set `LANG` to the `locale_name` locale and verify that `date` formats the date as desired.

6

Developing International Software

Read this chapter if you are: ▷ A programmer of internationalized software

This chapter covers the standard programming issues for:

- Developing international software using standardized interfaces
- Internationalizing existing software

(For a discussion of special cases see Chapter 8.)

General Programming Issues

The programming issues your software must accommodate are:

- Initialization
- Preservation of data integrity
- Processing of characters and strings

Aspects of International Program Design

Developing an internationalized application (or converting an existing program) increases the *flexibility* of your application. For the purist, an internationalized program is not without a certain aesthetic aspect due to its structural and functional robustness. An internationalized program is engineered such that all language dependent features are separate from the main program logic.

The WPI approach is the preferred approach, because you don't need any knowledge of any language other than your own, and you don't even need to be aware of the standard ways different languages and codesets can vary. The linguistic capacity of your program transcends your own linguistic knowledge, functioning "fluently" in many different linguistic environments. Screens, prompts, and error messages are displayed in the user's language. Data is processed according to the grammar of the user's language. All of this is accomplished without altering your source code. Translators merely create external message catalogs to interface with your program; language tables installed on the system are referenced at run time, specifying language and locale specific information.

6 Some advantages of an internationalized application are:

- Using message catalogs and language tables reduces the complexity and the amount of time required to localize a program. Source code documents do not have to be altered, which also eliminates the need to debug and re-test software.
- Adapting the user interface to local needs is simplified. Translators work with a discrete file containing the text to translate, instead of digging through a voluminous amount of source code. This means localizers do not have to be programmers.
- Using language tables consolidates the grammar and processing rules of each language. Once installed, these tables will support any internationalized program.
- Using message catalogs and language tables makes your software product easier to manufacture, stock, and ship.

This chapter will show you step-by step how create an internationalized application. What does creating an internationalized program involve? Table 6-1 contrasts a standard application with its internationalized counterpart.

Table 6-1.
A Comparison of a "Standard" Application with its NLS Version

Non-Internationalized	NLS Version
Uses ASCII coded character set only.	Support a variety of codesets necessary for the languages of international users. For example, `Roman8` for Western European and `american` locale users.
Supports only single-byte characters of the ASCII character set.	Supports single-byte and multi-byte characters.
Source code must be altered by programmers to create new language-specific versions of the program because messages are hard-coded into the source code.	One discrete program is structured to support a variety of linguistic environments. Translators without programming knowledge can translate message catalogs to extend program support; no alteration of source code is required because there are no hard-coded messages.
Restricts user to a single user-interface due to hard-coded messages.	Displays screens in the native language of the user by means of the message catalog system.
Manipulates data according to rules of the English language	Uses NLS tools to handle data in a language sensitive way.
Displays data on the screen in a left-to-right format	Displays data according to the text direction of the user's language.
Restricted to one date, time, numeric, and monetary format.	Adapts the presentation of data, time, numeric, and monetary strings to the user's unique requirements.

To develop an internationalized program, you must:

- Use NLS routines to handle data in a language-sensitive way.
- Query language tables for locale-specific formatting conventions.
- Learn to use message catalogs.

When developing an international program, identify those areas of the program that are language dependent. Structure your program so that it uses NLS routines to query the language tables of the user's native language for language dependent information. Use message catalogs instead of hard-coded messages, (see Chapter 7).

Other Aspects

Designing an international program with NLS is usually a straightforward process. Nevertheless, there are a number of special considerations. For example, make sure you reserve enough space in arrays and other data structures to accommodate wide characters. Since an international program supports character sets that contain multi-byte characters, the number of characters in a string is no longer equivalent to the number of bytes. You must allocate additional space to accommodate the larger character size.

We mention these special considerations before delving into the details of NLS in order to provide you with some of the "flavor" of international design. If you prefer, you can deal with these issues later. Now that the scope of international design has been outlined in general terms, it is time to begin the design process in earnest. The first thing you must do is to provide the appropriate initialization.

The first step in the proper initialization is the retrieval of the user's environment.

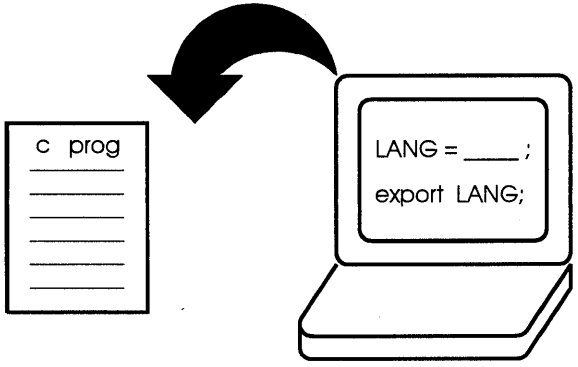

Figure 6-1. Retrieving the User's Environment

Note The user and program environment are not automatically identical.

Initializing NLS

At run-time your program must activate a specific linguistic environment. Users of your application set a series of NLS environmental variables according to their unique language requirements. This information is not made available to your program until you retrieve it. When you design international software, it is always necessary to provide the appropriate NLS initialization.

There are two elements of NLS that must be initialized to activate the NLS behavior of a program:

- The program locale
- The program messages

Two NLS routines are used to provide the appropriate initialization:

- The locale for a program is initialized by calling `setlocale` to make locale information accessible to the program.
- The messages for a program are initialized by calling `catopen` to locate the appropriate messages and make them accessible to the program.

6 **Note** The two initialization routines are independent. The routine `setlocale` copies information from the appropriate language tables onto the process heap. Your program can later reference this information to provide language-sensitive processing. `catopen` opens a message catalog, and thus affects aspects of the user interface and not language-sensitive processing.

Setting Program Locale

To activate your program's locale, you must supply two parameters to the `setlocale` routine:

`setlocale(`*category*`,` *locale*`)`

The *category* parameter specifies which areas of the program's NLS environment you wish to set. The options you have for this parameter are outlined in the following table.

Table 6-2. Categories for setlocale

Category Option	Description
`LC_ALL`	Affects the behavior of all categories below.
`LC_COLLATE`	Affects the behavior of regular expressions and the NLS string collation functions.
`LC_CTYPE`	Affects the behavior of regular expressions, character classification and conversion functions, and all routines which process multi-byte characters.
`LC_MESSAGES`	Affects affirmative and negative response expressions.
`LC_MONETARY`	Affects the behavior of functions which handle monetary values.
`LC_NUMERIC`	Affects the handling of the radix character in the formatted input/output functions, the string conversion functions, and the numeric values found in the `localeconv` structure.
`LC_TIME`	Affects the behavior of time conversion functions.

The locale parameter specifies the specific language table to use as a template for the setting of the values for the selected category. To set the program's locale for the specified category, `setlocale` will accept the following parameters:

Table 6-3. Locale parameters

Option	Description
locale name	If *locale* is a valid locale name, `setlocale` will set that part of the NLS environment associated with *category* as defined for that locale.
`C` or `POSIX`	If the value of *locale* is set to `C`, `setlocale` will set that part of the NLS environment associated with *category* as defined for the "C" locale. ("C" is a standard required by X/Open and POSIX).
"" (empty string)	If the value of *locale* is the empty string, the setting of that part of the NLS environment associated with *category* will depend on the current settings of the user's environmental variables.

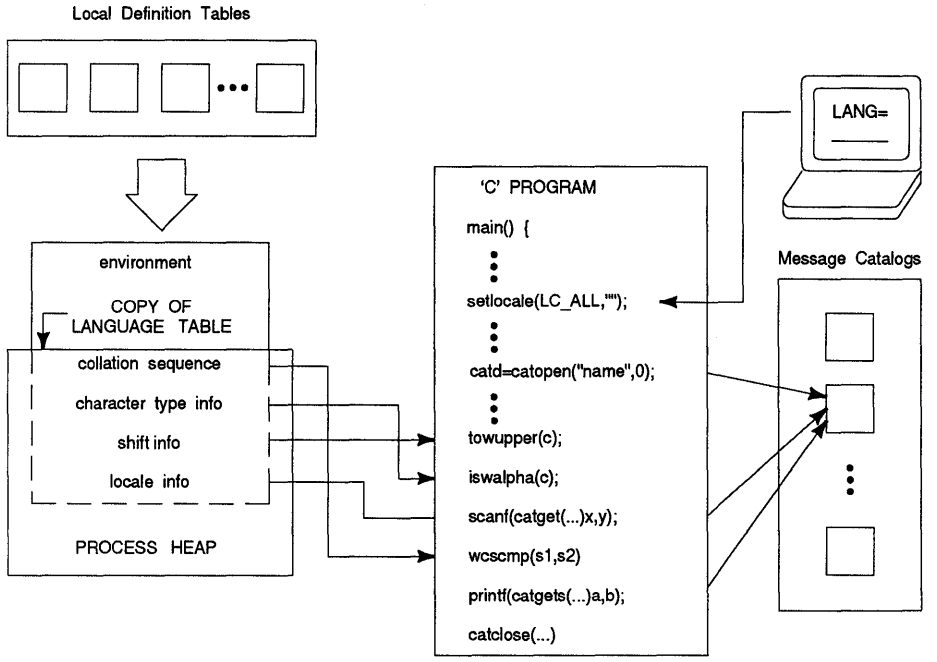

Figure 6-2. The Program Environment

6

Retrieving Locale Information

Accessing Language Tables

Each supported language has an associated language table that contains locale-specific information.

You can also use the NLS `nl_langinfo` routine to programmatically retrieve locale-specific information in your programs. To call `nl_langinfo`, you must supply a single parameter indicating the type of information you are requesting:

```
nl_langinfo(type);
```

The parameter *type* may be any of the categories shown in Table 6-4.

Table 6-4. Parameters Defined for nl_langinfo

Item	Description
D_T_FMT	Date and time format string appropriate for the current language.
D_FMT	Date format string appropriate for the current language.
T_FMT	Time format string appropriate for the current language.
DAY_*n*	The name of *n*th day of the week, where *n* ranges from 1 to 7.
ABDAY_*n*	The abbreviated name for the *n*th day of the week, where *n* ranges from 1 to 7.
MON_*n*	The name of the *n*th month in the Gregorian year, where *n* ranges from 1 to 12.
ABMON_*n*	The abbreviated name of the *n*th month in the Gregorian year, where *n* ranges from 1 to 12.
RADIXCHAR	Radix character ("decimal point" in English).
THOUSEP	Separator for thousands ("comma" in English).
YESEXPR	Affirmative response expression for yes/no questions.
NOEXPR	Negative response expression for yes/no questions.
YESSTR	Affirmative response for yes/no questions. [Note that this item will be withdrawn in a future POSIX revision.]
NOSTR	Negative response for yes/no questions. [Note that this item will be withdrawn in a future POSIX revision.]
CRNCYSTR	Symbol for currency preceded by: - (minus) if it precedes the monetary value, + if it follows the monetary value, and . if it replaces the radix symbol in the monetary value.
BYTES_CHAR	Maximum number of bytes per character for the character set used to represent the language.

Table 6-4. Parameters Defined for nl_langinfo (continued)

Item	Description
DIRECTION	Value to indicate text direction. The values **null** and **0** indicate the characters are arranged from left-to-right within a line and lines are arranged from top-to-bottom. A value of "1" indicates characters are arranged from right-to-left within a line and lines are arranged from top-to-bottom.
ALT_DIGIT	A string of characters that are mapped into the ASCII equivalent string **0123456789 +-.,eE**. A null value for the string indicates the language has no alternative digits.
ALT_PUNCT	A string whose characters are mapped into the ASCII equivalent string **!"#%&()*+-./:;<=>?@[ ]^_'{|}~** in **american** locale usage. A null value for the string indicates the language has no alternative punctuation characters.
AM_STR	Equivalent symbols for AM (before noon). Used with 12-hour time.
PM_STR	Equivalent symbols for PM (after noon).
YEAR_UNIT	Symbol for years.
MON_UNIT	Symbol for month.
DAY_UNIT	Symbol for day.
HOUR_UNIT	Symbol for hours.
MIN_UNIT	Symbol for minute.
SEC_UNIT	Symbol for second.
ERA_D_FMT	Default string for formatting the Emperor/Era name and year.
T_FMT_AMPM	Appropriate time representation in the 12-hour clock format with AM_STR and PM_STR.

The following code segment prints the days of the week for the current locale. To view different formats, set your **LANG** to various locales before running the program.

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

void display_days()
{
   printf("%s\n", nl_langinfo(DAY_1));
   printf("%s\n", nl_langinfo(DAY_2));
   printf("%s\n", nl_langinfo(DAY_3));
   printf("%s\n", nl_langinfo(DAY_4));
   printf("%s\n", nl_langinfo(DAY_5));
   printf("%s\n", nl_langinfo(DAY_6));
   printf("%s\n", nl_langinfo(DAY_7));
}

main()
{
   if (!setlocale(LC_ALL,""))
      fprintf(stderr, "error: cannot set locale\n");
   display_days();
}
```

Programming with the Worldwide Portability Interface

One objective of international program design is to create an application that is codeset independent. The Worldwide Portability Interface (WPI) automatically converts data to/from its internal coding to/from wide characters, which keeps the programmer from having to know anything about wide character sets.

For some applications, character processing may be more convenient if multi-byte characters are represented as constant width characters—so-called wide characters. (For more information on multi-byte characters see Appendix A.)

For such situations, a set of routines is available to convert between multi-byte characters and wide characters. The wide character representation is more convenient because it offers a single data type for all languages.

WPI Interfaces

Use the following steps to internationalize an existing program with the WPI Interfaces:

6

1. Convert `char` data types to `wchar_t`.
2. Where parallel routines exist, change calls to the WPI versions.

Instead of this	Use this
`strcmp`	`wcscmp`
`isdigit`	`iswdigit`
`fgetc`	`fgetwc`

3. In cases where no parallel routine exists, you need to add code to convert wide character data to multibyte, using `wcstombs`, for example. Then pass multibyte data to "standard" routines.

Character and String Processing

Character and string processing for international software must ensure that local customs are observed in:

- Treating of accented characters
- Formatting date and time
- Formatting numeric and monetary quantities
- Comparing string data

Character Handling

Table 6-5 lists the WPI routines that are similar to the standard character and string processing routines.

Table 6-5. WPI Routines for Character and String Processing

string(3C) Routine	*wcstring*(3C) Routine (WPI)
char *strcat	wchar_t *wcscat
char *strncat	wchar_t *wcsncat
int strcmp	int wcscmp
int strncmp	int wcsncmp
char *strcpy	wchar *wcscpy
char *strncpy	wchar_t *wcsncpy
size_t strlen	size_t wcslen
char *strchr	char *wcschr
char *strrchr	wchar_t *strrchr
char *strpbrk	wchar *wcspbrk
size_t strspn	size_t wcsspn
size_t strcspn	size_t wcscspn
char *strstr	wchar_t *wcswcs
char *strtok	wchar_t *wcstok
int strcoll	int wcscoll
size_t strxfrm	size_t wcsxfrm

Upshifting and Downshifting Characters. The WPI provides the following routines to upshift and downshift characters in a language-sensitive way. You can obtain more information about this specific group of routines in *conv*(3C) in the *HP-UX Reference*

Table 6-6. Character and String Processing Routines

Routine	Description
`towupper(`*c*`)`	If *c* represents a valid lowercase letter for the current language, `towupper(`*c*`)` returns the integer character code of the upshifted value of *c*; otherwise `towupper(`*c*`)` returns *c* unaltered.
`towlower(`*c*`)`	If *c* represents a valid uppercase letter for the current language `towlower(`*c*`)` returns the integer character code of the downshifted value of *c*; otherwise, `towlower(`*c*`)` returns *c* unaltered.

Identifying Character Traits. The WPI provides tools to identify the traits of a character. Non-internationalized versions of these routines have been a part of the standard C offering for years; however, the enhanced NLS versions can correctly identify the traits of characters in all supported languages (If the correct environmental initialization has been successful).

All the routines included in this grouping return a non-zero integer if the indicated condition is satisfied. For more information about these routines refer to *wctype*(3C) in the *HP-UX Reference*

Table 6-7. Character Identification Routines

Routine	Condition
`iswalpha(`*c*`)`	Is a wide alphabetic character.
`iswupper(`*c*`)`	Is a wide uppercase alphabetic character.
`iswlower(`*c*`)`	Is a wide lowercase alphabetic character.
`iswdigit(`*c*`)`	Is a wide decimal digit.
`iswxdigit(`*c*`)`	Is a wide hexadecimal digit.
`iswalnum(`*c*`)`	Is a wide alphanumeric character.
`iswspace(`*c*`)`	Is a tab, new-line, space, or any wide character that creates "white space" in displayed text.
`iswpunct(`*c*`)`	Is a wide punctuation character.
`iswprint(`*c*`)`	Is a wide printing character.
`iswgraph(`*c*`)`	Is a wide character with a visible representation.
`iswcntrl(`*c*`)`	Is a wide control character.

Numeric Formatting

The numeric formatting routines (for example, `printf`, and `fprintf`) have been internationalized and give locale-sensitive results for wide character data.

Table 6-8. Numeric Formatting Routines

Routine	Description
`wcstod`	Converts a wide character string to a *double*, correctly interpreting the radix character according to the currently active value of the `LC_NUMERIC` category.
`wcstol`	Converts a wide character string to a long.
`wcstoul`	Converts a wide character string to an unsigned long.

Date and Time

The *ctime*(3C) date and time routines: `ctime` and `asctime` always give `C` locale results. To get locale-sensitive results, use `wcsftime`.

6

Table 6-9. Date and Time Routines

Routine	Description
`wcsftime`	Like `ctime`, but accepts an optional format string (which should be placed in a message catalog).
`nl_langinfo(D_T_FMT)`	Returns a local time/date format string.

Note See "Input and Output in Internationalized Programs" for a better understanding of time/date format strings and their use.

Input and Output in Internationalized Programs

In addition to the regular formatting of input and output strings and numeric values, the WPI provides a flexible formatting feature to help you develop applications supporting the diverse conventions of various languages and locales.

Printing Formatted Output. The conversion character **%** may be replaced with the sequence **%***n*$, where *n* is a decimal digit in the range 1 to 9, specifying which argument you want the conversion applied to. For instance, if `string1 = "there"` and `string2 = "hello"`, `printf` will perform as follows:

```
Statement:    printf("%2$s%1$s.", string1, string2);
Result:       hello there.
```

The following print routines also support the NLS flexible formatting feature:

`fprintf(`*stream, format [arg list]*`)`
`sprintf(`*s, format [arg list]*`)`

And there are the wide character conversion characters, `C` and `S` that can be used with the *printf*(3C) routines:

Character	Conversion Function
C	format a single wide character
S	format a wide character string

Reading Formatting Input. The flexible formatting feature also applies to all NLS-supported read routines. The conversion character **%** is replaced by the sequence **%***n*$, where *n* is in the range 1 to 9. The conversion is applied to the *n*th argument rather than the next unused one.

The following read routines support the flexible formatting feature:

`scanf(`*format* `[,` *pointer*`]...)`
`fscanf(`*stream, format* `[,` *pointer*`]...)`
`sscan(`*s, format* `[,` *pointer*`]...)`

And there are the wide character conversion characters, `C` and `S` that can be used with the *scanf*(3C) routines:

Character	Conversion Function
C	format a single wide character
S	format a wide character string

Using Flexible Formatting. The flexible formatting feature enables you to read and write order-sensitive information for different locales. Examples of such order-sensitive information include:

- Relative position of day and month in date.
- Position of the currency symbol.
- Radix symbol.

For instance, to print a locale-specific version of the date include the following statement in your program:

```
printf((catgets(nlmsg,NL_SET,3,"%1$d%2$d/%3$d")), month,day,yr)
```

To handle the differences in conventions, each message catalog contains the specific formatting option needed. For example, to print the date, an `american` locale catalog contains the formatting string:

```
1 The date is: %1$d/%2$d/%3$d
```

A German catalog contains:

```
1 Heute ist: %2$d.%1$d.%3$d
```

6

Suppose `month` = 1 (Jan), `day` = 25, and `year` = 90 (1990). If `LANG=german`, the following is printed:

```
25.1.90
```

If `LANG=american`, then the following is printed:

