Native Language Support HP-UX Concepts and Tutorials

HP 9000 Series 300/800 Computers

HP Part Number 97089-90058

HEWLETT PACKARD

Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

Legal Notices

Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. However, minor changes may be made at reprint without changing the printing date. The manual part number will also change when extensive changes are made.

Manual updates may be issued between editions to correct errors or to document product changes. To ensure that you receive updates or new editions, you may subscribe to the appropriate product support service, available from your HP sales representative.

September 1989. First Edition

Contents

1 Using this Manual

This manual is for people who are using, writing, or translating programs in a multi-lingual environment and who will need to make use of the various elements of Native Language Support (NLS).

You will find specific sections of this manual to be written at a technical level appropriate to general users, system administrators, NLS coordinators, and applications programmers.

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

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

To find this information ...	Please see ...
Using this Manual: This chapter explains the typographical conventions used in this manual and identifies other manuals referenced in the contents.	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, messages, and internationalization.	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
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.	Chapter 4
Administering International Software (for the system administrator and the NLS coordinator): This chapter identifies the HP-UX directories and files, how to set up the user environment, installing message catalogs and optional locales, and configuring terminals and peripherals.	Chapter 5
Localizing International Software (for the system administrator and the NLS coordinator): This chapter explains the details of localization for special user requirements, localizing message catalogs, and creating specialized locales.	Chapter 6
Advanced NLS Features (for the programmer:). This chapter explains the NLS character- and string-processing tools, processing non-Latin character input/output, and special treatment of locales and message catalogs.	Chapter 7

To find this information ...	Please see ...
Examples of Internationalized Software: Character processing, collation, monetary formatting, messaging, and date/time.	Appendix A
NLS References: An alphabetic listing of *HP-UX Reference* locations for all NLS commands and routines.	Appendix B
Previous Usage: Tables of current and obsolete NLS commands and routines.	Appendix C
Languages and Codesets: A listing of the native languages that are supported by HP codesets.	Appendix D
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*.
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.

- 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.
- The *HP-UX Portability Guide* documents the guidelines and techniques for maximizing the portability of programs written on and for HP 9000 Series 200, 300, and 500 computers running the HP-UX operating system. It covers the portability of high-level source code (C, Pascal, FORTRAN) and transportability of data and source files between commonly used formats.
- *HP-UX System Administration Tasks* 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.

- *HP-UX Concepts and Tutorials: Facilities for Series 200, 300, and 500* contains valuable guidance for setting up your terminal and configuring the softkey definitions.
- The *NLIO System Administrator's Guide* provides installation and configuration procedures for NLIO, which is the set of servers and filters used to input and output 16-bit characters efficiently on 16-bit hardware. It also contains fileset and font descriptions for the supported languages.
- The *Native Language I/O Access User's Guide* describes how to use the NLIO system.
- The *NLIO Code Books* and the *NLIO Input Method Guides* describe the language-specific codes and the input methods used for the supported languages.
- *Finding HP-UX Information* (2 vols.) provides a cross-index, detailed descriptions, and part numbers for the Series 300 and 800 HP-UX manuals.
- 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.

2

Introduction to NLS

Overview of Software Internationalization

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

To do so effectively, software products are required to 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.

There are two processes involved in the NLS approach to enhancing software for international use:

- The process of **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 for translated messages and prompts, and changing functions and conventions to comply with local requirements. For a number of reasons, it is also desirable that such internationalization be accomplished with a minimum of change to the program code itself.
- The process of **localizing** adapts the software to a particular locale, including the translation of messages and the use of appropriate language tables on the local system.

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

- Preserving the integrity of the data
- Proper handling of characters
- Appropriate message interfacing in the local 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 then 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 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.
- 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.

Hewlett-Packard's Native Language Support has been adopted as the basis for the *X/Open Portability Guide (XPG) Issue 3*. HP has an ongoing test process to ensure compliance with applicable standards of POSIX and ANSI-C

What is Native Language Support (NLS)?

NLS provides a number of features to aid the international user:

- 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.