```
1/25/90
```

If `LANG` is not set or if `catopen` failed during program initialization, the default string included in the call to `catgets` is used.

Note It is enough to understand the general principle of flexible formatting here. The specifics of messaging are explained in Chapter 7 and flexible formatting is explained in greater detail in Chapter 8.

Table 6-10. Multi-byte Character and String Conversions

Routine	Function
`mblen`	If *s* is not a null pointer, the function examines the next *n* bytes. If the next *n* or fewer bytes form a valid multi-byte character, the number of bytes in the multi-byte character is returned. If they do not form a valid multi-byte character, `-1` is returned. If *s* points to the null character, `0` is returned.
`mbtowc`	Converts a multi-byte character pointed to by *s* to its wide character representation, storing the result in the array pointed to by *pwc*. The number of bytes in the original multi-byte character is returned; at most *n* bytes are examined.
`wctomb`	Converts a wide character pointed to by *wchar* to its multi-byte representation, storing the result in the array pointed to by *s*. The number of bytes in the converted character is returned.
`mbstowcs`	Converts a sequence of multi-byte characters in the array pointed to by *s* into a sequence of corresponding wide character codes, storing the result in the array pointed to by *pwcs*, and stopping after *n* codes are examined, or a null character is encountered.
`wcstombs`	Converts a sequence of wide character codes in the array pointed to by *pwcs* into a sequence of multi-byte characters codes and stores them in the array pointed to by *s*, stopping after *n* bytes are examined, or a null character is encountered.

Example program using wide characters.. The following program uses wide characters, and works for both HP-15 and EUC encoding schemes. The program folds characters strings. A field size is specified defining the WIDTH for text displayed on the screen. As a given string is printed, any character whose display would fall outside the designated text region is "folded" onto the next line.

For instance, if the `WIDTH = 5` and the string is `0123456789`, the result is:

```
01234
56789
```

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <nl_ctype.h>

#define WIDTH 5
#define MAX_MULTI 100                          /* Max number of bytes in
                                                  multi-byte array */

#define MAX_WCHAR (MAX_MULTI * (sizeof(wchar_t)))  /* Max number of bytes in
                                                      w_char array */

main(argc, argv)
int argc;
char **argv;
{
   unsigned char mb_out_array[WIDTH * 2 + 1]; / *array for output multi-byte
                                                string */
   unsigned char wc_array[sizeof(wchar_t)];   /* array to hold one wchar */
   char *mb_in_ptr;                           /* pointer to input multi-byte
                                                 array */
   wchar_t wc_in_array[MAX_WCHAR];            /* array for input wchar string */
   wchar_t wc_out_array[WIDTH * (sizeof(wchar_t)) + 1];
                                              /* array for output wchar
                                                 string */
   wchar_t *in_wc_ptr, *out_wc_ptr, wc;
   int counter, screen_size;

   if (!setlocale(LC_ALL, "")) {              /* set the locale */
      fprintf(stderr,"error: cannot set locale\n");
   }
   mb_in_ptr = argv[1];                       /* set pointer to beginning of
                                                 multi-byte string */
   counter = 0;                               /* initialize counter to 0 */
   in_wc_ptr = wc_in_array;                   /* set pointer to beginning of
                                                 input wc array*/
```

```
    out_wc_ptr = wc_out_array;                  /* set pointer to beginning of
                                                   output wc array */
    mbstowcs(wc_in_array, mb_in_ptr, MAX_MULTI);
                                                /* convert input multi_byte
                                                   string to wc array*/
    while (wc = *in_wc_ptr++) {                 /* get a character from the wc
                                                   array until '\0' encountered */
       wctomb(wc_array, wc);                    /* convert wc to a multi-byte
                                                   character */
       screen_size = C_COLWIDTH(*wc_array);     /* get screen size of wc */
       if (counter + screen_size > WIDTH) {     /* if output array is complete
                                                   process it */
          *out_wc_ptr = '\0';                   /* null terminate the output
                                                   array*/
          wcstombs(mb_out_array, wc_out_array, MAX_MULTI);
                                                /* convert the wc array to a
                                                   multi-byte array */
          puts(mb_out_array);                   /* output the multi-byte array */
          counter = 0;                          /* reset counter to 0 */
          out_wc_ptr = wc_out_array;            /* reset wc output pointer to
                                                   beginning of wc array */
       }
       counter += screen_size;                  /* if output array not complete,
                                                   increment counter */
       *out_wc_ptr++ = wc;                      /* add wc to the output array
                                                   and increment pointer */
    }
    *out_wc_ptr = '\0';                         /* null terminate output array */
    wcstombs(mb_out_array, wc_out_array, MAX_MULTI);
                                                /* convert wc output array to
                                                   multi-byte */
    puts(mb_out_array);                         /*output the multi-byte array*/

}
```

Conversion of Existing Programs

Converting existing programs is an ad hoc process. Use the `grep` command on existing source code to find calls to routines, such as `ctime` and `strcmp`, which may require changes.

Non-WPI Interfaces

These standards-based interfaces were developed prior to the WPI and are still used in many existing programs.

Character and String Processing

Character and string processing for international software must ensure that local customs are observed in:

- Treating of accented characters
- Formatting date and time
- Formatting numeric and monetary quantities
- Comparing string data

Character Handling

Most character and string processing is provided by internationalized library routines that give correct results for the currently active locale. Note that there are restrictions in some library routines and minor program changes may be needed.

6

Upshifting and Downshifting Characters. NLS provides the following routines to upshift and downshift characters in a language-sensitive way. You can obtain more information about this specific group of routines in *conv*(3C) in the *HP-UX Reference*.

Table 6-11. Character and String Processing Routines

Routine	Description
`toupper(c)`	If *c* represents a valid lowercase letter for the current language, `toupper(c)` returns the integer character code of the upshifted value of *c*; otherwise `toupper(c)` returns *c* unaltered. The domain of this routine is `-1` to `255`
`tolower(c)`	If *c* represents a valid uppercase letter for the current language `tolower(c)` returns the integer character code of the downshifted value of *c*; otherwise, `tolower(c)` returns *c* unaltered. The domain of this routine is `-1` to `255`.
`_toupper(c)`	This macro performs like `toupper` but is faster and has a restricted domain of `0` to `255`. *c* must be a valid lower case character.
`_tolower(c)`	This macro performs like `tolower` but is faster and has a restricted domain of `0` to `255`. *c* must be a valid upper case character.

Identifying Character Traits. NLS provides tools to identify the traits of a character. Non-internationalized versions of these routines have been a part of the standard C offering for years; however, the enhanced NLS versions can correctly identify the traits of characters in all supported languages (If the correct environmental initialization has been successful).

All the routines included in this grouping return a non-zero integer if the indicated condition is satisfied. For more information about these routines refer to *ctype*(3C) in the *HP-UX Reference*.

Table 6-12. Character Identification Routines

Routine	Condition
`isalpha(`*c*`)`	Is an alphabetic character.
`isupper(`*c*`)`	Is an uppercase alphabetic character.
`islower(`*c*`)`	Is a lowercase alphabetic character.
`isdigit(`*c*`)`	Is a decimal digit.
`isxdigit(`*c*`)`	Is a hexadecimal digit.
`isalnum(`*c*`)`	Is an alphanumeric character.
`isspace(`*c*`)`	Is a tab, new-line, space, or any character that creates "white space" in displayed text.
`ispunct(`*c*`)`	Is a punctuation character.
`isprint(`*c*`)`	Is a printing character.
`isgraph(`*c*`)`	Is a character with a visible representation.
`iscntrl(`*c*`)`	Is a control character.
`isascii(`*c*`)`	Is a ASCII character.

Note Although the form of the new language-sensitive character identification routines does not differ from the "ASCII only" routines, the type of argument passed has changed from `char` to `int`. When passing 8-bit bytes to these routines, the data should be `unsigned char`, or cast to it, to ensure correct sign extension.

Numeric Formatting

The numeric formatting routines (for example, `ecvt`, `gcvt`, `atof`, `printf`, and `fprintf`) have been internationalized and give locale-sensitive results for single-byte and multi-byte data. For information about restrictions on the use of multi-byte data, see *ecvt*(3C), *strtod*(3C), and *printf*(3C) in Section 3 of the *HP-UX Reference*.

Table 6-13. Numeric Formatting Routines

Routine	Description
`atof`	Converts a string to a *float*, correctly interpreting the radix character according to the currently active value of the `LC_NUMERIC` category.
`strtod`	Converts a string to a *double*, correctly interpreting the radix character according to the currently active value of the `LC_NUMERIC` category.
`gcvt`	Converts a *double* to a string and places the correct radix character according to the currently active value of the `LC_NUMERIC` category.

Date and Time

The *ctime*(3C) date and time routines: `ctime` and `asctime` always give `C` locale results. To get locale-sensitive results, use `strftime`.

Table 6-14. Date and Time Routines

Routine	Description
`strftime`	Like `ctime`, but accepts an optional format string (which should be placed in a message catalog).
`nl_langinfo(D_T_FMT)`	Returns a local time/date format string.

Monetary Formatting

Generalized monetary formatting is more involved than numeric formatting since in some countries the currency symbol is placed before the amount, while in other countries it is placed within or after the amount. There are no library routines that provide monetary formatting; you will have to provide your own.

The currency symbol and position information is available in the structure returned by `localconv` (this information is also available through the slower `nl_langinfo` call described shortly). The following example illustrates how you may use this information to flexibly format monetary values for an internationalized application.

The following example program prints a monetary value for the currently active locale. The NLS library routine `localeconv` returns a structure containing locale-specific numeric formatting conventions. Two members of this structure are accessed:

`p_cs_precedes` — Set to `1` if the currency symbol precedes a non-negative monetary quantity; otherwise set to `0`.

`n_cs_precedes` — Set to `1` if the currency symbol precedes a negative monetary quantity; otherwise set to `0`.

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

main(argc, argv)
int argc;
```

```
char **argv;
{
   struct lconv *lconv_ptr;            /* pointer to structure returned by
                                          localeconv*/
   float  number;
   char   *cs_precedes, *cs_succeeds;  /* pointers for the preceding currency
                                          symbol and the succeeding currency
                                          symbol*/

   if (!setlocale(LC_ALL,""))          /*set the locale*/
      fprintf(sterr, "error: cannot set locale\n");
   lconv_ptr = localeconv();
   number = (float)atof(argv[1]);      /* convert string to a float */

   /* if the number is non-negative and the currency symbol precedes a
      non-negative formatted monetary quantity OR if the number is
      negative and the currency symbol succeeds a negative formatted
      monetary quantity set the currency symbol precedes pointer to
      the currency symbol */

<rev begin>
    if (number >= 0 && lconv_ptr->p_cs_precedes == 1 ||
       number < 0 && lconv_ptr->n_cs_precedes == 1 )   {
<rev end>
       cs_precedes = lconv_ptr->currency_symbol;
       cs_succeeds = "";
    }
    /* otherwise set the currency symbol succeeds to the currency symbol */

    else {
       cs_precedes = "";
       cs_succeeds = lconv_ptr->currency_symbol;
    }
    printf("%s %6.2f %s\n", cs_precedes, number, cs_succeeds);
}
```

Other information in the `lconv` structure describes decimal point, thousands separator, spaces used with the currency symbol, and other locale-specific formatting information.

String Comparisons

The *string*(3C) string comparison routines **strcmp** and **strncmp** always give C locale results. To get locale-sensitive results use **strcoll**.

For some applications, in particular ones in which many strings are compared against one constant string, you can improve performance by using **strxfrm** to convert strings to a form that can be compared using **strcmp**. The following program illustrates this application:

```
char *s1, *s2, *t1, *t2;
int  n1, n2;
     :
strxfrm(s1, t1, n1);
strxfrm(s2, t2, n2);
     :
if ( strcmp(t1, t2) > 0 ) {
    /* == strcoll(s1, s2) */
     :
```

Note that error checking the conversion by **strxfrm** is omitted.

Guidelines for Creating Internationalized Programs

- Whenever possible, use the WPI to handle character data. This allows a single-compiled application to handle all system-supported single- and multi-byte character encodings.
- Always *maintain data integrity*. Do not do *any* processing or examining of character data if you do not have to. If you are simply passing data through a routine or application, do not zero out the high bit of each byte, or pad it into even-sized blocks, or substitute linefeeds for carriage returns, etc.
- Never use the 8th bit of a character byte as a flag. This was a practice that grew out of the fact the ASCII is a 7-bit code. It is no longer acceptable.
- Never hardcode character constants. The run-time character set may be different from the compile-time character set. Comparing against a character constant may not work if the character has a different code in the run-time environment than it did in the compile-time environment. This is especially likely for characters not found in the ASCII set. If it is possible to put these values in files, where they can be changed, do so. If not, at least assign symbolic names to them (like `slash`) so it will be easy to track down and change such codeset-dependent information later. Likewise, do not hardcode numeric constants to represent characters (like decimal 32 for "space").
- Never use "ASCII" rules for character transformation or identification. Upshifting characters by adding decimal 32, or testing for alphabetic characters by comparing against numeric range, will probably not work for non-ASCII character sets. Use system-supplied language/character-set sensitive routines for identifying or up/down shifting characters.
- When sorting data for end-user viewing, do not use numeric comparisons of strings or characters for collation. System routines exist for the proper, language-sensitive, collation of text strings. The only cases where comparisons based on character's numeric representations should be used are when testing two strings for equality, or for hash tables, b-trees, etc., which are never directly viewed or accessed by end-users. In these cases the higher-performing, numeric comparisons may be appropriate.

7

The Message Catalog System

Read this chapter if you are: ▷ A programmer of internationalized software

This chapter covers the following programming issues related to using message catalogs with internationalized software :

- Initialization
- Retrieving Messages
- Creating a New Message Catalog
- Installing a Message Catalog

Creating and Using a Message Catalog System

The HP-UX message catalog system allows program messages to be stored separately from the logic of the program, to be translated into different languages, and to be retrieved at run-time, according to the language requirements of each user.

Program messages might be:

- Information to the user, e.g. `file not found`.
- Responses from the user, e.g. `tomorrow` as used by the `at` command.
- Strings used to format other messages, e.g. `%1$d %2$s\n`.

These messages would ordinarily appear in the source program as quoted strings, such as:

```
printf("file not found\n");
    :
if ( strcmp(s, "tomorrow") == 0 ) ...
    :
```

To produce a program that is internationalized for messages, do the following:

1. Separate the program logic from program messages by using message routine calls in place of quoted messages in the source program. The message routines will retrieve message text at run-time.
2. Create a message text source file for localization. This file contains messages that would ordinarily appear as quoted strings in the source program.
3. Generate a message catalog from the message text source file. This file contains messages that are retrieved by the message routines.

Localized messages can then be provided by translating the strings in the message text source file into another native language and then generating the native language message catalog.

Programming for Messages

The programming tools for messaging are:

- The `findmsg` command extracts messages from a C program source file and writes them to the standard output in a format suitable for input to `gencat`.
- The `dumpmsg` command extracts messages from a message catalog file created by `gencat`. The messages are written to standard output in a format suitable for editing and re-input to `gencat`.
- *gencat*(1); `gencat` produces a message catalog from message text source files.
- The `catopen` function, which locates a named message catalog and prepares it for use by `catgets` and `catclose`.
- The `catgets` function, which retrieves messages from a message catalog opened by a call to `catopen`.
- The `catclose` function, which closes a message catalog opened by `catopen`.

Opening a Message Catalog with catopen

`catopen` opens a message catalog for reading and returns a catalog descriptor of type `nl_catd`. (Include `nl _types.h` at the beginning of your program to use this type.) Call `catopen` by:

`catopen("`*name*`",` *oflag*`)`

where:

name is a file name enclosed in quotes, indicating the name of the catalog to open.

oflag is reserved for future use and should be set to `0`.

- `catopen` returns a message catalog descriptor if successful.
- Otherwise, a value of `(nl_cat)-1` is returned.

Note We recommend that you use the program name as the *name* argument supplied to `catopen`. This provides a generic descriptor of the message catalog to open. At run time your program selects the particular message catalog supporting the user's language according to the current setting of `LANG` and the path(s) specified by environmental variable `NLSPATH`.

Recommended Initialization

Use `setlocale` and `catopen` to initialize your program. For most applications, the following initialization is recommended:

```
#include <nl_types.h>
   :
nl_catd catd;
   :
if ( !setlocale(LC_ALL, "") ) {
    fputs("setlocale failed,
       continuing with \"C\" locale.", stderr);
    putenv("LC_ALL=");
    catd = (nl_catd)-1;
    }
else
    catd = catopen("name", 0);
    :
```

7

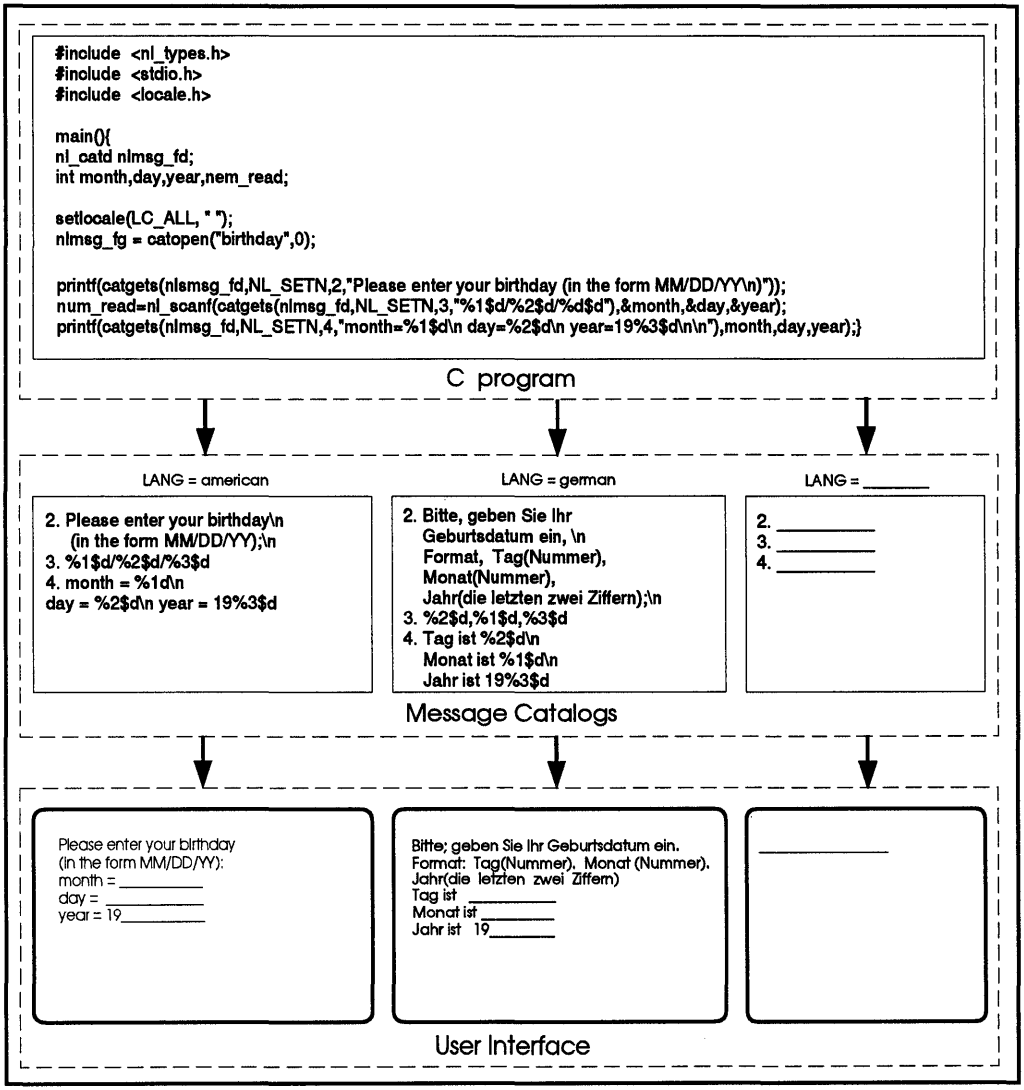

Figure 7-1. The Message Catalog System

Note

- The catalog descriptor `catd` is used by `catgets` and consequently must be accessible to every `catgets` call.
- Use the program name as the *name* argument.

Search Path and Naming Conventions

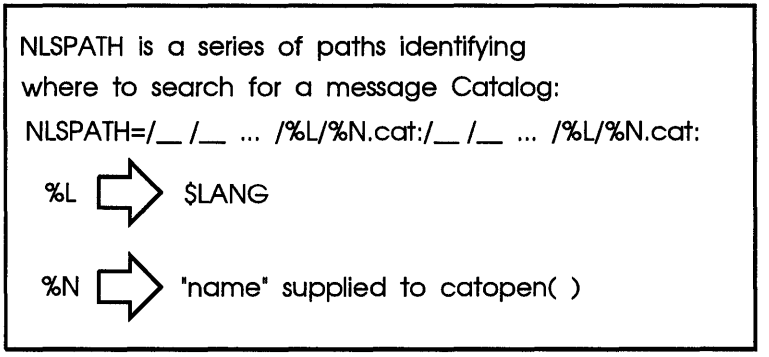

Figure 7-2. Naming and Locating Message Catalogs

The names of message catalogs and their location in the file system can vary from one system to another. Individual applications may choose to name or locate message catalogs according to their own special needs.

7 The flexibility to allow general location and naming of message catalogs is provided via the NLS environment variable `NLSPATH` which gives both the location of message catalogs and the naming conventions. You can define message catalog naming conventions by using substitution field descriptors that permit the use of run-time information. For example:

```
NLSPATH=/usr/local/lib/%L/%N.cat:./%N.cat
```

This specifies two paths, separated by `:`, to search for a message catalog. The metacharacter, `%`, in a search path introduces a substitution field descriptor, where `%N` is replaced by the *name* parameter passed to `catopen`, and `%L` is replaced by `$LANG`.

Thus, for the above value of `NLSPATH`, the call `catopen("prog", 0)` will first attempt to open `/usr/local/lib/$LANG/prog.cat`. Failing this, it will

attempt to open `./prog`. Note that if `LANG` is not set, the first path would be `/usr/local/lib/prog.cat` and would probably result in a failure to find a catalog.

If `catopen` can't find a message catalog with the path names specified in `NLSPATH`, it searches the default path:

```
/usr/lib/nls/%l/%t/%c/%N.cat
```

where: `%l` is replaced by the *language* element of `LANG`, `%t` is replaced by the *territory* element of `LANG`, and `%c` is replaced by the *codeset* element of `LANG`. This is summarized in the following table:

Table 7-1. Summary of NLSPATH Replacement Specifiers

Replacement Specifiers	Expansion by NLS
`%L`	Replaced by the value of `LANG`.
`%N`	Replaced by the *name* parameter passed to `catopen`.
`%l`	Replaced by the language element of `LANG`.
`%t`	Replaced by the territory element of `LANG`.
`%c`	Replaced by the codeset element of `LANG`.

For further details on `LANG` and `NLSPATH`, see *environ*(5) in the *HP-UX Reference*

7

Retrieving Messages

Once the message catalog is open, the program can retrieve messages from the catalog using:

```
catgets(catd, set_num, msg_num, def_str);
    ⋮
```

where *catd* is the catalog descriptor returned by `catopen`, *set_num* and *msg_num* identify the message to be retrieved, and *def_str* ("default string") is a string that is returned if the call fails. Ordinarily *def_str* is the `C` locale message.

To retrieve messages, `catgets` uses an internal buffer that is overwritten on each call. This is rarely a problem since a message is ordinarily used immediately by being printed or tested. However, see "Special Considerations for Messaging" later in this chapter.

Closing a Message Catalog

When the program no longer needs access to the message catalog, the catalog file should be closed. This can be done with the `catclose` call but it is generally simpler to let `exit` close the catalog file when the program terminates.

Default Messages

A program should make provision for the case when the message catalog is not available. This could happen, for example, if the file system containing the catalog is not mounted or if there is no catalog for the current language. Note that `catopen` does not take a default action if a catalog cannot be opened. Provisions for default messages must be made by the program. There are two general strategies for handling this situation:

- The "standard" method is to include the default message as the *def_str* in the `catgets` call. If the `catopen` call fails, it returns `(nl_catd)-1`, an invalid file descriptor. This subsequently causes `catgets` to fail and return *def_str*, the default message. This is the recommended method of handling default messages.
- Alternatively, you can use a default message catalog. Note that even the default message catalog may not be available (e.g., if the file system containing it were not mounted). Commands using this method should consider the probability of this situation for their application and plan accordingly. Applications that use this method often use error message numbers as the default string in `catgets` calls.

If a message catalog is missing, it is seldom useful to issue an error message unless it is reasonable to expect the catalog to be available. If a message catalog is missing and the catalog is critical to the successful execution of the program, it may be best to issue a message and terminate the program.

Compiling and Linking

There are no special requirements for compiling and linking. All messaging routines are in standard libraries and will be linked with the usual compile/link commands.

Creating a New Message Catalog

Creating a message catalog is a two step process:

1. Create the message text source file.
2. Use `gencat` to generate a message catalog from the message text source file.

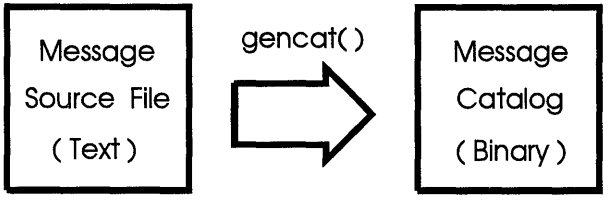

Figure 7-3. Using `gencat()` to Generate Message Catalog

The Message Text Source File

A message text source file contains the messages from the source program. Each message is numbered with the message number used in the corresponding `catgets` call.

A simple message catalog text file might be:

```
$ Comment:  a simple message text source file
1 text for message 1
2 text for message 2
```

A message consists of a message number followed by a single space or tab followed by the message text and terminated by a new-line. The message text is a `C` string, including spaces, tabs and \ (backslash) escapes, but without surrounding quotes. Message numbers are unsigned integers and must be in ascending order but need not be consecutive. A line beginning with $ followed by a single space or tab is treated as a comment. Note that comments in the message text source file are not saved in the message catalog created by `gencat`.

For a large or complex group of messages it may be useful to arrange the messages into groups called sets. Message sets allow the programmer to group similar messages together within a catalog. For example, one set might contain all prompts, and another set might contain all error messages. A set is introduced by a `$set` directive. Messages belong to the set specified by the most recently appearing `$set` directive. Like message numbers, set numbers are unsigned integers and must be in ascending order but need not be consecutive. Message numbers in different sets are independent.

A default set, `NL_SETD` is defined in `<nl_types.h>` for use in source programs. If a `$set` directive does not appear in the message text source file, messages will be assigned to set `NL_SETD`. Using the default set and directives in the same message text source file is not recommended.

A message text source file with sets might look like the following:

```
$set 100
$ user prompts
1 Text of message number 1
4 Text of message number 4
9 Text of message number 9
    ⋮
$set 200
$ error messages
1 Text of message number 1
3 Text of message number 3
    ⋮
```

7

To make leading or trailing blanks visible, the `$quote` directive can specify a quote symbol. For example:

```
$ show blanks
$quote "
1 "   leading blanks"
2 "trailing blanks  "
```

For more details on the format of the message text source file see *gencat*(1).

Compiling a Message Catalog

Once the message text source file is correct, a message catalog can be generated. For example, if `prog.msg` contains the messages for `prog.c`, then you would type the following:

```
gencat prog.cat prog.msg
```

This generates `prog.cat`, a message catalog for `prog.c`. This step is analogous to compiling the source program: the message text source file is "compiled" into a binary message catalog for use by the program at run-time.

An Example of Programming with Message Catalogs

To see how this all fits together, suppose `prog.c` is the standard sample program:

```
main()
{
printf("hello world\n");
}
```

When converted to use message catalogs, `prog.c` would look like this:

```
#include <nl_types.h>
main()
{
nl_catd catd;
catd = catopen("prog", 0);
printf(catgets(catd, NL_SETD, 1, "hello world\n"));
}
```

The message text source file would be:

```
$ message catalog for hello world
1 hello world\n
```

The program would be compiled as:

```
cc -o prog prog.c
```

and the message catalog would be generated as:

```
gencat prog.cat prog.msg
```

For this example,

- We have used "standard" default message handling: default messages are the default strings in `catgets` calls, and these will be returned as messages if `catopen` fails.
- The program name is also the message catalog name so that `catopen` will search the standard places when looking for a message catalog.
- The default set, `NL_SETD`, is used in the source program and the use of a set directive in the message text source file is omitted.

Special Considerations for Messaging

- Messages in variables require special treatment. For example, the message in:

```
char *msg = "message";
    :
printf(msg);
    :
```

would, given a "direct" conversion, result in:

```
char *msg = catgets(catd, set_num, msg_num, "message");
    :
printf(msg);
    :
```

7 This would generate a compile error. The required conversion is:

```
char *msg = "message";
    :
printf(catgets(catd, set_num, msg_num, msg));
    :
```

- Messages in arrays require somewhat more elaborate treatment. Before conversion, an original source might contain the following:

```
static char *msg_tbl[] = {
    "message 1",
    "message 2",
       :
    "message N"
```

```
    };
    :
printf(msg_tbl[i]);
    :
```

This would need conversion to:

```
printf(catgets(catd, set_num, msg_num, msg_tbl[i]));
    :
```

and *set_num*, *msg_num* and message index `i` must be synchronized. In particular, note that `msg_tbl[0]` is `message 1` and that `0` is not a valid message number.

- Multiple messages in a `printf` call might appear as:

```
printf("message 1", "message 2");
    ...
```

But, because `catgets` overwrites its message string on each call, these cannot be translated as:

```
printf(catgets(catd, set_num_1, msg_num_1, "message 1"));
catgets(catd, set_num_2, msg_num_2, "message 2"));

    :
```

For this situation it is necessary to copy one of the messages:

```
char *m1[N];
    :
strcpy(m1, catgets(catd, set_num_1, msg_num_1, "message 1"));
printf(m1, catgets(catd, set_num_2, msg_num_2, "message 2"));
    :
```

- Both `catgets` and `gencat` impose limits on the length of messages they can handle. These limits may make it necessary to compose a large message, such as a help screen, from several smaller messages. Nevertheless, realize that splitting a message must be done with care as it can impose serious difficulties on the translation process. If a message must be split, each part should still express a complete sentence or idea. For further information, see *catgets*(3C) and other references in the *HP-UX Reference*.

- The message system does not check to see that the correct catalog is used with a program. If an incorrect version of a message catalog is inadvertently installed, your program will issue messages but they will probably not make sense. You may wish to add validation messages that contains the program revision code and the locale so the program can validate the message catalog it uses. This could be done as follows:

```
    :

char *p_rev =            /* program revision */
     "$Revision: 1.4 $"; /* catgets 1 */
char *c_rev;             /* catalog revision */
char *p_loc =            /* program locale */
     "C";                /* catgets 2 */
char *c_loc;             /* catalog locale */

    :

c_rev = catgets(catd, NL_SETN, 1, p_rev);
if ( strcmp(c_rev, p_rev) != 0 ) {
    printf("program/message catalog revision mis-match\n");
    catd = (nl_catd)-1;
    }
p_loc = getenv("LANG");
c_loc = catgets(catd, NL_SETN, 2, p_loc);
if ( strcmp(c_loc, p_loc) != 0 ) {
    printf("program/message catalog locale mis-match\n");
    catd = (nl_catd)-1;
    }

    :
```

7

This example uses an *rcs*(1) **$Revision$** line (see discussion in *co*(1)) so that the revision code can be updated automatically. The special comments **/* catgets 1 */** and **/* catgets 2 */** enable **findmsg** to find the validation messages. See the discussion in the "Source Code Management" section of this chapter.

The message text source file for this program would contain:

```
1 $Revision: 1.4 $
2 C
```

Note that both of these messages are potential problems for someone attempting to localize the program. Message 1, the revision line must not be localized. Message 2, specifying the locale, must be localized but the

translation is not obvious to someone unfamiliar with the program. Comments in the message text source file won't help since they are not saved in the message catalog. See "Guidelines for Using Messaging" later in this chapter for a description of a "cookbook" to help the translator avoid errors.

Libraries with Messages

Library routines, as well as programs, can use message catalogs. For example, the C library routine *perror*(3C) uses a message catalog and can be used by a program that also uses a message catalog. All the considerations for programs apply to libraries. There are also some special considerations.

In general, the scope of variables of a library routine is restricted to the routine so that they do not conflict with the variables of the main program. The catalog descriptor must be declared so that there can be no conflict with the main program, since the main program may also use a message catalog.

Since a library routine might be called several times by a program, some consideration should be given to the way the message catalog file is opened. There are two general strategies:

- The easy strategy is to open the catalog when it is needed and close it after use. This uses a file descriptor only when it is needed.
- For cases in which the library routine is called frequently, it may be desirable to avoid multiple opens/closes of the catalog. This can be done with the following:

```
static nl_catd catd;
static int oflg = FALSE;
    :
if ( ! oflg ) {
    oflg = TRUE;
    catd = catopen(...);
    }
    :
catgets(catd, set_num, msg_num, def_str);
    :
```

Once open, the file descriptor remains in use for the remainder of the program. The catalog will be closed by `exit` at program termination. Note, however,

that this method cannot be used if **LANG** can change between calls to the routine.

Conversion of Existing Programs for NLS Messaging

The following diagram shows the process used to convert a non-internationalized program.

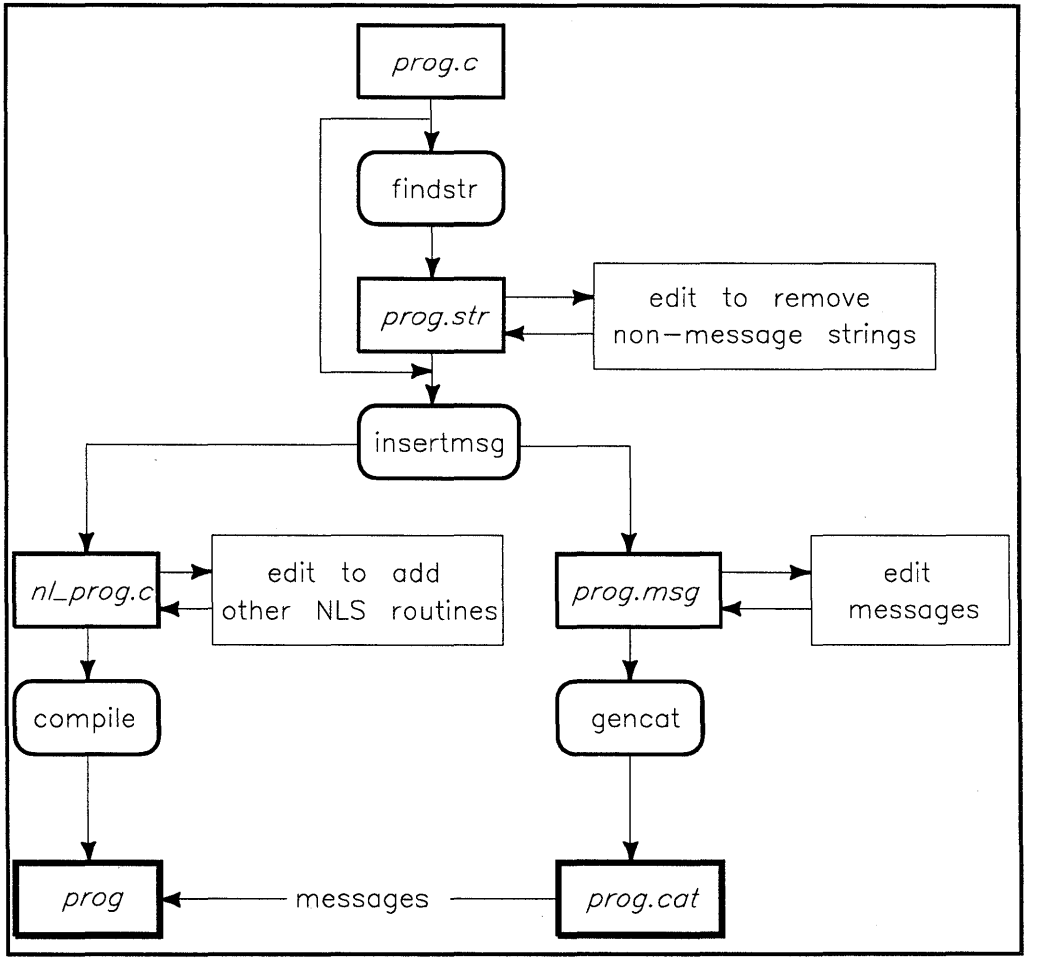

Figure 7-4. Converting a Non-Internationalized Program

The conversion of an existing program to use messages can be automated to a substantial degree. Consequently, even when writing a new program you may find it easier to write the program without messages and, when it is working, convert it to use messages.

The conversion process is:

1. Find all quoted strings in the source program. Such strings may be messages.
2. Review the list of quoted strings and remove any that are not messages.
3. Assign a message number to each message string and replace the string in the source program by an appropriate call to `catgets`.
4. Generate a message catalog from the numbered message strings.

HP-UX commands are available that make this process fairly easy.

Step 1. Finding Strings in a Program

Use the `findstr` routine to extract strings and write them to an external file.

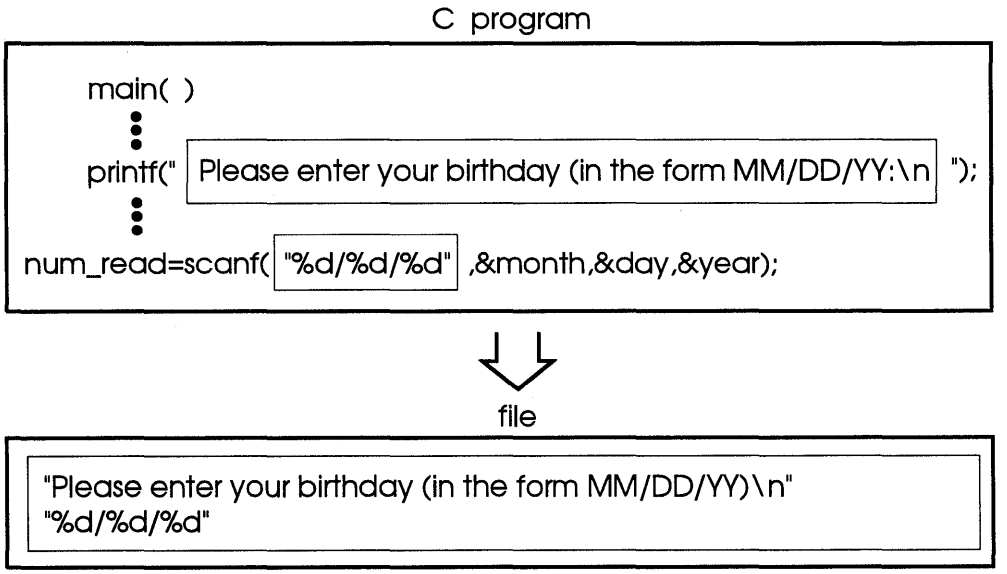

Figure 7-5. Using `findstr` to Locate String Constants

The command `findstr` will examine a `C` source program and find all string constants (other than those that appear in comments). These strings, along with their quotes, are written to standard output along with information indicating the position of the string in the source file. A typical use would be:

```
findstr prog.c > prog.str
```

The string file `prog.str` would now contain a copy of each string found in `prog.c`.

The `findstr` command expects the strings of your program to be syntactically correct with the quotes properly matched. To ensure that this is the case, it is a good policy to use `findstr` only on tested programs.

Step 2. Removing Non-Messages from the Strings

Most of these strings in the string file are messages and would need to be localized. Some of the strings, however, would never be localized. For example, the type specifier for `fopen` is a string such as `"r"` or `"w+"`. These strings are not messages and would not be localized. Some format strings would be localized but some would not. The string file must be reviewed and any entries for non-message strings should be removed.

When editing the string file, take care not to modify the location information for strings that are left in the file. Also, note that if the source file is changed, the string file may be invalidated and should be re-generated.

Step 3. Inserting catgets Calls

Once the string file contains only the strings that will need localization, you are ready to create a messaging version of your program.

This is done using the `insertmsg` command which takes care of a few administrative details:

- It assigns a message number to each string in the string file and writes the numbered messages to standard out in a format suitable for use by `gencat`. This is the message text source file for the program.
- It creates a copy of the source program in which each string identified in the string file is replaced by a `catgets` call with the assigned message number. The name of the new source program file is the name of the original source file with the prefix `nl_`.

A typical use would be:

```
insertmsg prog.str > prog.msg
```

If the `prog.str` file were created from the source file `prog.c`, the new source file would then be `nl_prog.c`.

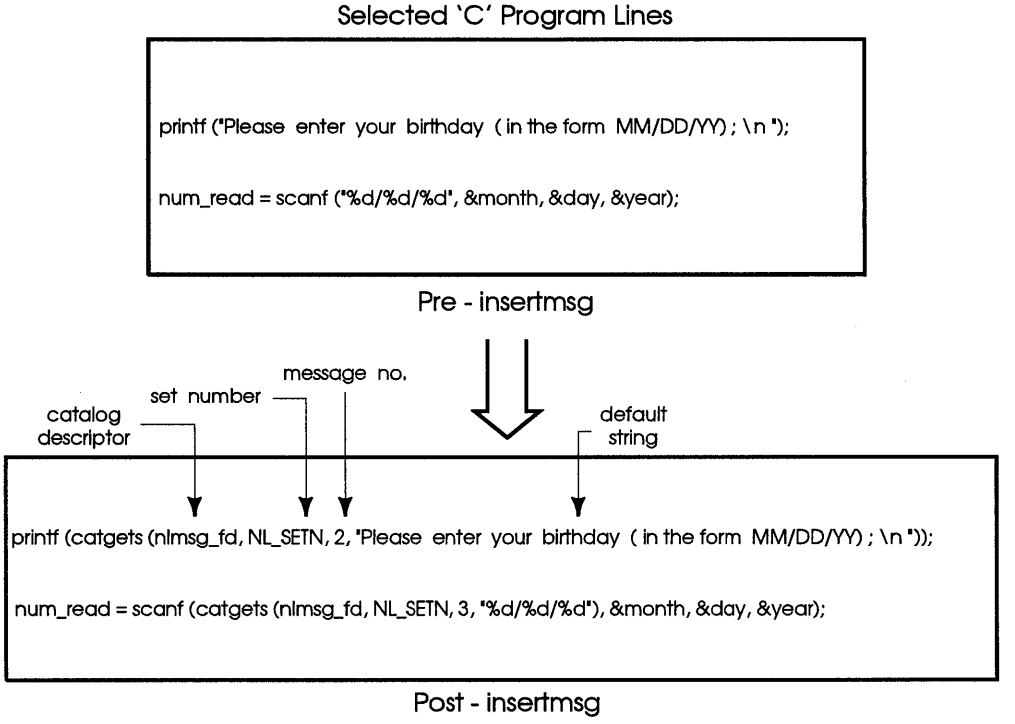

Figure 7-6. Using the `insertmsg` Command

Note

The `findstr` and the `insertmsg` command will not recognize the problem cases identified in the "Special Considerations" section in this chapter, and they will convert them without comment. Some of these conversions will draw a syntax error from the compiler; others will give incorrect results with no indication. The recommended strategy is to let the compiler find the syntax errors and to review the remaining conversions.

Step 4. Editing the Modified Source Program

Your new source program will need some minor editing before it can be used. A string such as:

```
... "string" ...
```

in the original source file, would have been changed to:

```
... catgets(catd, NL_SETN, msg_num,
          "string") ...
```

The *msg_num* was assigned by **insertmsg**. You must provide definitions for `catd` and `NL_SETN`. This can be done by adding the following lines near the beginning of the program:

```
#include <nl_types.h>
#define NL_SETN 1
     ⋮
nl_catd catd;
     ⋮
catd = catopen("name", 0);
     ⋮
```

The `catopen` call would ordinarily be part of the "standard" initialization. (See the section "Recommended Initialization" in this chapter for additional information.)

7 After these modifications, the new source program can be compiled and linked.

Step 5. Editing the Message Text Source File

For many cases, the message text source file, `prog.msg` in the above example, will need no modification. However, if you are using sets, appropriate `$set` directives must be inserted.

Step 6. Creating a Message Catalog

After any changes to the message text source file, the message catalog can be created using `gencat`. As in the earlier example:

```
gencat prog.cat prog.msg
```

Testing a Message Catalog

Once you have an executable program and a message catalog, you can test the program to be sure that it retrieves messages from the correct message catalog.

If you used the "standard" message initialization, the use of `NLSPATH` makes testing easy. For the following example, we assume:

- The executable program is named `prog`.
- The catalog is opened by `catopen("prog",0)`.
- The original messages are in `prog.msg`.
- The default value for `LANG` is null, i.e., unset.

The following script prepares test directories and catalogs:

Make a directory

```
mkdir ./french
```

Make a copy of the message text source file

```
cp prog.msg french.msg
```

Modify the messages to distinguish default messages from catalog messages

```
vi french.msg
   :
```

Generate a message catalog with the modified messages

```
gencat french/prog.cat french.msg
```

The catalog in directory `./french` is now ready for testing.

The following script tests the program for default messages and "french" messages.

Set `NLSPATH`:

```
NLSPATH=./%L/%N.cat ; export NLSPATH
echo $NLSPATH
```

Test the default messages:

```
echo LANG = $LANG
```

```
prog
```

Test the catalog messages:

```
LANG=french ; export LANG
echo LANG = $LANG
prog
```

Installing a Message Catalog

When you are satisfied that your messaging program correctly accesses its message catalog, it can be installed. See "Administering International Software" for more details.

Source Code Management

Following are some suggestions and comments on the management of messaging source programs.

Keeping nl_prog.c Files

There are two approaches regarding the modified source files:

- You can rename the `nl_*` files to the original names and keep the modified version as the source program. This is the more commonly used approach. It eliminates the need for reconversion but means the source files have the `catgets` calls in them and are more awkward to read.
- Or you can keep the original source files and convert them whenever they are modified. This eliminates the need to read the messaging statements but means the source files must be converted whenever a change is needed. This approach may be feasible only if editing of the string file and converted source files is minimal or can be automated.

Multi-file Programs

If your program consists of a number of files, the conversion process is only slightly more complex than for a single file. The `findstr`, `insertmsg`, and `gencat` commands all take multiple file input and perform appropriately. For more information, please see the appropriate pages in Section 1 of the *HP-UX Reference*.

Adding a Message to a Messaging Program

Once your program provides message catalog support, you may need to add a message to the program. If you keep the original version of the source program (without the message catalog calls), adding a new message is done simply by adding the message to the source program and converting the program as above.

If you keep the `nl_*` version of the source program (with the message catalog calls), adding a message means that you must assign a message number to the new message and this new number must not conflict with those already used in the message catalog. To assign new message numbers, you will need a list of existing message numbers. These are available from two places: in the message catalog and in the source program.

The **dumpmsg** command will list the messages in a message catalog:

```
dumpmsg prog.cat >prog.msg
```

If there are multiple versions of the program, be sure that the message catalog and the source program are for the same version.

The **findmsg** command will list the messages in a source program:

```
findmsg prog.c >prog.msg
```

This method is generally preferred since it ensures that the message text source file agrees with the source program. The messages found are the quoted strings in **catgets** calls in the source program. If a program uses messages in variables, you must add special comments to the source program so that **findmsg** can find these messages. For example, a message in a variable and its corresponding **catgets** call would look like the following:

```
     :
char *msg = "message"; /* catgets msg_num */
     :
printf(catgets(catd, NL_SETN, msg_num, "message"));
     :
```

Both of the message listing commands produce as output, a message text source file in a form suitable for input to **gencat**.

Once a message list is available, message numbers can be assigned to new messages and the source program appropriately modified with new **catgets** calls that have the newly assigned message numbers. The new messages can then be added to the message text source file and a new message catalog generated.

Although **gencat** can merge new messages into an existing message catalog, it is just as easy and less error prone to re-create the complete message catalog. Once the new **catgets** calls have been added to the source program, this can be done as the following:

Remove previous message catalog to preclude update.

```
rm -f prog.cat
```

Generate a message text source file with the new messages.

```
findmsg prog.c >prog.msg
```

Generate the new catalog.

```
gencat prog.cat prog.msg
```

List the new messages for review.

```
dumpmsg prog.cat
```

Using "make" Files

7

With the **.msg** and **.cat** file suffix conventions, it is possible to use **make** to automate message catalog creation. The following **make** file illustrates the procedure:

```
      :
SOURCE = prog.c sub.c ...
      :
all:    prog prog.cat
      :
prog.msg $(SOURCE)
        findmsg $(SOURCE) >$@
      :
.msg.cat:
        gencat $*.cat $*.msg
      :
```

The command:

```
make prog.msg
```

will generate the message text source file `prog.msg`. The command:

```
make prog.cat
```

will generate the message catalog `prog.cat` from the message text source file `prog.msg`. Also see the "make Example", in Appendix B for another illustration of this procedure.

Guidelines for Using Messaging

Here are some overall guidelines which you should keep in mind when programming for messages.

- Provide a *cookbook* for the translator which contains the numbered messages and, carefully separated (e.g, by brackets), any additional explanatory information or paraphrase they may need. A message that is obvious to you may be a mystery to a translator. You should assume that the translator:
  1. Has a different native language from yours.
  2. Is hundreds or thousands of kilometers away from you.
  3. Is doing the translation months or years after you finish the program.
- All text that needs to be localized should be put in the message catalog. This includes: prompts, help text, error messages, format strings, softkey definitions, and command names.
- Any text that will not be localized should not be put in the message catalog. Including unnecessary text will not affect the program behavior but it may be confusing to a translator.
- Provide a unique, unambiguous message for each situation. A single message in your own language may appear to cover several different situations. However, when the message is translated into another language, each different situation may require a different local language translation.
- Do not split messages into separate messages in a message catalog unless absolutely necessary. The messages could be untranslatable if the word order changes when translated. If messages absolutely must be split, do so in a

logical way. A large block of text, for example, should be split along sentence boundaries.

- Allow at least 60% extra space in text buffers and screen layouts to allow for text expansion when messages are translated. It may take more space to convey information in another language. Define these buffers in a localization cookbook or put them in comments at the beginning of each message catalog.
- Decide what to do if a message catalog cannot be found by your program. If the local language is vital to the operation of the program, you may want the program to issue a default error message and exit. If the local language is not vital to this part of your program, you might allow the program to continue to operate with a default language (such as `C`).

8

Advanced NLS Topics

Read this chapter if you are:

- ▷ A programmer or software developer who has special requirements
- ▷ Anyone in need of additional background information on NLS

This chapter covers the following:

- Character and string processing in more detail
- Special requirements for localizing
- Special situations for messaging

Codeset Conversion

If you need to transport data between systems that use different codesets, you will probably need to convert codesets. To assist this conversion, two codeset conversion tools are available.

- The `iconv` command operates on files and converts characters from one codeset to another. Conversion can be performed between HP codesets and a number of widely-used non-HP codesets. See *iconv*(1) in the *HP-UX Reference* for details.
- The `iconv` routines are intended for special situations not covered by the conversion command. Using these routines, it is possible to provide special treatment that may be needed in the conversion. See *iconv*(3C) in the *HP-UX Reference* for details.
- NLIO provides some other routines for Asian languages. Please refer to the appropriate NLIO manual for more information (see "Related HP-UX Manuals" in Chapter 1).

The Character Conversion Command—iconv(1)

The syntax for evoking the `iconv` command is:

```
iconv -f fromcode -t tocode [file...]
```

where:

fromcode	Identifies the source codeset.
tocode	Identifies the target codeset.
[file...]	Identifies input and output files, if any. If no arguments are specified standard input and standard output are used.

For more information or HP defined values for *fromcode* and *tocode* see *iconv*(1) in the *HP-UX Reference*.

Conversion Routines—iconv(3C)

When you convert codesets within an application you can avoid the overhead involved in a command line call to a systems routine by using library routines instead. These routines give you more control over the conversion process (for example, you can specify your own default values for unmappable characters). When you convert codesets using these routines, the first step is to determine if a conversion table is needed. If a table is required, you must determine its size so you can allocate enough memory. Use `iconvsize` to perform this function. The syntax of the routine is:

```
#include <iconv.h>

int iconvsize (tocode, fromcode)
char  *tocode;
char  *fromcode;
```

8

This routine evaluates the requirements for converting the codeset specified by *fromcode* to the codeset specified by *tocode*:

- If a conversion table is needed and exists, the size of the table in bytes is returned.
- If a table is needed, but the table is nonexistent, then `-1` is returned.
- If a conversion table is not needed, `0` is returned.

Once you determine the size of the conversion table (if it is required and exists), you must provide the necessary initialization. Use `iconvopen` to provide all initializations necessary for converting characters from the codeset specified by *fromcode* to the codeset specified by *tocode*. The initialization process includes:

- The retrieval of the necessary table. (It is your responsibility to allocate sufficient memory for the table based on the value returned by `iconvsize`).
- The global initialization of default values for unmappable characters within multi-byte codesets. Default characters are used with multi-byte characters since multi-byte codesets usually do not have the same number of characters; this makes a one-to-one mapping difficult. No default values are used with single-byte codesets.

The syntax of the routine is as follows:

```
iconvd iconvopen (tocode, fromcode, table, d1, d2)
char *tocode;
char *fromcode;
unsigned char *table;
int d1, d2;
```

Where:

tocode	Identifies the target codeset.
fromcode	Identifies the source codeset.
table	Points to the start of the conversion table if needed, otherwise points to null.
d1	Specifies a default character for a unmappable one-byte character (within a multi-byte codeset).
d2	Specifies a default character for an unmappable two-byte character.
`iconvd`	Acts as a conversion descriptor and function return value.

Two additional auxiliary routines are provided besides the conversion routines (which will be explained shortly). These are `iconvclose` and, the less frequently required, `iconvertlock`. `iconvclose` closes the conversion descriptor *cd*, freeing it up for a subsequent `iconvopen`. A non-negative

number is returned if the routine is successful; otherwise `-1` is returned. The syntax of this routine is:

```
int iconvclose(cd)
inconvd cd;
```

The routine `iconvlock` can be used to initialize lockshift information. This routine is required by X/Open (For more information see *iconv*(3C) in the *HP-UX Reference.*)

Once the preliminaries are performed (determination of table size and the necessary initialization) you are ready to convert your codesets. Three conversion routines are provided. The syntax of these routines is as follows:

```
int ICONV (cd, inchar, inbytesleft, outchar, outbytesleft)
iconvd cd;
unsigned char **inchar
int  *inbytesleft;
unsigned char  **outchar;
int *outbytesleft;

int ICONV1 (cd, to, from, buflen)
iconvd cd;
unsigned char  *to;
unsigned char *from;
int buflen;

int ICONV2 (cd, to, from, buflen)
iconvd cd;
unsigned  char *to;
unsigned  char *from;
int       buflen;
```

These routines perform as follows:

Table 8-1. Conversion Routines

Routine	Action
`ICONV`	Fetches a character from the buffer pointed to by *inchar* and converts it to the target codeset, placing it plus any lock-shift information in the buffer pointed to by *outchar*. The descriptor *cd* identifies the conversion to perform. *Inbytesleft* and *outbytesleft* point to the number of bytes left in the input and output buffers, respectively. While conversions are done from the input buffer to the output buffer, these variables are incremented or decremented to reflect the current status of each buffer.
`ICONV1`	`ICONV1` converts single-byte characters in *from* according to the conversion identified by *cd*, placing the result in *to*. Use this routine and the routine below when it is more efficient to handle single- and multi-byte information separately. This routine does not check for lock-shift information. *Buflen* specifies the number of bytes to convert.
`ICONV2`	Same as above, except `ICONV2` assumes *from* contains only double-byte characters.

For more information see *iconv*(3C) in the *HP_UX Reference*

Processing Right-to-Left Languages

Processing right-to-left languages requires the programmer to deal with issues of data directionality that are not ordinarily a concern.

Directionality refers to two properties of the text:

- The direction the language is naturally read.
- The order of characters in a file.

Mode can be:

- Latin: left-to-right.
- Non-Latin: right-to-left.

Order can be:

- Keyboard: the order in which the user enters keystrokes.
- Screen: the order in which characters are displayed.

Some codesets contain Latin and non-Latin characters so that it is possible to mix left-to-right and right-to-left text. If we use L*i* to indicate a Latin character, N*i* to indicate a non-Latin character, and *i* to indicate the order in which the character is typed, the mixed text:

```
N1 N2 L3 L4 N5 N6 L7 L8
```

entered on a terminal configured for right-to-left display would appear as:

```
L7 L8 N6 N5 L3 L4 N2 N1
```

For additional information on directionality, see *hpnls*(5) in the *HP-UX Reference*.

Two commands are available to manage data directionality. The command `forder` allows users with screen data to use programs that do not support screen order data. It converts the order of characters in a file from screen order to keyboard order, or from keyboard to screen order. For example, `sort` cannot sort screen order data. However, such data could be sorted by:

```
forder file1 |        # put in keyboard order for sort
sort |                # sort it
forder > file2        # put back in screen order
```

Data order and mode (Latin vs. non-Latin) information is specified by the **LANGOPTS** environment variable. To set the **LANGOPTS** environment variable using the Bourne or Korn Shell:

```
LANGOPTS=[mode][_order] ; export LANGOPTS
```

Where:

mode may be either **l** (for Latin) or **n** (for non-Latin). Non-Latin mode is assumed for values other than **l** and **n**.

order describes the data order of a file and may be either **k** (for keyboard) or **s** (for screen).

For further details on the **LANGOPTS** environment variable, see *environ*(5).

Since most printers are designed for printing left-to-right languages, printing right-to-left data requires special formatting. The command **nljust** provides this special formatting. It aligns such data with the right margin and composes the data in right-to-left print order. For example, **nljust** would typically be used as a filter with the **lp** and **pr** commands, such as in:

```
pr file | nljust - | lp
```

As with **forder**, **nljust** also gets mode and order information from the **LANGOPTS** variable.

For special situations that cannot be handled by data ordering commands, the routine **strord** converts between screen order and keyboard order and can be used to provide any special processing that may be needed. As a simplified example, consider a program that reads data in either keyboard or screen order, and writes it to a terminal in screen order. The relevant portions of the program are:

```
   :
#include <nl_types.h>
   :
char *lopts;
   :
lopts = getenv("LANGOPTS");       /* "m_o" m = mode, o = order */
   :
fscanf(..., src, ...);          /* read in current mode/order */
if ( lopts[2] == 'k' )            /* if order is keyboard order */
    strord(dst, src, lopts[0]);  /* re-order before write */
fprintf(..., dst, ...);         /* write data */
   :
```

For an extended example of right-to-left processing, see Appendix B, "Example of Internationalized Software", in this manual.

Locale Information

Locale information is available in various ways. The `locale` and `nlsinfo` command provides selected portions of information for a specified locale. Information is displayed in tabular form convenient for reference. The `localedef` command `-d` option provides all information for a specified locale. This information is displayed in `localedef` input format and may be used to define a new locale.

You can use the `locale` command to display information about the current locale or about available locales. Here are some examples that illustrate the use of locale. `LANG` was set to french for all examples.

The following example displays all items in the LC_TIME category:

```
% locale -ck LC_TIME
LC_TIME
abday="Dim";"Lun";"Mar";"Mer";"Jeu";"Ven";"Sam"
day="Dimanche";"Lundi";"Mardi";"Mercredi";"Jeudi"; \
"Vendredi";"Samedi"
abmon="janv";"févr";"mars";"avr";"mai";"juin"; "juil"; \
"août";"sept";"oct";"nov";"déc"
mon="janvier";"février";"mars";"avril";"mai";"juin"; \
"juillet";"août";"septembre";"octobre";"novembre";"décembre"
d_t_fmt="%A %.1d %B %Y %H:%M:%S"
d_fmt="%A %.1d %B %Y"
t_fmt="%H:%M:%S"
am_pm="";""
t_fmt_ampm="%H:%M:%S"
year_unit=""
mon_unit=""
day_unit=""
hour_unit=""
min_unit=""
sec_unit=""
era_d_fmt=""
era=""
```

This example displays the value for the mon item:

```
% locale -ck mon
LC_TIME
mon="janvier";"février";"mars";"avril";"mai";"juin"; \
"juillet";"août";"septembre";"octobre";"novembre";"décembre"
```

This example displays the values for LANG and the LC_* categories:

```
% locale
LANG=french
LC_CTYPE="french"
LC_COLLATE="french"
LC_MONETARY="french"
LC_NUMERIC="french"
LC_TIME="french"
LC_MESSAGES="french"
LC_ALL=
```

Programmatic access to information about the currently active locale is provided by three library routines. The `nl_langinfo` routine provides access to all locale information. The `localeconv` routine provides access to the locale information that pertains to numeric formatting. The `getlocale` routine provides access to `setlocale` status information. See *setlocale*(3C).

Initialization

The following sections provide more detailed information on:

- Special locales
- Special message catalogs
- Default message catalogs
- Programs that call `exec`

Special Locales

The `setlocale` routine can set individual categories to specific locale values. For example, to have a program run with French date and time conventions and with Spanish sorting conventions, the following calls would establish the desired locale:

```
#include <locale.h>
   :
setlocale(LC_TIME,"french");
setlocale(LC_COLLATE,"spanish");
```

This use, however, defeats the adaptive nature of the NLS routines and is not recommended. A preferred way to get the desired effect would be to use the "standard" initialization and to set the NLS environment variables when the program is run:

```
LC_TIME=french ; export LC_TIME
LC_COLLATE=spanish ; export LC_COLLATE
```

Special Message Catalogs

The `catopen` routine can specify a path for the message catalog, as in:

```
catd = catopen("/usr/special.cat", 0);
```

This use, however, defeats the generality of `catopen` and is not recommended. A preferred way to get the desired effect would be use the "standard" initialization:

```
catd = catopen("special", 0);
```

Then set the NLS environment variable when the program is run:

```
NLSPATH="/usr/%N.cat" ; export NLSPATH
```

Default Message Catalogs

The "standard" default message handling is to use the C locale messages as the default string in `catgets` calls. This ensures that the program will be able to issue messages even if there is no message catalog available.

If your application must access a C message catalog for the default messages, the following is suggested:

```
    :
if (!setlocale(LC_ALL, "")) {
    fputs("Warning! call to setlocale failed\n", stderr);
    fputs("Continuing processing using the \"C\" locale\n", stderr);
    catd = (nl_catd)-1;
    }
else
    catd = catopen("name", 0);
if (catd == (nl_catd)-1) {
    /* if necessary, user may save LANG at this point */
    putenv("LANG=C");
    /* try NLSPATH */
    catd = catopen("name", 0);
    /* if necessary, user may restore LANG at this point */
    if (catd == (nl_catd)-1)
        /* try hard-coded path */
        catd = catopen("/usr/lib/nls/C/name.cat", 0);
    }
    :
```

Programs That Call exec

For commands that `exec` other commands, we recommend that the first command call `setlocale`. If the call is unsuccessful, use `putenv` to reset all the NLS environment variables to ensure that the other commands don't repeat the unsuccessful `setlocale` call and issue additional error messages.

Messaging: printf/scanf Data Formatting

Messages that contain run-time data will often need to be rearranged for display in different locales. For example, the following statement displays the date in C locale format:

```
printf("%d/%d/%d\n", mo, dy, yr);
```

and would give the following result:

```
10/31/91
```

If this date were displayed in the U.K., the `english` locale, it would need to appear as:

```
31/10/91
```

which could be done with a statement such as:

```
printf("%d/%d/%d\n", dy, mo, yr);
```

This solution, however, requires a change to the source program: the order of the `printf` arguments must be changed.

To provide flexible formatting of data, the *printf*(3C) family of routines permits a conversion specification of the form `%`*n*`$` to indicate that conversion should be applied to the *n*th argument. For the C locale, we can use:

```
printf("%1$d/%2$d/%3$d\n", mo, dy, yr);
```

and for the `english` locale, we can use:

```
printf("%2$d/%1$d/%3$d\n", mo, dy, yr);
```

This solution leaves the order of the `printf` arguments unchanged. It does require a change to the format string but the format string can be treated as a message and modified as needed for each locale. So our solution becomes:

```
printf((catgets(catd,NL_SETN,17,"%1$d/%2$d/%3$d\n")), mo, dy, yr);
```

Then, the C locale message catalog would contain:

```
    :
17 %1$d/%2$d/%3$d\n
    :
```

And the **english** locale message catalog would contain:

```
    :
17  %2$d/%1$d/%3$d\n
    :
```

The *%n$* conversion specification is also available in the *scanf*(3C) family of routines.

A

Special Topics for HP's 16-bit Interfaces

Read this appendix if you are:

- ▷ Maintaining an existing application with HP's 16-bit interfaces
- ▷ Porting an application that uses HP's 16-bit interfaces

The 16-bit macros described in *nl_tools_16*(3c) are HP proprietary and are not portable to other vendor's platforms. Thus, we recommend you use the WPI interfaces, instead. We include information about the 16-bit interfaces to enable programmers to maintain existing applications or port existing applications to use the WPI interfaces.

Aspects of Program Design

Designing an international program with NLS is usually a straightforward process. Nevertheless, there are a number of special considerations. For example, make sure you reserve enough space in arrays and other data structures to accommodate the needs of all your users. Since an international program supports character sets that contain multi-byte characters, the number of characters in a string is no longer equivalent to the number of bytes. You must allocate additional space to accommodate the larger character size in the codesets for certain languages.

The existence of multi-byte characters also affects your code in a number of other areas:

- **Scanning for a character match** - Suppose that you are writing an assembler and your code searches for the character ":". With the existence of two-byte characters, it is no longer possible to simply scan a data stream looking for a byte whose bit pattern matches that of ASCII ":". The byte that you match may be the second half of a two-byte character. Your program must search for *character matches*, not byte matches.
- **Reading characters** - Suppose that you allocate an 80 byte buffer for reading data. You must be careful to truncate excess input on character boundaries. Suppose that the 80th byte read into the buffer is the first half of a two byte character. When the contents of another buffer are appended, the first byte of the appended buffer will interpreted as the second half of the two byte character. Thus, data is corrupted.

Code Sets

One objective of international program design is to create an application that is codeset independent. To create a program that is sufficiently robust to accept any kind of codeset, you must know how data is represented in different languages and the potential problems you can encounter.

As a UNIX user, you are probably familiar with ASCII, the 7-bit codeset used to support American English. All codesets supporting the diverse languages of international users are supersets of the familiar ASCII. This ensures that these codesets can communicate with the operating system, utilities, and applications which have a dependency on ASCII.

The ISO 8859-1 and Roman8 codesets support Western European languages. These 8-bit codesets support an additional 128 character codes beyond those of ASCII. While this extension of the ASCII character set meets the needs of Western European users, it is not large enough to support languages such as Arabic and Greek that have alphabets completely different from those used in Western Europe or the U.S. For these languages, other 8-bit codesets have been designed such as ARABIC8 and GREEK8. ISO 8859-2 and ISO 8859-5 are used for supporting Eastern European languages such as Polish and Russian (a complete list of codesets and the languages they support is provided in Appendix E).

8-bit codesets provide support to international users who speak and write phonetic languages. A single byte, however, is not sufficient to represent the symbols of users whose language is ideographic (for example, Traditional Chinese which contains over 50,000 distinct ideographs). To provide for these users, codesets that support multi-byte characters were introduced.

With the introduction of encoding schemes with multi-byte characters a problem arose. Because users who read and write ideographics still need ASCII (for communicating with the operating system and backwards compatibility), it becomes possible to have a data stream consisting of a mixture of one and two byte characters. The resulting problem is one of character interpretation: How can a program interpret characters correctly, distinguishing between single and multi-byte characters? A number of solutions to this problem have been designed.

A group of 2-byte codesets were developed that adhere to a common definition for interpreting a byte stream called HP15. All codesets that adhere to the

HP15 definition use a set of bytes with the high bit set to differentiate between 8 and 16 bit data. If the high order bit is zero, then the byte represents a one-byte ASCII character.

Otherwise, the current byte may represent the first byte of a two-byte character or a one-byte non-ASCII character (correct interpretation is provided through system tables). Since the high order bit acts as a flag bit, only 15 bits remain for data, hence, the term HP15. While this representation allows 8-bit data to be distinguished from multi-byte data, there is one restriction. A byte stream must be examined sequentially. For an "arbitrary" byte it is not possible to tell if the byte represents a single byte-character or the second half of a two-byte character.

In addition to HP15, NLS defines support for EUC (for Extended UNIX Code), an encoding scheme defined by AT&T UNIX Pacific, Ltd. The EUC encoding scheme provides a general template for defining codesets and interpreting a byte stream. EUC allows four distinct code sets (CS0 through CS3) to co-exist in a byte stream by restricting the legal bit patterns for each set to a specific template. Under the EUC encoding scheme ASCII characters are uniquely identified by a most significant bit of "0" and occupy CS0; the EUC template restricts the most significant bit of *all* non-ASCII characters to a "1", even the second byte of two-byte characters.

Currently HP-UX NLS supports japanese.euc (UJIS), a codeset that supports the Japanese language. This support includes the 2 byte codesets: CS1, and CS2.

Data Integrity

Data integrity means that in processing codeset data, the data must not be corrupted. For single-byte codesets, the 8th bit must be preserved; it must not be stripped or used by the program. For multi-byte codesets, single-byte characters must be correctly distinguished from multi-byte characters.

HP's multi-byte codesets utilize an encoding scheme in which the single-byte character codes for ASCII can be intermixed with the two-byte character codes used to represent ideograms. In these codesets, it is possible for the second byte of a two-byte character to have the same value as an ASCII character.

For an arbitrary byte, it is not possible to know if the byte is a single-byte character or the second byte of a multi-byte character. This is the "byte redefinition" problem in which the second byte of a multi-byte character may be incorrectly interpreted as a one-byte character.

Even for those encoding schemes such as EUC that avoid much of the byte-redefinition problem, an efficient way of dividing a byte stream along character boundaries is needed.

To aid in processing multi-byte codesets and avoid the byte redefinition problem, there are a number of routines available to the programmer. These routines are described in the sections that follow.

Programming with Multi-byte Characters

For dealing with HP's multi-byte codesets, see *nl_tools_16*(3C) in the *HP-UX Reference* which describes a set of byte-status macros: `FIRSTof2`, `SECof2`, `BYTE_STATUS`, and `C_COLWIDTH`. The syntax for the routines is:

```
#include <nl_type.h>
int c, laststatus;

FIRSTof2(c);
SECof2(c);
BYTE_STATUS(c,laststatus);
C_COLWIDTH(c);
```

Table A-1. Multi-byte Macros

FIRSTof2	Evaluates a byte c and returns a non-zero value if c may be the first byte of a 2-byte character according to the loaded NLS environment, and zero if it cannot.
SECof2	Evaluates a byte c and returns a non-zero value if c may be the second byte of a 2-byte character according to the loaded NLS environment, and zero if it cannot.
BYTE_STATUS	Reports whether a byte c represents a single-byte character, the first byte of a 2-byte character, or the second byte of a 2-byte character based on the value of the current byte in c and the status of the previous byte interpreted in **laststatus** as returned by the last call to BYTE_STATUS.
C_COLWIDTH	Evaluates a byte which is assumed to be either a one byte character, or the first byte of a 2-byte character, and returns the number of columns the character would occupy on a terminal display.

Note These macros are undefined for values of c less than -1 or greater than 255. Also note, these are Hewlett-Packard proprietary routines; do not use them if portability is an issue (Use mblen described in "WPI Interfaces" in Chapter 6 instead).

The following example illustrates how a program may be adapted to handle multi-byte as well as single-byte characters. The program is offered in two versions: the first works only for single-byte codesets, while the second is codeset independent.

Version #1 (Single-Byte Codesets)

The following program folds characters strings. A field size is specified defining the WIDTH for text displayed on the screen. As a given string is printed, any character whose display would fall outside the designated text region is "folded" onto the next line. For instance, if the WIDTH = 5 and the string is 0123456789, the result is:

```
01234
56789
```

The following program works only for single-byte characters because two-byte characters can potentially be split between bytes and treated as single-byte codes.

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

#define WIDTH 5

main(argc, argv)
int  argc;
char **argv;
{
   unsigned char output_array[WIDTH], *output_ptr, *input_ptr;
   int c, counter;

   /* try to set the locale, even through it does not affect the program
      because there are no calls to routines that are affected by the locale */
   if (! setlocale(LC_ALL,"")) {
      fprintf(stderr,"error: cannot set locale\n");
   }
   input_ptr = argv[1];            /* initialize the pointers and counter */
   counter = 0;
   output_ptr = output_array;

   while (c = *input_ptr++) {     /* get a char until end of string */
     if (counter >= WIDTH {       /* if output array is complete, process it */

       *output_ptr = '\0';        /* null terminate the output array */
       puts(output_array);        /* print the output array */
       counter = 0;               /* reset counter */
       output_ptr = output_array; /* reset output pointer to beginning
                                     of array */
     }
     counter++;                   /* if output array not complete
                                     increment counter */
     *output_ptr++ = c;           /* set area pointed to by output_ptr
               to c, then increment output_ptr */
   }
   *output_ptr = '\0';            /* null terminate the output array */
   puts(output_array);            /* print the output array */
}
```

Version #2 (Code Set Independent)

The following program also folds character strings to fit within a designated text region. By using C_COLWIDTH, FIRSTof2, and SECof2 macros, the program now operates correctly on single-byte and multi-byte data. This program works for both HP15 and the EUC encoding schemes.

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

#define WIDTH 5

main(argc, argv)
int argc;
char **argv;
{
   /* allocate additional array storage to accommodate multi-byte data */
   unsigned char output_array[WIDTH * 2 + 1], *output_ptr, *input_ptr;
   int c, counter, screen_size;

   if (!setlocale(LC_ALL,"")) {        /* set correct locale */
     fprintf(stderr, "error: cannot set locale\n");
   }
   input_ptr = argv[1];                /* set input_ptr to beginning
                                          of the string to be folded */
   counter = 0                         /* initialize character counter */
   output_ptr = output_array           /* set output_ptr to beginning
                                          of the output array */
   while (c = *input_ptr++) {          /* get a char until end of string */
      screen_size = C_COLWIDTH(c);     /* determine display column width
               of current character */
      if (counter + screen_size > WIDTH) {  /* if output array is complete,
                  process it */
         output_ptr = '\0';             /* null terminate the output
                array */
         puts(output_array);            /* print the output array */
         counter = 0;                   /* reset counter */
         output_ptr = output_array      /* reset output pointer to
                beginning of array */
      }
      counter += screen_size;           /* if output array not complete,
               increment counter */
      *output_ptr++ = c;                /* set area pointed to by output_ptr
                to c, then increment output_ptr */
      if (FIRSTof2(c) & SECof2(*input_ptr)) /* if valid two-byte character, */
        *output_ptr++ = *input_ptr++;       /* put second byte in output
                                               array and increment input and
```

A

```
                                        output ptr's */
    }
    output_ptr = '\0';                  /* null terminate the output array */
    puts(output_array);                 /*print the output array */
}
```

Note Although these macros seem transparent, observe that they cannot determine byte status for an arbitrary byte within a string. In general, multi-byte strings must be examined sequentially from the beginning.

See *nl_tools_16*(3C) in the *HP-UX Reference* for more information on programming with multi-byte characters.

Conversion of Existing Programs

When internationalizing an existing program, use the following two steps to preserve data integrity.

1. Convert program to handle 8-bit codesets.
2. Convert program to handle multi-byte codesets.

Be careful when converting software to handle 8-bit codesets. Some programs use the 8th bit as a flag to indicate special treatment of the remaining 7-bit character. In general, it may not be easy to determine whether a program does this. Programs that use or remove the 8th bit must be changed. If the 8th bit is used for data, you must put the 8th bit in a new data structure, and you may need to design a new algorithm to access the new data structure.

Once a program is correct for 8-bit data, the conversion to multi-byte data is usually straightforward. No structural changes are needed, but you must add proper handling of multi-byte characters. There are exceptions to this, however. For instance, backwards parsing is not compatible with mixed sized characters in a string because evaluation of character boundaries depends on sequential examination of a byte stream. If your program parses backwards for a character match, it must be restructured to accommodate multi-byte as well as 8-bit data.

Also, you can not test multi-byte data with routines such as `isprint`. In general, it is necessary to examine each instance of byte processing to determine whether special handling of multi-byte data is needed.

In working with multi-byte data, 16-bit `curses` is available to facilitate display design. More information is available in *curses*(3C) in the *HP-UX Reference* and in the section on Curses in the *Terminal Control User's Guide*.

Guidelines for Processing Multi-byte Data

- Do not search for a single-byte character in multiple-byte data on a byte-by-byte basis. A given byte may represent either a single-byte character or part of a multiple-byte character. Either test the data byte-by-byte while scanning to identify the multiple-byte characters, or convert the data to a form where all characters have the same width, or use multiple-byte character intrinsics. The first two techniques are generally used in C, the third technique is generally used in other programming languages.
- Do not *back-scan*, truncate or substitute multiple-byte data byte-by-byte. This is an extension of the rule above. Backscanning multiple-byte data is difficult; this should be taken into account when designing algorithms. Similarly, if a string is truncated be sure that a multiple-byte character is not *split* at the end of the truncated string. Extra care must also be taken if multiple-byte characters are being substituted for single-byte characters, or vice-versa.

B

Example of Internationalized Software

Example Program Using NLS Routines - rtlcat

The following program is used to illustrate several internationalization features including:

- message catalogs
- *setlocale*(3c) routines
- right-to-left processing
- some multi-byte programming in the `get_basename` section

Program Syntax:

```
rtlcat [options] [files ... ]
```

Where the value for *options* can be:

-l force file mode to Latin.

-n force file mode to non-Latin.

-k force file order to keyboard.

-s force file order to screen.

This program does a right-to-left concatenation (`cat`). It reads the concatenation of input files (or standard input if none are given) and displays the input on standard output. If `-` appears as an input file name, `rtlcat` reads standard input at that point. You can use `--` to delimit the end of options.

The text orientation (mode) of a file can be right-to-left (non-Latin) or left-to-right (Latin). This text orientation can affect the way data is arranged in the file. The data arrangements that result are called screen order and keyboard order.

`rtlcat` determines the mode and order of the input files and the terminal. The file mode and order comes from the **LANGOPTS** environment variable (*environ*(5)). The terminal mode and order are obtained from the primary and secondary status bytes that result when the terminal is asked about its alpha-numeric capabilities. This inquiry is done only on HP 150 and HP 2392 terminals. `rtlcat` assumes the terminal is the stdout device.

If the input file mode and order and the terminal mode and order are the same, then a simple copy is done. If the input file order and the terminal order are different but their modes are the same, then the input file data is rearranged by *strord*(3c) so it displays properly on the terminal screen. If the input file mode and the terminal mode are different, `rtlcat` simply stops with an error message. It is not defined what a non-Latin file should look like when it is displayed on a terminal configured for Latin mode (or vice-versa).

Include Files:

```
#include &<stdio.h>  /* input - output */
#include &<string.h>  /* string function declarations */
#include &<varargs.h>  /* variable arguments */
#include &<termio.h>  /* for ioctl call */
#include &<nl_types.h>  /* for nl_catd */
#include &<nl_ctype.h>  /* for ADVANCE */
#include &<locale.h>  /* for setlocale */
#include &<langinfo.h>  /* for nl_langinfo */
```

External Declarations:

```
extern nl_catd catopen(); /* open message catalog */
extern char *catgets();  /* get message from catalog */
extern int catopen();  /* close message catalog */
extern char *_errlocale(); /* get bad locale settings */
extern void perror();  /* system error messages */
extern void exit();  /* leave */
extern int optind;  /* argv index of next arg */
extern int opterr;  /* error message indicator */
extern int errno;  /* error number */
extern int sys_nerr;  /* max error number */
extern char *getenv();  /* get environment variable */
extern char *strord();  /* change data order */
```

Forward References:

```
extern void Perror();  /* local system print error message */
extern void error();  /* local system error message */
extern char *get_basename(); /* get basename of command name */
extern int copy();  /* copy file */
extern int reorder();  /* rearrange input file data */
```

General Constants:

```
#define WARNING  0 /* warning error message */
#define FATAL  1 /* fatal error message */
#define GOOD  0 /* successful return value */
#define BAD  -1 /* unsuccessful return value */
#define TRUE  1 /* boolean true */
#define FALSE  0 /* boolean false */
```

Limits:

```
#define MAX_ERR  256 /* max Perror message length */
#define MAX_TBUF 128 /* max tbuf length */
#define MAX_LINE 1024 /* max input line length */
```

Right-to-Left Terminal Constants:

```
#define an_cap  "\033*s-1^" /* request alpha-numeric capabilities */
#define sec_status "\033~"  /* secondary status */
#define on_straps "\033&&s1g1H" /* strap G && H on -- no handshake */
#define off_straps "\033&&s0g0H" /* strap G && H off -- D1 */

#define DISPLAY  2  /* alpha-num display byte */
#define ORDER  0x10  /* alpha-num display ordering bit */
#define RTL_SEC  8  /* 2nd status byte 13 */
#define MODE  0x08  /* 2nd status mode bit */
```

Error Message Numbers:

```
#define NL_SETN  1 /* message catalog set number */
#define BAD_USAGE 1 /* usage error message */
#define NOT_RTL_LANG 2 /* not a right-to-left language */
#define NOT_RTL_TERM 3 /* not a right-to-left terminal */
#define BAD_MODE 4 /* terminal/file mode disagreement */
```

Error Message Strings:

```
static char *Message[] = {
 "usage: %s [-lnks] [files ... ]\n",   /* catgets 1 */
 "\"%s\" not a right-to-left language\n",  /* catgets 2 */
 "\"%s\" not a right-to-left terminal\n",  /* catgets 3 */
 "mode of terminal and mode of file do not agree\n", /* catgets 4 */
};
```

Types:

```
typedef int (*PFI) ();  /* ptr to function returning int type */
```

Global Variables:

```
static char *Progname;  /* program name */
static char **Filename;  /* ptr to ptr to current file name */
static FILE *Input = stdin; /* input file pointer (assume stdin) */
static PFI Process;  /* routine to do the process */
static nl_catd Catd;  /* message catalog descriptor */
static nl_mode File_mode; /* mode of file (Latin or Non-Latin) */
```

Main Program:

```
/*
*****************************************************************************
** main()
**
** description:
** driver routine for program
**
** assumptions:
** all input come from stdin or named files
** all output goes to stdout
** all errors go to stderr
** the terminal screen is the stdout device
** mode and order of the input files is given in LANGOPTS
**
** global variables:
** Input: FILE pointer to the current input file
** Filename: ptr to ptr to current file name
**
** return value:
** 0: everything went ok
** -1: had some trouble
*****************************************************************************
```

```
*/

main(argc, argv)
int argc;   /* initial argument count */
char **argv;   /* ptr to ptr to first program argument */
{

 /* assume a successful return  value*/
 register int retval = GOOD;

 /* initialize, parse cmd line options, get input files, etc. */
 if (start( argc, argv) == BAD) {
  retval = BAD;
 }

 /* open and process input files one at a time */

 for ( ; *Filename ; Filename++) {

  /* open input file and get next if can't open */
  if (! strcmp( *Filename, "-")) {
   Input = stdin;
  }
  else if (! (Input = fopen (*Filename, "r"))) {
   Perror( "fopen");
   retval = BAD;
   continue;
  }
  /* process the file */
  if ((*Process)( ) == BAD) {
   retval = BAD;
  }

  /* close input file unless it's stdin */
  if (Input != stdin) {
   if (fclose( Input) == EOF) {
    Perror( "fclose");
    retval = BAD;
   }
  }
 }

 /* end the program */
 if (finish( ) == BAD) {
  retval = BAD;
 }
 return retval;
}
```

```
/*
*****************************************************************************
** start()
**
** description:
** set up language tables
** open message catalogs
** parse command line
** set up global variables
**
** global variables:
** Catd: nl_catd message catalog descriptor
** Progname: char pointer to the program name
** Filename: pointer to pointer to current file name
** File_mode: mode (Latin or Non-Latin) of the current input file
**
** return value:
** 0: everything went ok
** -1: had some trouble
*****************************************************************************
*/

static int
start( argc, argv)
int argc;   /* current argument count */
char **argv;   /* ptr to ptr to current argument */
{
 nl_mode term_mode; /* mode of terminal (Latin-Non-Latin */
 nl_order term_order; /* order terminal (Key-Screen) */
 nl_order file_order; /* order of file (Key-Screen) */
 char *termname;  /* terminal name from TERM */
 char *lopts;  /* language options from LANGOPTS */
 int optchar;  /* option character for getopts(3c) */
 static char *deffiles[] = { "-", (char*) NULL };
    /* default input file name */

 /* get the program base name in case it is renamed via ln(1) */
 Progname = get_basename( *argv);

 /* get locale & initialize environment table */
 if (!setlocale( LC_ALL, "")) {
  /* bad initialization */
  (void) fputs( _errlocale(), stderr);
  Catd = (nl_catd) -1;
  (void) putenv( "LANG="); /* for perror */
 }
 else {
  /* good initialization: open message catalog,
     ... use hardcoded name for first parameter,
```

```
    ... keep on going if it isn't there */
 Catd = catopen( "rtlcat", 0);
}

/* get file mode and order from LANGOPTS */
if(*(lopts = getenv( "LANGOPTS")) == '\0') {
 /* if not set assume Non-Latin mode, keyboard order */
 lopts = "n_k";
}
/* and do a lazy parse */
File_mode = lopts[0] == 'l' ? NL_LATIN : NL_NONLATIN;
file_order = lopts[2] == 'k' ? NL_KEY : NL_SCREEN;

/* parse command line options
   ... and possibly override file mode and order */
opterr = 0;  /* disable getopt error message */
while ((optchar = getopt( argc, argv, "lnks")) != EOF) {
 switch (optchar) {
 case 'l': /* force latin mode */
  File_mode = NL_LATIN;
  break;
 case 'n': /* force non-latin mode */
  File_mode = NL_NONLATIN;
  break;
 case 'k': /* force keyboard order */
  file_order = NL_KEY;
  break;
 case 's': /* force screen order */
  file_order = NL_SCREEN;
  break;
 case '?': /* unrecognized option */
  error( FATAL, BAD_USAGE, Progname);
 }
}

/* initialize process routine */

if (strcmp( nl_langinfo( DIRECTION) , "1")) {
 /* do not have a right-to-left language:
       ... print a warning and do a copy */
 char *langname;
 if(*(langname = getenv( "LANG")) == '\0') {
  /* if not set assume C language */
  langname = "C";
 }
 error( WARNING, NOT_RTL_LANG, langname);
 Process = copy;
}
else if (! rtl_term( &term_mode, &term_order, &termname)) {
```

```
 /* do not have a right-to-left terminal:
    ... print a warning and do a copy */
 error( WARNING, NOT_RTL_TERM, termname);
 Process = copy;
}
else if ((File_mode == term_mode) && (file_order == term_order)) {
 /* mode the same, order the same: a regular copy */
 Process = copy;
}
else if ((File_mode == term_mode) && (file_order != term_order)) {
 /* mode the same, order different: must change the order */
 Process = reorder;
}
else {
 /* Currently it is undefined what should happen when
    .. the file mode and the terminal mode are different. */
 error( FATAL, BAD_MODE);
}

/* set up input file arguments */
Filename = ((argc - optind) < 1) ? deffiles : argv + optind ;

return GOOD;
}
/*
*****************************************************************************
** finish()
**
** description:
** get ready to leave: close message catalogs
**
** global variables:
** Catd: nl_catd message catalog descriptor
**
** return value:
** 0: everything went ok
** -1: had some trouble
*****************************************************************************
*/

static int
finish()
{
 /* close the message catalog
  ... and do not complain about a missing catalog */
 (void) catclose( Catd);

 return GOOD;
}
```

```
/*
*****************************************************************************
** copy()
**
** description:
** Input file and terminal have the same mode and the same order.
** Just copy it to stdout.
**
** global variables:
** Input: FILE pointer to the current input file
**
** return value:
** 0: everything went ok
** -1: had some trouble
*****************************************************************************
*/

static int
copy()
{
 char line[MAX_LINE];

 while ((fgets( line, MAX_LINE, Input)) != NULL) {
  if (fputs( line, stdout) == EOF) {
   Perror( "fputs");
   return BAD;
  }
 }
 return GOOD;
}

/*
*****************************************************************************
** reorder()
**
** description:
** Input file and terminal have the same mode but the order is
** different. Rearrange the input file line with strord(3c) and
** copy it to stdout.
**
** global variables:
** Input: FILE pointer to the current input file
** File_mode: mode (Latin or Non-Latin) of the current input file
**
** return value:
** 0: everything went ok
** -1: had some trouble
*****************************************************************************
*/
```

```
static int
reorder()
{
 char line[MAX_LINE];
 char new_line[MAX_LINE];

 while((fgets( line, MAX_LINE, Input)) != NULL) {
  if (fputs( strord( new_line, line, File_mode), stdout)
        == EOF) {
   Perror( "fputs");
   return BAD;
  }
 }
 return GOOD;
}

/*
*************************************************************************
** Perror()
**
** description:
** set up string with program name and the failed routine name
** display system error message on stderr using perror(3)
**
** assumption:
** perror string before the colon will not exceed MAX_ERR
**
** global variables:
** Progname: char pointer to the program name
**
** return value:
** no return value
*************************************************************************
*/

/* VARARGS 1 */

static void
Perror( rname)
char *rname;   /* bad routine name */
{
 char pstr[MAX_ERR]; /* perror string before the colon */

 /* set up perror string */
 (void) sprintf( pstr, "%s (%s)", Progname, rname);

 /* print the system message or errno */
 if (errno > 0  &&  errno < sys_nerr) {
  perror( pstr);
```

```
 }
 else {
  (void) fprintf( stderr, "%s: errno = %d\n", pstr, errno);
 }
}

/*
*****************************************************************************
** error()
**
** description:
** display error message on stderr and leave if fatal
** get message from a message catalog (catgets(3c))
**
** assumptions:
** all errors go to stderr
**
** global variables:
** Progname: char pointer to the program name
** Message: array of char pointers to format string messages
** Catd: message catalog descriptor
**
** return value:
** no return value
*****************************************************************************
*/

/* VARARGS 2 */

static void
error( fatal, num, va_alist)
int fatal;   /* Warning or Fatal error */
int num;   /* message number */
va_dcl    /* optional arguments */
{
 register char *fmt; /* points to format string */
 va_list args;  /* points to optional argument list */

 /* set up the optional argument list */
 va_start( args);

 /* sync stdout with stderr */
 if (fflush( stdout) == EOF) {
  Perror( "fflush");
 }

 /* get the message format string */
 fmt = catgets( Catd, NL_SETN, num, Message[num-1]);
```

```
 /* print the program name on stderr */
 if (fprintf( stderr, "%s: ", Progname) < 0) {
  Perror( "fprintf");
 }

 /* print the error message on stderr */
 if (vfprintf( stderr, fmt, args) < 0) {
  Perror( "vfprintf");
 }

 /* close down the optional argument list */
 va_end( args);

 /* leave if a fatal error */
 if (fatal) {
  (void) finish( );
  if (fclose( Input) == EOF) {
   Perror( "fclose");
  }
  exit( BAD);
 }
}

/*
*****************************************************************************
** get_basename()
**
** description:
** get the basename of the command
**
** assumptions:
** the command name may have multi-byte characters
**
** return value:
** ptr to start of base name
*****************************************************************************
*/

static char *
get_basename( p)
char *p;   /* ptr to start of command name */
{
 char *slash;  /* pointer to char after slash */

 for (slash = p ; *p ; ADVANCE( p)) {
  if (CHARAT( p) == '/') {
   slash = p + 1;
  }
 }
```

```
 return slash;
}

/*
*****************************************************************************
** rtl_term()
**
** description:
** right-to-left terminal
** If right-to-left terminal get primary and secondary status
** and see what the mode of order to the terminal is.
**
** assumptions:
** only a hp150 or hp2392 can be a right-to-left terminal
** TERM set to reflect the terminal type.
**
** return value:
** TRUE if right-to-left terminal
** FALSE if not right-to-left terminal
*****************************************************************************
*/

static int
rtl_term( term_mode, term_order, term)
nl_mode *term_mode;  /* mode of terminal */
nl_order *term_order;  /* order of terminal */
char **term;   /* terminal name */
{
 char buf[MAX_TBUF]; /* buffer for terminal information */
 struct termio tbuf; /* buffer for termio structure */
 struct termio tbufsave; /* save old info */

 /* assume right-to-left terminal is hp150 or hp2392 */
 *term = getenv( "TERM");
 if (strncmp( *term, "hp150", 5) && strncmp( *term, "hp2392", 6)) {
  return FALSE;
 }

 /* fetch & save current status of terminal driver */
 if (ioctl( 1, TCGETA, &tbuf) == -1) {
  Perror( "ioctl");
  return FALSE;
 }
 tbufsave = tbuf;

 /* turn off echo to prevent status bytes from appearing
       on screen */
 tbuf.c_lflag &= ~ECHO;
```

```
/* set status of terminal driver with echo off */
if (ioctl( 1, TCSETAF, &tbuf) == -1) {
 Perror( "ioctl");
 return FALSE;
}

/* turn off handshaking (G & H straps on) */
if (fputs( on_straps, stdout) == EOF) {
 Perror( "fputs");
 return FALSE;
}

/* get alpha-numeric capabilities: ordering is byte 2, bit 4 */
if (fputs( an_cap, stdout) == EOF) {
 Perror( "fputs");
 return FALSE;
}
if (! fgets( buf, MAX_TBUF, stdin)) {
 Perror( "fgets");
 return FALSE;
}
*term_order = (buf[DISPLAY] & ORDER) ? NL_KEY : NL_SCREEN;

/* get secondary status: mode is byte 13, bit 3 */
if (fputs( sec_status, stdout) == EOF) {
 Perror( "fputs");
 return FALSE;
}
if (! fgets( buf, MAX_TBUF, stdin)) {
 Perror( "fgets");
 return FALSE;
}
*term_mode = (buf[RTL_SEC] & MODE) ? NL_NONLATIN : NL_LATIN;

/* turn on D1 handshaking (G & H straps off) */
if (fputs( off_straps, stdout) == EOF) {
 Perror( "fputs");
 return FALSE;
}

/* restore status of terminal driver */
if (ioctl( 1, TCSETAF, &tbufsave) == -1) {
 Perror( "ioctl");
 return FALSE;
}

 return TRUE;
}
```

Makefile Example

```
FINDMSG         = /usr/bin/findmsg
GENCAT          = /usr/bin/gencat
LINT            = /usr/bin/lint
RM              = /bin/rm

CFLAGS          = -D_HPUX_SOURCE -O
LDFLAGS         = -s
IFLAGS          =
LIBS            =

SOURCE          = rtlcat.c
OBJECT          = rtlcat.o

all:            rtlcat rtlcat.cat

rtlcat:         $(OBJECT)
                $(CC) -o $@ $(OBJECT) $(LDFLAGS) $(LIBS)

rtlcat.cat:     rtlcat.msg

# NL_SETN defined once in the first source file or
# NL_SETN defined with different values for each source file

rtlcat.msg:     $(SOURCE)
                $(FINDMSG) $(SOURCE) > $@

.msg.cat:
                $(GENCAT) $*.cat $*.msg

.c.o:
                $(CC) -c $(CFLAGS) $(IFLAGS) $<

lint:           $(SOURCE)
                $(LINT) -u $(CFLAGS) $(IFLAGS) $(SOURCE) > lint