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

- The replacement of the 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`.
- The provision of tools for copying all hard-coded messages into external message catalogs and for updating the message catalogs.

Localization then adapts the internationalized software application or system for use in a specific linguistic environment. This includes translating the text in the message catalogs into the local language.

Message catalogs and language tables can be specified at run time, rather than having the messages compiled into the programs. For a given piece of software, this message cataloging process only needs to be done once.

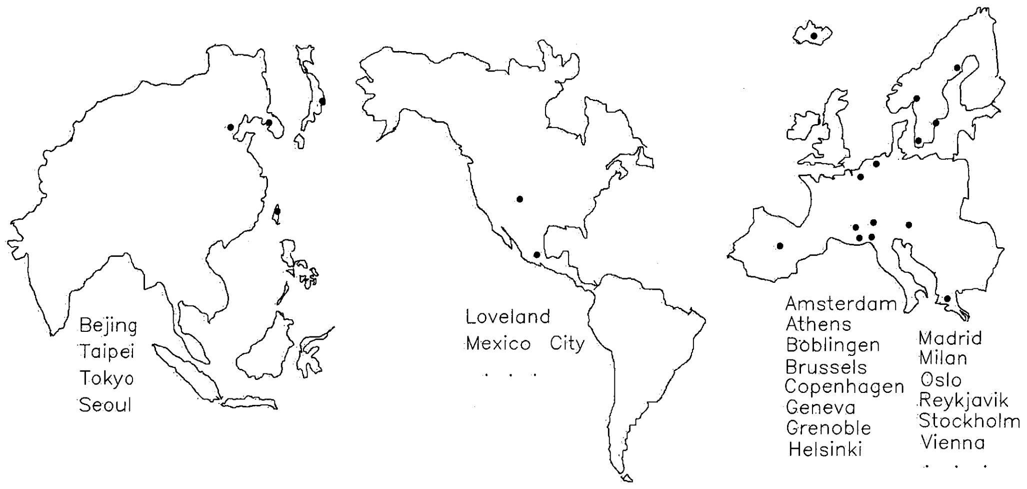

Figure 2-1. Hewlett-Packard Localization Centers

Localization and internationalization can often be facilitated by Localization Centers operated by various Hewlett-Packard Country Product Organizations, some of which are shown above.

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

Character and Text Handling

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

- **Character Sets.** 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 even sufficient to span the Latin based alphabet used in many European languages. And yet, for many Asian languages, character sets can contain several thousand members. 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 is used to support languages that have additional characters, accented vowels, consonants with special forms, and special symbols.
- Multi-byte characters may be used for character codesets which are exceptionally large.

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, HP has already defined character sets which permit the processing of several European, Middle Eastern, and Asian languages.

For European and Middle Eastern languages, HP has defined a series of 8-bit character sets. Every HP 8-bit character set is a superset of ASCII. The HP-supported 8-bit character set for Western European languages is ROMAN8. Other 8-bit character sets are defined for other locales. For a listing, please refer to Appendix D, "Languages and Codesets".

For alphabets of more than 256 characters, such as **Kanji** (a Japanese ideographic character set), multi-byte character codes are required. HP 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, HP provides support for the Japanese UJIS character set. These are used for data processing and storage. For input and output, HP uses a multi-byte character encoding scheme called HP-16. Appendix D lists both single- and multi-byte codesets available from HP.

Users can also define their own languages using `buildlang` with non-HP defined codesets. For more information on `buildlang`, see Chapter 6, "Localizing International Software", in this manual.

- **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 character set has 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.
- **Collation.** Each languages 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 dictionary sorting. Each language may order the characters in its character set differently, and certain character sets 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
  - The number of strokes required to represent the character
  - The radical (root) of the character
  - The number of strokes added to the radical

Comparing Strings and Comparing Characters

The order into which character strings are sorted is language-dependent. Traditionally, most comparative ordering is based on ASCII values. But, with the extension of the ASCII character set to ROMAN8 for support of other languages, the ordering of a character within a character set no longer coincides with the character's traditional alphabetical order.