clean:
                $(RM) -f *.o *.msg lint

clobber:        clean
                $(RM) -f rtlcat *.cat

.SUFFIXES:    .cat .msg
```

C

NLS References

This appendix provides two tables for NLS routines:

- The first table (Table C-1) is an alphabetical *HP-UX Reference* entry list along with routines associated with each.
- The second table (Table C-2) is an alphabetical listing of current NLS library routines with their associated entry in the *HP-UX Reference* along with a description of the routine's purpose.

This appendix provides one table of NLS commands:

- The third table (Table C-3) is an alphabetical listing of NLS commands, along with a brief description of each.

Table C-1. HP-UX Reference NLS Entries

HP-UX Reference Entry	Description	Associated Routines
catgets(3C)	Gets a program message.	
catopen(3C)	Open (`catopen`) and close (`catclose`) a message catalog for reading.	`catopen`, `catclose`
ctime(3C)	Convert date and time to string.	`ctime`, `localtime`, `gmtime`, `mktime`, `difftime`, `asctime`, `timezone`, `daylight`, `tzname`, `tzset`, `nl_ctime`, `nl_cxtime`, `nl_asctime`, `nl_ascxtime`
ctype(3C)	Classify character-coded integer values according to the rules of the coded character set.	`isalpha`, `isupper`, `islower`, `isdigit`, `isxdigit`, `isalnum`, `isspace`, `ispunct`, `isprint`, `isgraph`, `iscntrl`, `isascii`
ecvt(3C)	Convert floating-point number to a string.	`ecvt`, `fcvt`, `gcvt`, `nl_gcvt`
environ(5)	The user environment and environment variables.	
fgetws(3C)	Get a wide character string from a stream file.	
findmsg(1)	Create message catalog file for modification.	`findmsg`, `dumpmsg`
findstr(1)	Find strings for inclusion in message catalogs.	
forder(1)	Convert file data order.	
fputws(3C)	Put a wide character string on a stream file.	
gencat(1)	Generate a formatted message catalog file.	

Table C-1. HP-UX Reference NLS Entries (continued)

HP-UX Reference Entry	Description	Associated Routines
getc(1)	Get character or word from a stream file.	`getchar`, `fgetc`, `getw`
gets(1)	Get a string from a stream.	`fgets`
getwc(3C)	Get a wide character from a stream file.	`getwchar`, `fgetwc`
hpnls(5)	HP Native Language Support (NLS) Model.	
iconv(1)	Code set conversion command.	
iconv(3C)	Code set conversion routines.	`iconvsize`, `iconvopen`, `iconvclose`, `iconvlock`, `ICONV`, `ICONV1`, `ICONV2`
insertmsg(1)	Use findstr(1) output to insert calls to `catgets`.	
lang(5)	Description of supported languages.	
langinfo(5)	Language information constants.	
locale(1)	Display current locale environment.	
localeconv(3C)	Query the numeric formatting conventions of the current locale.	
localedef(1M)	Create or dump `locale.inf` locales.	

Table C-1. HP-UX Reference NLS Entries (continued)

HP-UX Reference Entry	Description	Associated Routines
multibyte(3C)	Multibyte characters and strings conversions.	`mblen`, `mbtowc`, `mbstowcs`, `wctomb`, `wcstombs`
nl_langinfo(3C)	Language information.	
nl_tools_16(3C)	Tools to process 16-bit characters.	`firstof2`, `secof2`, `byte_status`, `c_colwidth`, `FIRSTof2`, `SECof2`, `BYTE_STATUS`, `C_COLWIDTH`, `CHARAT`, `ADVANCE`, `CHARADV`, `WCHAR`, `WCHARADV`, `PCHAR`, `PCHARADV`
nljust(1)	Justify lines, left or right, for printing.	
nlsinfo(1)	Display native language support information.	
printf(3C)	Print formatted output.	`printf`, `nl_printf`, `fprintf`, `nl_fprintf`, `sprintf`, `nl_sprintf`
putwc(3C)	Put a wide character on a stream file.	`putwchar`, `fputwc`
setlocale(3C)	Set and get the locale of a program.	`setlocale`, `getlocale`
strftime(3C)	Convert date and time to string.	
string(3C)	Character string operations.	`strcat`, `strncat`, `strcmp`, `strncmp`, `strcpy`, `strncpy`, `strdup`, `strlen`, `strchr`, `strrchr`, `strpbrk`, `strspn`, `strcspn`, `strstr`, `strtok`, `strcoll`, `strxfrm`, `nl_strcmp`, `nl_strncmp`

Table C-1. HP-UX Reference NLS Entries (continued)

HP-UX Reference Entry	Description	Associated Routines
strord(3C)	Convert string data order.	
strtod(3C)	Convert string to double-precision number.	`strtod`, `atof`, `nl_strtod`, `nl_atof`
ungetwc(3C)	Push a wide character back into an input stream.	
wconv(3C)	Translate wide characters.	`towupper`,`towlower`
wcsftime(3C)	Convert date and time to wide character string.	
wcstod(3C)	Convert wide character string to double-precision number.	
wcstol(3C)	Convert wide character string to long integer.	`wcstoul`
wctype(3C)	Classify wide characters.	`iswalpha`, `iswupper`, `iswlower`, `iswdigit`, `iswxdigit`, `iswalnum`, `iswspace`, `iswpunct`, `iswprint`, `iswgraph`, `iswcntrl`, `iswctype`

Table C-2. NLS Library Routines

Routine	HP-UX Reference Entry	Description
`atof(`*const char *string*`)`	*strtod*(3C)	Converts a string to a double.
`byte_status(`*int c*`,`*int laststatus*`)`	*nl_tools_16*(3C)	Indicates if *c* is a one-byte character, first byte of a two-byte character, or second byte of a two-byte character.
`catopen(`*const char *name*`,`*int oflag*`)`	*catopen*(3C)	Opens a message catalog and returns a catalog descriptor. *name* specifies the name of the message catalog being opened. [*oflag* is reserved for future use and should be set to 0 (zero).]
`catclose(`*nl_catd catd*`)`	*catopen*(3C)	Closes message catalog *catd*.
`catgets(`*nl_catd catd, int set_num, int msg_num, const char *def_str*`)`	*catgets*(3C)	Reads message *msg_num* in set *set_num* from the message catalog identified by *catd*, a catalog descriptor returned from a previous call to *catopen*(3C). *def_str* points to a default message string returned by **catgets** if the call fails.
`C_COLWIDTH(`*int c*`)`	*nl_tools_16*(3C)	Takes *c*, which is assumed to be either a one-byte character or the first byte of a two-byte character, and returns the number of columns the character would occupy on a terminal display.
`fgetwc(`*FILE *stream*`)`	*getwc*(3C)	Returns the next character from the named input *stream*, converts that to the corresponding wide character and moves the file pointer ahead one character in *stream*. **fgetwc()** is defined only as a function.

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`fgetws(`*wchar_t *ws, int n, FILE *stream*`)`	*fgetws*(3C)	Reads characters from the *stream*, converts them into corresponding wide characters and places them into the array pointed to by *ws*, until *n-1* characters are read, a new-line is read and transferred to *ws*, or an end-of-file condition is encountered. The wide string is then terminated with a null wide character.
`FIRSTof2(`*int c*`)`	*nl_tools_16*(3C)	Returns a non-zero value if *c* can be the first byte of a two-byte character according to the NLS environment loaded, and zero if it cannot.
`fprintf(`*FILE *stream, const char *format,* `/* [arg,] */ ... )`	*printf*(3C)	Places output in the named output *stream*. Converts, formats, and prints its `[args]` under control of the *format*.
`fputws(`*const wchar_t *ws, FILE *stream*`)`	*fputws*(3C)	Writes a character string corresponding to the null-terminated, wide-character string pointed to by *ws* to the named output *stream*, but does not append a new-line character or a terminating null character.
`fscanf(`*FILE *stream, const char *format,* `/* [pointer,] */ ... )`	*scanf*(3C)	Reads characters from the stream. Interprets them according to the control string *format* argument, and stores the results in its `[pointer]` arguments.
`gcvt(`*double value, size_t ndigit, char *buf*`)`	*ecvt*(3C)	Converts *value* into a null-terminated string in the array pointed to by *buf* with *ndigit* significant digits, and returns *buf*.

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`getwc(`*FILE *stream*`)`	*getwc*(3C)	Returns the next character from the named input *stream*, converts that to the corresponding wide character and moves the file pointer ahead one character in *stream*. `getwc()` is defined both as a macro and a function.
`getwchar(void)`	*getwc*(3C)	Is defined as `getwc(stdin)`, so it is both a macro and a function.
`isalnum(`*int c*`)`	*ctype*(3C)	Indicates if *c* is alphanumeric (letters or digits). Defined for range -1(EOF) to 255.
`isalpha(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a letter. Defined for range -1(EOF) to 255.
`isascii(`*int c*`)`	*ctype*(3C)	Indicates if *c* is any ASCII character code between 0 and 0177, inclusive. Defined on all integer values.
`iscntrl(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a control character (in ASCII: character codes less than 040 and the delete character (0177)). Defined for range -1(EOF) to 255.
`isdigit(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a digit. (in ASCII: characters [0-9]) Defined for range -1(EOF) to 255.
`isgraph(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a visible character. Defined for range -1(EOF) to 255.
`islower(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a lowercase alphabetic character. Defined for range -1(EOF) to 255.
`isprint(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a printing character. Defined for range -1(EOF) to 255.

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`ispunct(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters). Defined for range -1(EOF) to 255.
`isspace(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a character that creates "white space" in displayed test (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed). Defined for range -1(EOF) to 255.
`isupper(`*int c*`)`	*ctype*(3C)	Indicates if *c* is an uppercase alphabetic character. Defined -1(EOF) to 255.
`isxdigit(`*int c*`)`	*ctype*(3C)	Indicates if *c* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]). Defined for range -1(EOF) to 255.
`iswalnum(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is an alphanumeric (letters or digits).
`iswalpha(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a letter.
`iswdigit(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a decimal digit (in ASCII: characters [0-9]).
`iswcntrl(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).
`iswctype(`*wint_t wc*`,`*wctypet prop*`)`	*wctype*(3C)	Indicates if *wc* has the property defined by *prop*.
`iswgraph(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a visible character (in ASCII: printing characters, excluding the space character (040)).

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`iswlower(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a lowercase letter.
`iswprint(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a printing character.
`iswpunct(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letter.
`iswspace(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).
`iswupper(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is an uppercase letter.
`iswxdigit(`*wint_t wc*`)`	*wctype*(3C)	Indicates if *wc* is a hexadecimal digit (in ASCII: characters [0-9], [A-F], or [a-f]).
`mblen(`*const char *s, size_t n*`)`	*multibyte*(3C)	Determines the number of bytes in the multi-byte character pointed to by *s*.
`mbtowc(`*wchar_t *pwc, const char *s, size_t n*`)`	*multibyte*(3C)	Determines the number of bytes in the multi-byte character pointed to by *s*, determines the code for the value of type *wchar_t* corresponding to that multibyte character, and then stores the result in the object pointed to by *pwc*.
`mbstowcs(`*wchar_t *pwcs, const char *s, size_t n*`)`	*multibyte*(3C)	Converts a sequence of multi-byte characters from the array pointed to by *s* into a sequence of corresponding codes and stores these codes into the array pointed toby *pwcs*, stopping after either *n* codes or a code with value zero (a converted null character) is stored.

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`nl_langinfo(`*nl_item item*`)`	*langinfo*(3C)	Returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the program's locale (see *setlocale*(3C)). The manifest constant names and values of *item* are defined in `langinfo.h`.
`sprintf(`*char *s, const char *format, /* [arg,] */ ...)*	*printf*(3C)	Places "output", followed by the null character (\0), in consecutive bytes starting at **s*. [It is the user's responsibility to ensure that enough storage is available.] Converts, formats, and prints its `[arg]`s under control of the *format*.
`sscanf(`*const char *s, const char *format, /* [pointer,] */ ...)*	*scanf*(3C)	Reads characters from the character string *s*. Interprets them according to the control string *format* argument, and stores the results in its `[pointer]` arguments.
`SECof2(`*int c*`)`	*nl_tools_16*(3C)	Takes a byte *c* and returns a non-zero value if it can be the second byte of a two-byte character according to the loaded NLS environment, and zero if it cannot.
`setlocale(`*category, locale*`)`	*setlocale*(3C)	Sets the program environment for the specified *category* according to the language definition table for the specified *locale*.
`strcoll(`*const char *s1,' const char *s2*`)`	*string*(3C)	Compares two strings, indicating if *s1* is less than, greater than, or equal to *s2*, according to the collating sequence for the user's language.

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`strftime(`*char *s, size_t maxsize, const char *format, const struct tm *timeptr*`)`	*strftime*(3C)	Converts contents of a *tm* structure to a formatted date and time string. Places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* characters are placed into the array. The appropriate characters are determined by the program's locale, by the values in the structure pointed to by *timeptr*, and by the `TZ` (time zone) environment variable.
`strord(`*char *s1, const char *s2, nl_mode m*`)`	*strord*(3C)	Converts the order of characters in *s1* from screen to keyboard order or vice versa and places the result in *s1*. The arguments *s1* and *s2* point to strings. The conversion is based on mode information indicated by the argument *m*.
`strtod(`*const char *str, char **ptr*`)`	*strtod*(3C)	Returns, as a double-precision floating-point number, the value represented by the character string pointed to by *str*. The string is scanned (leading white-space characters as defined by `isspace` are ignored) up to the first unrecognized character. If the value of *ptr* is not `(char **)NULL`, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, **ptr* is set to *str*, and zero is returned.

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`strxfrm(`*char *s1, const char *s2, size_t n*`)`	*string*(3C)	Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the `strcmp()` function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `strcoll()` function applied to the same two original strings. No more than *n* bytes are placed into the resulting string, including the terminating null character.
`tolower(`*int c*`)`	*conv*(3C)	Returns the corresponding lowercase letter if *c* points to an uppercase letter; otherwise returns *c* unaltered.
`toupper(`*int c*`)`	*conv*(3C)	Returns the corresponding uppercase letter if *c* points to a lowercase letter; otherwise returns *c* unaltered.
`towlower(`*wint_t wc*`)`	*wconv*(3C)	Has as domain a *wint_t*, the value of which is representable as a *wchar_t* or the value `WEOF`. Returns the corresponding lowercase letter if *wc* points to an uppercase letter; otherwise returns *wc* unaltered.
`towupper(`*wint_t wc*`)`	*wconv*(3C)	Has as domain a *wint_t*, the value of which is representable as a *wchar_t* or the value `WEOF`. Returns the corresponding uppercase letter if *wc* points to a lowercase letter; otherwise returns *wc* unaltered.

Table C-2. NLS Library Routines (continued)

Routine	HP-UX Reference Entry	Description
`wcstombs(`*char *s, const wchar_t *pwcs, size_t n*`)`	*multibyte*(3C)	Converts a sequence of codes corresponding to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters and stores them into the array pointed to by *s*, stopping if a multibyte character exceeds the limit of *n* total bytes or if a null character is stored.
`wctomb(`*char *s, wchar_t wchar*`)`	*multibyte*(3C)	Determines the number of bytes needed to represent the multi-byte character corresponding to the code whose value is *wchar* and stores the multibyte character representation in the array object pointed to by *s*.

Table C-3. NLS Commands

Command	Description
`dumpmsg`	Extracts messages and set declarations from a message catalog and writes then to standard out.
`findmsg`	Examines C source files and writes to standard output strings embedded in `catgets()` calls and strings tagged with the comment `/* catgets */`
`findstr`	Examines C source files for double quoted strings, writing the strings and additional information to standard output; the form of the information is suitable as input to `insertmsg`.
`forder`	Converts the order of character in a file from screen order to keyboard order or from keyboard order to screen order.
`gencat`	Creates a message catalog from one or more message catalog source files, optionally merging the new catalog with an existing catalog.
`insertmsg`	Replaces double-quoted strings with `catgets()` calls in a C program source file.
`nljust`	Formats data with a right-to-left orientation for printing. (Used with the command `pr` and `lp`).

D

Previous Usage

NLS is a relatively new technology. As a result of its recent introduction into the UNIX industry, it is still in the process of development and refinement. Industry-wide standards have not been firmly established, and thus there are different implementations of NLS.

In keeping with HP-UX's standardization efforts, HP-UX has undergone some changes. Some routines that HP-UX offers are HP-UX specific. Other NLS interfaces are specified by different standards (X/Open, POSIX). The HP-UX specific routines are currently supported, but will be withdrawn at some time in the future. Use of the standard interfaces is recommended instead.

Obsolete Routines

Obsolete routines have replacements that will provide similar or related functionality. About half of the obsolete routines have an equivalent replacement; the only difference is the name of the routine. However, the remaining routines do not have an equivalent replacement; instead, several routines may need to be called or the parameter list may be slightly different. Table D-1 lists these obsolete routines and their replacements.

D

Table D-1. Obsolete Routines and Recommended Replacements

Obsolete Routine	Recommended Routine	Man Page Reference	Notes
`buildlang`	`localedef`	*localedef*(1M)	no 1-1 replacement
`catgetmsg`	`catgets`	*catgets*(3C)	no 1-1 replacement
`catread`	`catgets`	*catgets*(3C)	no 1-1 replacement
`currlangid`	*none*		no longer needed
`fprintmsg`	`fprintf`	*printf*(3C)	
`getmsg`	`catgets`	*getmsg*(3C)	No 1-1 replacement. Removed from lib.c due to name conflict with Streams.
`ICONV`	`iconv`	*iconv*(3C)	XPG4 interface coming
`iconvclose`	`iconv_close`	*iconv*(3C)	XPG4 interface coming
`iconvlock`	*none*	*iconv*(3C)	
`iconvopen`	`iconv_open`	*iconv*(3C)	XPG4 interface coming
`iconvsize`	*none*	*iconv*(3C)	
`idtolang`	*none*		no longer needed
`langid`	*none*		no longer needed
`langinfo`	`nl_langinfo`	*nl_langinfo*(3C)	
`langinit`	`setlocale`	*setlocale*(3C)	

Table D-1.
Obsolete Routines and Recommended Replacements (continued)

Obsolete Routine	Recommended Routine	Man Page Reference	Notes
`langtoid`	*none*		no longer needed
`msgbuf.h`			non-standard
`n-computer`	`C` or `POSIX`	*lang*(5)	language name
`nl_asctime`	`strftime`	*strftime*(3C)	
`nl_ascxtime`	`strftime`	*strftime*(3C)	
`nl_atof`	`atof`	*strtod*(3C)	
`nl_catopen`	`catopen`	*catopen*(3C)	
`nl_ctime`	`strftime`	*strftime*(3C)	
`nl_cxtime`	`strftime`	*strftime*(3C)	
`nl_fprintf`	`fprintf`	*string*(3C)	
`nl_fscanf`	`fscanf`	*scanf*(3S)	
`nl_gcvt`	`gcvt`	*ecvt*(3C)	
`nl_init`	`setlocale`	*setlocale*(3C)	
`nl_isalnum`	`isalnum`	*ctype*(3C)	
`nl_isalpha`	`isalpha`	*ctype*(3C)	
`nl_iscntrl`	`iscntrl`	*ctype*(3C)	
`nl_isdigit`	`isdigit`	*ctype*(3C)	
`nl_isgraph`	`isgraph`	*ctype*(3C)	
`nl_islower`	`islower`	*ctype*(3C)	
`nl_isprint`	`isprint`	*ctype*(3C)	
`nl_ispunct`	`ispunct`	*ctype*(3C)	

Table D-1.
Obsolete Routines and Recommended Replacements (continued)

Obsolete Routine	Recommended Routine	Man Page Reference	Notes
nl_isspace	isspace	*ctype*(3C)	
nl_isupper	isupper	*ctype*(3C)	
nl_isxdigit	isxdigit	*ctype*(3C)	
nl_msg	catgets	*catgets*(3C)	
nl_printf	printf	*printf*(3S)	
nl_scanf	scanf	*scanf*(3S)	
nl_sprintf	sprintf	*printf*(3S)	
nl_sscanf	sscanf	*scanf*(3S)	
nl_strcmp	strcoll	*string*(3C)	
nl_strncmp	strcoll	*string*(3C)	no 1-1 replacement
nl_strtod	strtod	*strtod*(3C)	
nl_tolower	tolower	*conv*(3C)	
nl_toupper	toupper	*conv*(3C)	
nlsinfo	locale	*locale*(1)	no 1-1 replacement
printmsg	printf	*printf*(3S)	
sprintmsg	sprintf	*printf*(3S)	
strcmp8	strcoll	*string*(3C)	
strcmp16	strcoll	*string*(3C)	
strncmp8	strcoll	*string*(3C)	no 1-1 replacement
strncmp16	strcoll	*string*(3C)	no 1-1 replacement

Proprietary Commands and Interfaces

Table D-2 and Table D-3 list some NLS routines that exist only on HP-UX systems. These routines may be obsoleted over time as standard commands and interfaces emerge. Usage of these utilities and routines in applications may not be portable to other vendor's platforms.

Table D-2. Proprietary Commands

Utility Name	Man Page Reference
`dumpmsg`	*findmsg*(1)
`findmsg`	*findmsg*(1)
`findstr`	*findstr*(1)
`forder`	*forder*(1)
`insertmsg`	*insertmsg*(1)
`nljust`	*nljust*(1)

WPI should be used instead of the library calls listed in Table D-3.

Table D-3. Proprietary Library Calls

Utility Name	Man Page Reference
ADVANCE	*nl_tools_16*(3C)
advance	*nl_tools_16*(3C)
BYTE_STATUS	*nl_tools_16*(3C)
byte_status	*nl_tools_16*(3C)
C_COLWIDTH	*nl_tools_16*(3C)
c_colwidth	*nl_tools_16*(3C)
CHARADV	*nl_tools_16*(3C)
CHARAT	*nl_tools_16*(3C)
FIRSTOF2	*nl_tools_16*(3C)
firstof2	*nl_tools_16*(3C)
PCHAR	*nl_tools_16*(3C)
getlocale	*setlocale*(3C)
ICONV1	*iconv*(3C)
ICONV2	*iconv*(3C)
PCHARADV	*nl_tools_16*(3C)
SECOF2	*nl_tools_16*(3C)
secof2	*nl_tools_16*(3C)
strord	*strord*(3C)
WCHAR	*nl_tools_16*(3C)
WCHARADV	*nl_tools_16*(3C)

E

Languages and Codesets

Following are native languages and the HP codesets that support them. Simply find the language (or country) that you are interested in. The appropriate `LANG` environment variable is in the second column.

Note We recommend the use of ISO 8859 codesets for new applications or applications that require a high degree of portability. These codesets are based on a widely accepted industry standard.

Table E-1. Language Codesets

lang-id	LANG=	Language	Codeset
0	n-computer	American English	
101	american.iso88591	American English	ISO 8859-1
1	american	American English	ROMAN8
51	arabic	Arabic	ARABIC8
52	arabic-w	Western Arabic	ARABIC8
301	arabic.iso88596	Arabic	ISO 8859-6
181	bulgarian	Bulgarian	ISO 8859-5
99	C	Computer	
102	c-french.iso88591	French Canadian	ISO 8859-1
2	c-french	French Canadian	ROMAN8
201	chinese-s	Simplified Chinese	PRC15
211	chinese-t	Traditional Chinese	ROC15
212	chinese-t.big5	Traditional Chinese	BIG5
142	czech	Czechoslovakian	ISO 8859-2
103	danish.iso88591	Danish	ISO 8859-1
3	danish	Danish	ROMAN8
104	dutch.iso88591	Dutch	ISO 8859-1
4	dutch	Dutch	ROMAN8
105	english.iso88591	English	ISO 8859-1
5	english	English	ROMAN8
106	finnish.iso88591	Finnish	ISO 8859-1
6	finnish	Finnish	ROMAN8
107	french.iso88591	French	ISO 8859-1
7	french	French	ROMAN8
108	german.iso88591	German	ISO 8859-1
8	german	German	ROMAN8
61	greek	Greek	GREEK8
321	greek.iso88597	Greek	ISO 8859-7
71	hebrew	Hebrew	HEBREW8
341	hebrew.iso88598	Hebrew	ISO 8859-8
143	hungarian	Hungarian	ISO 8859-2

Table E-1. Language Codesets (continued)

lang-id	LANG=	Language	Codeset
114	icelandic.iso88591	Icelandic	ISO 8859-1
14	icelandic	Icelandic	ROMAN8
109	italian.iso88591	Italian	ISO 8859-1
9	italian	Italian	ROMAN8
222	japanese.euc	Japanese	JAPANEUC
221	japanese	Japanese	JAPAN15
41	katakana	Katakana	KANA8
231	korean	Korean	KOREA15
110	norwegian.iso88591	Norwegian	ISO 8859-1
10	norwegian	Norwegian	ROMAN8
144	polish	Polish	ISO 8859-2
111	portuguese.iso88591	Portuguese	ISO 8859-1
11	portuguese	Portuguese	ROMAN8
100	POSIX	POSIX default	
145	rumanian	Rumanian	ISO 8859-2
180	russian	Russian	ISO 8859-5
146	serbocroatian	Serbo-Croatian	ISO 8859-2
148	slovene	Slavic (Slovenia)	ISO 8859-2
112	spanish.iso88591	Spanish	ISO 8859-1
12	spanish	Spanish	ROMAN8
113	swedish.iso88591	Swedish	ISO 8859-1
13	swedish	Swedish	ROMAN8
91	thai	Thai	THAI8
81	turkish	Turkish	TURKISH8
361	turkish.iso88599	Turkish	ISO 8859-9

b_8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b_7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b_6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
b_5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

b_4	b_3	b_2	b_1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	NUL	DLE	SP	0	@	P	‘	p				¯	â	Å	Á	Þ
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q			À	Ý	ê	Î	Ã	þ
0	0	1	0	2	STX	DC2	"	2	B	R	b	r			Â	ý	ô	Ø	ã	•
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s			È	°	û	Æ	Ð	µ
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t			Ê	Ç	á	å	ð	¶
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u			Ë	ç	é	í	Í	‰
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v			Î	Ñ	ó	ø	Ì	—
0	1	1	1	7	BEL	ETB	’	7	G	W	g	w			Ï	ñ	ú	æ	Ó	¼
1	0	0	0	8	BS	CAN	(	8	H	X	h	x			´	¡	à	Ä	Ò	½
1	0	0	1	9	HT	EM	)	9	I	Y	i	y			`	¿	è	ì	Õ	ª
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z			^	¤	ò	Ö	õ	º
1	0	1	1	11	VT	ESC	+	;	K	[	k	{			¨	£	ù	Ü	Š	«
1	1	0	0	12	FF	FS	,	<	L	\	l	|			~	¥	ä	É	š	■
1	1	0	1	13	CR	GS	-	=	M	]	m	}			Ù	§	ë	ï	Ú	»
1	1	1	0	14	SO	RS	.	>	N	^	n	~			Û	*f*	ö	ß	Ÿ	±
1	1	1	1	15	SI	US	/	?	O	_	o	DEL			£	¢	ü	Ô	ÿ	

Figure E-1. Roman8 Coded Character Set

Displaying Character Sets on Your Terminal

You can display a table of one-byte characters supported on a particular device by directing the output of this program to that device.