For example, "ä" *follows* "b" in the character set ordering but is *sorted before* "b" in many cases. This situation makes sorting based on the code of each character inappropriate when internationalizing software.

In addition, sorting based on character code does not provide true dictionary sorting even in the case of the ASCII character set. Dictionary order sorts "a" after "A" and before "B", whereas ASCII based order sorts "a" after both "A" and "B". The following is an example of sorting the same list based on the C sorting method, and based on a German sorting method.

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 the chapter "Developing International Software", in this manual.

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 *Text Editors and Processors, HP-UX Concepts and Tutorials.* 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" (e, é, ë, è, etc.).

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 so extensive. For example, multi-byte characters are allowed as single character elements in these 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.

Local Customs and Conventions

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

- **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 the U.S.)
- □ The digit grouping symbol (the comma in the U.S.) which serves to separate groups of integers
- □ The convention for grouping integer digits (by three’s in the U.S.)

In the U.S., a number is represented as follows:

```
2,345.678
```

But when representing the same number in France, the decimal point and the digit-grouping symbol are reversed:

```
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 the U.S. and France, the symbols are transposed.

  ```
  $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. 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 ruling emperor.

- **Display of Days, Weeks, Months.**

  Names for days of the week and months of the year may vary with language. Rules for abbreviating these also differ. The order of the year, month, and day, as well as the separating delimiters, are not universally defined. For example, October 7, 1986 would be represented in the U.S. as:

  ```
  10/7/86
  ```

  in Germany, it would be represented as:

  ```
  7.10.1986
  ```

  and in the U.K. as:

  ```
  7/10/86
  ```

- The chapter "Advanced NLS Features" in this manual, describes the library routines used to handle these local customs.

Messages

The ability to customize messages for different countries is an important aspect of using NLS. NLS enables you to choose the language to be used 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 is 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 command)

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 the chapter "Localizing International Software", in this manual.

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 NLS commands successfully. The information and the tasks are minimal because, in most situations, you will be receiving help from two people on your staff:

- Your local NLS Coordinator, whose tasks are:
  - Advising on optimal use of NLS features
  - Ordering NLS software
  - Communicating special configuration needs to your System Administrator
  - Installing message catalogs
  - Coordinating translation activities
- Your System Administrator, whose tasks are:
  - Installing and updating the operating system
  - Configuring the system
  - Installing and initializing additional software
  - Maintaining system software

Your System Administrator should have already provided the appropriate configuration and initialization.

NLS Environment Variables

Internationalized commands adapt their behavior to that specified by a set of NLS environment variables. These variables indicate user requirements for various aspects of NLS capabilities:

`LANG`	Specifies native language, local customs, and coded character set and messages.
`LC_COLLATE`	Specifies string collation.
`LC_CTYPE`	Specifies character classification and case conversion.
`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.

By setting these variables, you can cause internationalized software to perform in a manner appropriate to your particular needs. For additional information on the NLS environment variables and their use, see *environ*(5) in the *HP-UX Reference.*

Setting Your Environment

Local system default values for the NLS environment variables are ordinarily determined by your local NLS Coordinator and set by your System Administrator. These system default values are what you get when you log in unless you make provisions for something different.

You can determine the setting of your NLS environment variables by typing:

```
env
```

If none of the NLS environment variables is set, as indicated by `env`, or if you are not sure what NLS environment you need, consult with your NLS Coordinator to determine the appropriate settings for your locale.

If the local system default values are not satisfactory, you can get the NLS environment you need by setting the environment variables appropriately.