```
#include <stdio.h>
main()
{
   int c1,c2,pchar;
   printf("  \\  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15\n");
      for (c1=0; c1<16; c1++){
         printf("%3d  ",c1);
         for (c2=0; c2<16; c2++){
            pchar=(((c2*16+c1)&0x7f)>31)?(c2*16+c1):' ';
            printf("%c  ",pchar);
         }
         printf("\n");
   }
}
```

LC_COLLATE Example for Spanish

```
###################################
# Set up the LC_COLLATE category

LC_COLLATE
modifier "fold"
collating-element <CH> from "CH"
collating-element <Ch> from "Ch"
collating-element <cH> from "cH"
collating-element <ch> from "ch"
collating-element <LL> from "LL"
collating-element <Ll> from "Ll"
collating-element <lL> from "lL"
collating-element <ll> from "ll"


order_start
' ' ' '
\xa0 \xa0
'0' '0'
...
'9' '9'
'A' 'A';'A'
\xd3 'A';"AE"
'a' 'A';'a'
\xd7 'A';"ae"
\xe0 'A';\xe0
\xc4 'A';\xc4
\xa1 'A';\xa1
\xc8 'A';\xc8
\xa2 'A';\xa2
\xc0 'A';\xc0
\xd8 'A';\xd8
\xcc 'A';\xcc
\xd0 'A';\xd0
\xd4 'A';\xd4
\xe1 'A';\xe1
\xe2 'A';\xe2
'B' 'B';'B'
'b' 'B';'b'
```

```
'C' 'C';'C'
'c' 'C';'c'
\xb4 'C';\xb4
\xb5 'C';\xb5
<CH> <CH>;<CH>
<Ch> <CH>;<Ch>
<cH> <CH>;<cH>
<ch> <CH>;<ch>
'D' 'D';'D'
'd' 'D';'d'
\xe3 'D';\xe3
\xe4 'D';\xe4
'E' 'E';'E'
'e' 'E';'e'
\xdc 'E';\xdc
\xc5 'E';\xc5
\xa3 'E';\xa3
\xc9 'E';\xc9
\xa4 'E';\xa4
\xc1 'E';\xc1
\xa5 'E';\xa5
\xcd 'E';\xcd
'F' 'F';'F'
'f' 'F';'f'
'G' 'G';'G'
'g' 'G';'g'
'H' 'H';'H'
'h' 'H';'h'
'I' 'I';'I'
'i' 'I';'i'
\xe5 'I';\xe5
\xd5 'I';\xd5
\xe6 'I';\xe6
\xd9 'I';\xd9
\xa6 'I';\xa6
\xd1 'I';\xd1
\xa7 'I';\xa7
\xdd 'I';\xdd
'J' 'J';'J'
'j' 'J';'j'
'K' 'K';'K'
'k' 'K';'k'
'L' 'L';'L'
'l' 'L';'l'
<LL> <LL>;<LL>
<Ll> <LL>;<Ll>
<lL> <LL>;<lL>
<ll> <LL>;<ll>
'M' 'M';'M'
```

F

```
'm' 'M';'m'
'N' 'N';'N'
'n' 'N';'n'
\xb6 \xb6;\xb6
\xb7 \xb6;\xb7
'O' 'O';'O'
'o' 'O';'o'
\xe7 'O';\xe7
\xc6 'O';\xc6
\xe8 'O';\xe8
\xca 'O';\xca
\xdf 'O';\xdf
\xc2 'O';\xc2
\xda 'O';\xda
\xce 'O';\xce
\xe9 'O';\xe9
\xea 'O';\xea
\xd2 'O';\xd2
\xd6 'O';\xd6
'P' 'P';'P'
'p' 'P';'p'
'Q' 'Q';'Q'
'q' 'Q';'q'
'R' 'R';'R'
'r' 'R';'r'
'S' 'S';'S'
\xde 'S';"SS"
's' 'S';'s'
\xeb 'S';\xeb
\xec 'S';\xec
'T' 'T';'T'
't' 'T';'t'
'U' 'U';'U'
'u' 'U';'u'
\xed 'U';\xed
\xc7 'U';\xc7
\xad 'U';\xad
\xcb 'U';\xcb
\xae 'U';\xae
\xc3 'U';\xc3
\xdb 'U';\xdb
\xcf 'U';\xcf
'V' 'V';'V'
'v' 'V';'v'
'W' 'W';'W'
'w' 'W';'w'
'X' 'X';'X'
'x' 'X';'x'
'Y' 'Y';'Y'
```

```
'y' 'Y';'y'
\xee 'Y';\xee
\xef 'Y';\xef
'Z' 'Z';'Z'
'z' 'Z';'z'
\xf0 \xf0;\xf0
\xf1 \xf0;\xf1
\xb1 \xb1
\xb2 \xb2
\xf2 \xf2
\xf3 \xf3
\xf4 \xf4
\xf5 \xf5
'(' '('
')' ')'
'[' '['
']' ']'
'{' '{'
'}' '}'
\xfb \xfb
\xfd \xfd
'<' '<'
'>' '>'
'=' '='
'+' '+'
'-' '-'
\xfe \xfe
\xf7 \xf7
\xf8 \xf8
\xb3 \xb3
'%' '%'
'*' '*'
'.' '.'
',' ','
';' ';'
':' ':'
\xb9 \xb9
'?' '?'
\xb8 \xb8
'!' '!'
'/' '/'
'\' '\'
'|' '|'
'@' '@'
'&' '&'
'#' '#'
\xbd \xbd
'$' '$'
\xbf \xbf
```

```
\xbb \xbb
\xaf \xaf
\xbc \xbc
\xbe \xbe
\xba \xba
;";  ;";
;‘;  ;‘;
;';  ;';
;^;  ;^;
;~;  ;~;
\xa8 \xa8
\xa9 \xa9
\xaa \xaa
\xab \xab
\xac \xac
;_;  ;_;
\xf6 \xf6
\xb0 \xb0
\xf9 \xf9
\xfa \xfa
\xfc \xfc
\x0 \x0
\x1 \x1
...
\x1f \x1f
\x80 \x80
...
\x9f \x9f
\x7f \x7f
\xff \xff
order_end

modifier "nofold"
collating-element <CH> from "CH"
collating-element <Ch> from "Ch"
collating-element <LL> from "LL"
collating-element <Ll> from "Ll"
collating-element <cH> from "cH"
collating-element <ch> from "ch"
collating-element <lL> from "lL"
collating-element <ll> from "ll"


order_start
;  ;  ;  ;
\xa0 \xa0
;0;  ;0;
...
;9;  ;9;
```

F

```
'A' 'A';'A'
\xd3 'A';"AE"
\xe0 'A';\xe0
\xa1 'A';\xa1
\xa2 'A';\xa2
\xd8 'A';\xd8
\xd0 'A';\xd0
\xe1 'A';\xe1
'B' 'B'
'C' 'C';'C'
\xb4 'C';\xb4
<CH> <CH>;<CH>
<Ch> <CH>;<Ch>
'D' 'D';'D'
\xe3 'D';\xe3
'E' 'E';'E'
\xdc 'E';\xdc
\xa3 'E';\xa3
\xa4 'E';\xa4
\xa5 'E';\xa5
'F' 'F'
'G' 'G'
'H' 'H'
'I' 'I';'I'
\xe5 'I';\xe5
\xe6 'I';\xe6
\xa6 'I';\xa6
\xa7 'I';\xa7
'J' 'J'
'K' 'K'
'L' 'L'
<LL> <LL>;<LL>
<Ll> <LL>;<Ll>
'M' 'M'
'N' 'N'
\xb6 \xb6
'O' 'O';'O'
\xe7 'O';\xe7
\xe8 'O';\xe8
\xdf 'O';\xdf
\xda 'O';\xda
\xe9 'O';\xe9
\xd2 'O';\xd2
'P' 'P'
'Q' 'Q'
'R' 'R'
'S' 'S';'S'
\xeb 'S';\xeb
'T' 'T'
```

```
'U' 'U';'U'
\xed 'U';\xed
\xad 'U';\xad
\xae 'U';\xae
\xdb 'U';\xdb
'V' 'V'
'W' 'W'
'X' 'X'
'Y' 'Y';'Y'
\xee 'Y';\xee
'Z' 'Z'
\xf0 \xf0
'a' 'a';'a'
\xd7 'a';"ae"
\xc4 'a';\xc4
\xc8 'a';\xc8
\xc0 'a';\xc0
\xcc 'a';\xcc
\xd4 'a';\xd4
\xe2 'a';\xe2
'b' 'b'
'c' 'c';'c'
\xb5 'c';\xb5
<cH> <cH>;<cH>
<ch> <cH>;<ch>
'd' 'd';'d'
\xe4 'd';\xe4
'e' 'e';'e'
\xc5 'e';\xc5
\xc9 'e';\xc9
\xc1 'e';\xc1
\xcd 'e';\xcd
'f' 'f'
'g' 'g'
'h' 'h'
'i' 'i';'i'
\xd5 'i';\xd5
\xd9 'i';\xd9
\xd1 'i';\xd1
\xdd 'i';\xdd
'j' 'j'
'k' 'k'
'l' 'l'
<lL> <lL>;<lL>
<ll> <lL>;<ll>
'm' 'm'
'n' 'n'
\xb7 \xb7
'o' 'o';'o'
```

```
\xc6 'o';\xc6
\xca 'o';\xca
\xc2 'o';\xc2
\xce 'o';\xce
\xea 'o';\xea
\xd6 'o';\xd6
'p' 'p'
'q' 'q'
'r' 'r'
\xde \xde;"ßS"
's' \xde;'s'
\xec \xde;\xec
't' 't'
'u' 'u';'u'
\xc7 'u';\xc7
\xcb 'u';\xcb
\xc3 'u';\xc3
\xcf 'u';\xcf
'v' 'v'
'w' 'w'
'x' 'x'
'y' 'y';'y'
\xef 'y';\xef
'z' 'z'
\xf1 \xf1
\xb1 \xb1
\xb2 \xb2
\xf2 \xf2
\xf3 \xf3
\xf4 \xf4
\xf5 \xf5
'(' '('
')' ')'
'[' '['
']' ']'
'{' '{'
'}' '}'
\xfb \xfb
\xfd \xfd
'<' '<'
'>' '>'
'=' '='
'+' '+'
'-' '-'
\xfe \xfe
\xf7 \xf7
\xf8 \xf8
\xb3 \xb3
'%' '%'
```

F

```
'*' '*'
'.' '.'
',' ','
';' ';'
':' ':'
\xb9 \xb9
'?' '?'
\xb8 \xb8
'¡' '¡'
'/' '/'
'\' '\'
'|' '|'
'@' '@'
'&' '&'
'#' '#'
\xbd \xbd
'$' '$'
\xbf \xbf
\xbb \xbb
\xaf \xaf
\xbc \xbc
\xbe \xbe
\xba \xba
'"' '"'
'`' '`'
''' '''
'^' '^'
'~' '~'
\xa8 \xa8
\xa9 \xa9
\xaa \xaa
\xab \xab
\xac \xac
'_' '_'
\xf6 \xf6
\xb0 \xb0
\xf9 \xf9
\xfa \xfa
\xfc \xfc
\x0 \x0
...
\x1f \x1f
\x80 \x80
...
\x9f \x9f
\x7f \x7f
\xff \xff
order_end
END LC_COLLATE
```

Glossary

Note For additional information on terms used with HP-UX, please see the Glossary section of the *HP-UX Reference, vol. 1*.

alternate character set
A code set used to represent special, ancillary characters.

application program
A program which performs a specific task for the end-user.

ARABIC8
The Hewlett-Packard supported 8-bit code set for the Arabic language.

ASCII
American Standard Code for Information Interchange. A 128-character code set represented by 7-bit binary values. (ASCII does not define the value of the eighth bit.) Also, referred to as USASCII.

base character set
A code set consisting of the linguistic characters fundamental to a language.

BIG5
The HP-supported 16-bit code set for BIG5 Traditional Chinese, the language of the Republic of China.

bit
A contraction of BInary digiT. A bit can have a value of 0 or 1.

byte
A unit of data storage consisting of 8 bits. A byte can represent one ASCII, KANA8, GREEK8, TURKISH8, ARABIC8, or ROMAN8 character.

byte redefinition
Corruption of a multi-byte character when any one of its bytes is treated as a 1-byte character.

C (locale)
An invented, artificial computer locale which specifies the minimal environment for C translation. C locale is the default when natural languages/locales are not installed or are not called by a program.

character
A series of one or more bytes representing a single graphic symbol or control code.

coded character set
See **code set.**

code set
A set of unambiguous rules that establishes a one-to-one relationship between each character of a character set and its byte value.

7-bit: A code set that uses seven bits to represent a collection of characters, control codes, and the space character. A 7-bit code set allows a maximum of 128 characters which does not accommodate international languages. ASCII is an example of a 7-bit code set.

8-bit: A code set that uses all eight bits of a single byte to encode each character in the code set. These code sets are designed so the range 0 through 127 are ASCII including the control codes and space character. Non-ASCII characters appear in the range 128 through 255. (Note, the KANA8 character set substitutes the yen symbol for the backslash symbol, so it is not a superset of ASCII.)

multi-byte: A code set that uses two or more bytes to encode characters. Languages such as Chinese, Japanese, and Korean require more than 256 characters, which is the maximum provided by 8-bit character sets. Under different circumstances, 2 bytes can be interpreted as one multi-byte value or two single-byte values.

single-byte: a 7-bit or 8-bit code set.

collating sequence
The ordering sequence assigned to characters or a group of characters when they are sorted and ordered by a computer.

command
A program which is executed by the shell command interpreter. Arguments following the command name are passed to the command program. You can write your own command programs, either as compiled programs or as shell scripts (written in the shell command language).

command interpreter
A program that reads lines typed at the keyboard or from a file, and interprets them as requests to execute other programs. The command interpreter for HP-UX is called the shell.

comment
An expression used to document a program or routine that has no effect on the execution of the program.

compiler
A program that translates a high-level language (source code) into a machine-dependent form (object code or "binary").

context analysis
The process of determining the proper shape of a character based on its position in the word. For some languages, a character can have a different shape if it is at the start of a word, in the middle of a word, at the end of a word, or standing alone. Currently, context analysis is defined for the Middle Eastern and North African Arabic languages.

control character or control code
A nonprinting member of a character set that produces action in a device. In ASCII, control characters are those in the code range 0 through 31, and 127. These values and the space character, with code value 32, are not used for any other purpose. Code values 128 through 160 and 255 are also treated as control codes in some cases, except in ISO 8859 where 255 is a valid character. Most control characters can be generated by simultaneously pressing a displayable character key and CTRL.

data directionality
Refers to the direction text will appear on the screen; left-to-right or right-to-left.

data ordering
Refers to the arrangement of data within a file, internal buffer, or during a transfer to or from peripherals. The modes of data ordering are "keyboard (phonetic) order" and "screen order".

default search path
The sequence of directory prefixes that `sh`, `csh`, and other HP-UX commands apply when searching for a file known by an incomplete path name. It is defined by PATH in `environ`. Log in sets PATH = `.:bin:/usr/bin`, which means that your working directory is the first directory searched, followed by `/bin`, followed by `/usr/bin`.

directionality
See **data directionality**.

downshifting
The provision for producing lowercase letters by using the Shift key, or a conversion of an uppercase character to its lowercase character.

ECMA
The European Computer Manufacturers Association standards organization.

encoding scheme
A set of rules for parsing a byte stream into a group of coded characters.

EUC
(Extended Unix Code) An encoding scheme defined by AT&T USO Pacific to support multi-byte character sets. EUC comprises a primary code set (CS0) which is 7 bit ASCII, and three supplementary code sets that can be any character set that the user chooses. These code sets are distinguished by the high bit of the code and the single-shift characters (SS2 and SS3).

external code
Character data that is used for system-to-system or system-to-peripheral communications. Equivalent to **transmission code**.

file code
Character data that the Operating Systems, subsystems, and application software uses.

GREEK8
The Hewlett-Packard supported 8-bit code set for the Greek language.

HEBREW8
The HP-supported 8-bit code set for the Hebrew Language.

Hindi digits
An alternate representation of numbers used in some Arabic countries. Other Arabic countries use the Latin representation of numbers.

HP8
HP proprietary implementations of several one-byte character sets. These codes are suitable for ISO 2022 encoding.

HP15
The HP encoding scheme for multi-byte coded character sets. It is used as file code. This encoding scheme is state independent. All code sets which adhere to the HP15 definition use a set of bytes with the high bit set to differentiate between 8 and 16 bit data. If the high order bit is zero, then the byte represents a one-byte ASCII character. Otherwise, it may represent the first byte of a two-byte character or a one-byte non-ASCII character; this information is provided in system tables.

HP16
The HP encoding scheme for multi-byte code sets used for communicating 8 and 16-bit data between a peripheral and a computer. It is derived from the ISO 2022 character processing standard. This encoding scheme is state dependent. It uses escape sequences to differentiate between single- and multi-byte characters.

ideogram or ideograph
A pictographic symbol used to represent whole words or syllables.

internal code
Equivalent to **file code.**

internationalization
Design and modification of products to make them localizable. For example, modification of application programs before compilation to make use of locale-independent library routines and to ensure that single-byte and multi-byte data can be handled in a locale-sensitive way by hardware and software.

ISO7
International Standards Organization 7-bit character substitution, in which the character graphics associated with some less-used ASCII codes are changed to other characters needed for a particular language.

ISO 8859-1
ISO defined single byte code set for Latin alphabet No.1 characters, or Latin-1. Used for most Western European languages and in many North and South American languages. Based on Part 1 of the ISO 8859 International Standard.

ISO 8859-2
ISO defined single byte code set for graphic characters used in many Eastern European countries. Based on Part 2 of the ISO 8859 International Standard.

ISO 8859-5
ISO defined single byte code set of 191 graphic characters identified as the Latin/Cyrillic alphabet. Based on Part 5 of the ISO 8859 International Standard.

JAPAN15
The HP-supported 16-bit code set for the Japanese language. HP15 encoding scheme is used.

JAPANEUC
The HP-supported 16-bit code set for the Japanese language. EUC encoding scheme is used.

KANA8
The HP-supported 8-bit code set for support of phonetic Japanese (Katakana).

Kanji
The Japanese ideographic characters based on Chinese characters. Kanji, as Chinese, is an open character set; no one really knows how many characters exist. The current JIS level 1 and 2 defines less than 10,000 characters.

Katakana
One of the sets of Japanese phonetic characters typically used for foreign words in writing. The set consists of 64 characters, including punctuation.

keyboard order
Characters arranged the way they are entered from the keyboard.

KOREA15
The HP-supported 16-bit code set for the Korean language.

LANG
The HP-UX environment variable (LANGuage) that should be set to the name of the locale corresponding to the native language to be used.

LANGOPTS
The HP-UX environment variable that defines the options for mode (Latin or non-Latin) and data order (keyboard or screen).

language
Associated, as listed below, with computer, native, natural, programming, or supported.

computer: An artificial language consisting of a set of characters and rules, with specific functions for computer programming. The `C` language is an example of a computer language.

native: The first language of the user. Alternatives are "national" or "local" language.

natural: The spoken or written language used by humans.

programming: Alternative to "computer language".

supported: The computer-implemented version of a written or spoken language. See `/usr/lib/nls/config` for a list of NLS-supported languages.

Latin mode
The mode where the terminal is configured so that the text display order is from left to right.

library
A set of subroutines contained in a file that can be accessed by a user program.

library routine
A subroutine contained in a library file used to perform a task.

literal
Computer code, displayed as it would appear in the output, or as it would be typed in.

locale
That part of the environment of a process which contains international data.

local environment files
Files external to the code of a software product containing locale-dependent information such as messages, prompts, commands, icons, etc.

localizability
The attribute of a hardware or software product which allows it to be localized through predefined steps (normally without redesign or recoding). The outcome of the internationalization effort.

localization
The adaptation of an internationalized hardware/software system for use in different countries or local environments.

localization center
An organization in a country or region that assists in providing software or hardware products specifically tailored for use in that country or region.

message catalog
The external file containing prompts, responses to prompts, and error messages that can be localized into a user's native language.

message catalog system
A set of tools developed by Hewlett-Packard to extract print statements from C programs and place them in, or retrieve them from, the message catalog.

mode
The order in which text is displayed: Latin (left-to-right), or non-Latin (right-to-left).

multi-bye character
See **character.**

n-computer (native-computer)
An invented, artificial computer locale which specifies the minimal environment. Now replaced by the `C` locale.

non-Latin mode
The mode where the terminal is configured so that the text display order is from right to left.

opposite language
When the terminal is in non-Latin mode, Latin characters are the "opposite language" and when the terminal is in Latin, non-Latin characters are the "opposite language". NLS allows both Latin and non-Latin characters to appear on the same line. Opposite language characters are inserted on the screen in the opposite direction by using an opposite language key.

order
The temporal order in which data is used: screen order (the order in which characters are displayed) or keyboard order (the order in which the user enters keystrokes).

path name
A sequence of directory names separated by slashes (/), and ending in any type of file name.

phonetic order
The ordering of characters by the way they are read or spoken.

PRC15
The HP-supported 16-bit code set for Simplified Chinese, the language of the People's Republic of China.

prelocalize; prelocalization
See **internationalization.**

process code
Character data that is used within a process.

radix character
The actual or implied character that separates the integer portion of a number from the fractional portion.

ROC15
The HP-supported 16-bit code set for Traditional Chinese, the language of the Republic of China.

root directory
The highest level directory of the hierarchical file system, in which other directories are contained. In HP-UX, the "/" refers to the root directory.

routine
See **library routine.**

screen order
The order in which characters appear on the screen.

shell
The shell is both a command language and a programming language that provides the user-interface to the HP-UX operating system.

shell script
A sequence of shell commands and shell programming language constructs, usually stored in a text file, for invocation as a user command or program by the shell.

single-byte character
A byte representing a single graphic symbol or control code.

space
A blank character. In ASCII a space is represented by character code 32 (decimal).

standard input
The source of input data for a command or a program. The default standard input is the terminal keyboard, but the shell may redirect the standard input from a file or a pipe.

standard output
The destination of output data from a command or a program. The default standard output is the terminal display, but the shell may redirect the standard output to a file or a pipe.

syntax
The rules governing sentence structure in a spoken language, or statement structure in a computer language such as that of a compiler program.

THAI8
The Hewlett-Packard supported 8-bit code set for the Thai language.

transmission code
Character data that is used for system-to-system or system-to-peripheral communications.

TURKISH8
The HP-supported 8-bit code set for the Turkish language.

upshifting
The means by which the peripheral produces uppercase letters by using the Shift key, or the conversion of a lowercase character to its uppercase character.

USASCII
A less common name for ASCII, the American Standard Code for Information Interchange.

variable
A storage location for data.

working directory
The current directory where the user's files will be placed by default.

WPI
Worldwide Portability Interface (functions which allow programming in a codeset independent manner). WPI is based upon the Multibyte Support Extension (MSE) to ISO/IEC 9899-1990.

X/Open
An international standards group dedicated to an open software standard. The group is concerned with standards selection and adoption, using International Standards where they exist.

Index

E

F

G

L

M

X

Y

Printed in USA E0892

Reorder No. or
Manual Part No.
B2355-90036

**Manufacturing
Part No.
B2355-90036**

B2355-90036