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
```

It is generally convenient to add these commands to your `.profile` or `.login` file so that your preferred environment will be set when you login.

If you will be running applications that need an NLS environment different from the system default and different from your individual environment, it is convenient to 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. For further information on terminal configuration, see *Facilities for 200/300/500: HP-UX Concepts and Tutorials.*

To use international software, your terminal should also be set so that single-byte 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" 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 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 created.

See the chapter "Localizing International Software", in this manual, for more information on localizing message catalogs.

Using Internationalized Commands

To see what locales are installed on your system run:

```
nlsinfo
```

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.

4

Developing International Software

Read this chapter if you are:

- A programmer for the local system

This chapter covers the standard programming issues for:

- Developing international software
- Internationalizing existing software

For a discussion of special cases see the chapter “Advanced NLS Topics”.

General Programming Issues

The programming issues your software must accommodate are:

- Initialization
- Preservation of data integrity
- Character and string processing
- Messaging

Initializing NLS

When you work with internationalized software, it is always necessary to provide the appropriate NLS initialization, and, in some cases, it is also necessary to use NLS library routines rather than conventional library routines. More extensive programming changes may be needed in special cases.

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

- The program locale
- The program messages

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.

The two initialization routines are independent. The program's locale does not affect messaging, and messaging does not affect the program's locale.

Recommended Initialization

For most applications, the following "standard" initialization is recommended:

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

With this initialization, all LC_*categories* will be set to the value of LANG, except for those categories in which the corresponding environment variable is set to another valid locale. For environment variables that are set to a valid locale, the value of the environment variable will override the value of LANG for that category. If the value of `LANG` is not set or is set to the empty string, then the `C` locale is used.

With this initialization, `LANG` and `NLSPATH` specify a series of paths to search for a message catalog. If a catalog is found on one of these paths, messages issued by the program will be messages from the selected message catalog. If a message catalog is not found, messages issued by the program will be `C` locale messages.

Note that even if `setlocale` is successful, it is possible for `catopen()` to fail.

This "standard" initialization assumes that messaging uses the "standard" default messages described in the section "Programming for Messages" below. For special cases, a non-standard initialization may be required. See the chapter "Advanced NLS Topics" in this manual for more information.

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 nor 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 a coding 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.

To aid in processing multi-byte codesets and avoid the byte redefinition problem, there are two sets of routines available to the programmer.

Programming with Multi-byte Characters

For dealing with HP's multi-byte codesets, see *nl_tools_16(3C)* in *HP-UX Reference* which describes a set of byte-status macros: `FIRSTof2`, `SECof2`, and `BYTE_STATUS`. These macros can be used to determine whether a byte value represents an single-byte character or part of a multi-byte character.

Probably more useful, are character pointer macros that are analogous to byte pointer operations:

Macro Call	**Byte Pointer Analog**
`CHARAT(p)`	`(*p)`
`ADVANCE(p)`	`(p++)`
`CHARADV(p)`	`(*p++)`
`WCHAR(c, p)`	`(*p = c)`
`WCHARADV(c, p)`	`(*p++ = c)`

These macros operate on byte pointers, but they make the appropriate calls to `FIRSTof2`, etc., and advance the pointer one or two bytes as needed.

These macros are not always needed. For example, the following program will correctly copy single-byte as well as multi-byte character strings.

```
char *f, *t;
    :
while ( *t++ = *f++ );
```

However, to copy only the printable characters of a character string requires special treatment. The following program will work for single-byte codesets but not for multi-byte codesets because of the byte redefinition problem: it is possible for the second byte of multi-byte character to be a non-printable ASCII character. Such a byte would not be copied to the destination string.

```
#include <ctype.h>
    :
char *f, *t;
int c;
    :
while ( c = *f++ )
```

```
        if ( isprint(c) )
            *t++ = c;
    *t++ = c;
```

Using CHARAT macros, the preceding program can be made to operate correctly on single- and multi-byte data:

```
    #include <ctype.h>
    #include <nl_ctype.h>
        :
    char *f, *t;
    int c;
        :
    while ( c = CHARADV(f) )
        if ( c > 255 || isprint(c) )
            WCHARADV(c, t);
    WCHARADV(c, t);
```

Note that `isprint()` is defined for single-byte codesets only and that all multi-byte characters (c > 255) are considered printable.

Note Although these macros seem transparent, there are some cautions that must be observed when using them.

- First, they cannot determine byte status for an arbitrary byte within a string. In general, multi-byte strings must be examined sequentially from the beginning.
- Second, the macros are not perfect analogs of the byte pointer versions. In particular, the program sequence:

  ```
  *t++ = *f++;
  ```

  cannot be done as:

  ```
  WCHARADV(CHARADV(f), t);
  ```

  It must be done as:

  ```
  int c;
      :
  c = CHARADV(f), WCHARADV(c, t);
  ```

- Using the macros will increase program size and reduce performance. For example, when *t++ = *f++ is converted to CHARAT macros, it generates about 350 bytes of additional code. Where size is a problem, the function versions of the macros can be used at some reduction in performance.

- The extent of size and performance impact is application dependent. To reduce this impact, a common strategy for processing multi-byte character data is to use byte pointer operations where character interpretation is not an issue, and to use multi-byte routines only where needed.

See *NL_TOOLS_16(3C)* in the *HP-UX Reference* for more information on programming with multi-byte characters.

Programming with Wide-Characters

For some applications, character processing may be more convenient if multi-byte characters are represented as constant width characters—so-called wide-characters.

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 for some things, for example, pointer manipulation works without the need for the `FIRSTof2` type of macros.

However, it is less convenient for others. For example, multi-byte string manipulation routines such as `strcoll()` and `printf()` do not work for wide-character strings.

The "copy printable characters" example, written to use wide-characters, would appear as the following:

```
#include <ctype.h>
#include <stdlib.h>
   :
char fm[NM], tm[NM];
wchar_t fw[NW], tw[NW];
wchar_t *f = fw, *t = tw;
int c;
   :
mbstowcs(fw, fm, NW);    /* convert multi-byte to wide-character/
while ( c = *f++ )
    if ( c > 255 || isprint(c) )
        *t++ = c;
*t++ = c;
wcstombs(tm, tw, NM);    /* convert wide-character to multi-byte/
   :
```

Note

- The issue of printable characters is handled as above.
- Error-checking the conversion between multi-byte and wide-character data is omitted.
- `NM` and `NW` are assumed to be appropriately defined.

Conversion of Existing Programs

When internationalizing an existing program, conversion to preserve data integrity is conveniently done in two steps:

1. Conversion to single-byte data.
2. Conversion to multi-byte data.

Conversion to single-byte data can be subtle. Some programs use the 8th bit as a flag to indicate special treatment of the 7-bit character. In general, it may not be easy to determine whether a program does this. In any event, programs that use or remove the 8th bit must be changed. If the 8th bit is used for data, it will be necessary to put the 8th bit data in a new data structure and it may be necessary to design a new algorithm to access the new data structure.

Once a program is correct for single-byte data, the conversion to multi-byte data is straightforward. No structural changes are needed, but proper handling of multi-byte characters is needed.

As we saw above, for example, multi-byte data cannot be tested with `isprint()`. In general, it is necessary to examine each instance of byte processing to determine whether special handling of multi-byte data is needed.

Character and String Processing

Character and string processing for international software must ensure that local customs are observed regarding such things as

- Treatment of accented characters
- Formatting of date and time
- Formatting of numeric and monetary quantities
- Comparison of string data

Most character and string processing is provided by internationalized library routines that give correct results for the currently active locale. Note that there may be restrictions in the use of some library routines and minor program changes may be needed. You can find more information in *NL_TOOLS_16(3C)* in the *HP-UX Reference.*

The *ctype(3C)* routines `isalpha()`, `isupper()`, etc. and the *conv(3C)* routines `toupper()`, etc., are internationalized and give locale-sensitive results. However, they are defined only for single-byte data and cannot be used for multi-byte data.

The numeric formatting routines `ecvt`, `gcvt`, `strtod`, `atof`, `printf`, `fprintf`, etc., 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), strod(3C),* and *printf(3C)* in the *HP-UX Reference, Section 3C.*

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

Generalized monetary formatting is more involved than numeric formatting since in some countries the currency symbol is placed before the amount. In other countries the currency symbol is placed 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 `localeconv` and can be used as:

```
      :
  #include <locale.h>
      :
  struct lconv *lcs;
  float number;
  char *cs_p, *cs_f;
      :
  lcs = localeconv();
      :
  number = ...
      :
  if ( number >= 0 && lcs->p_cs_precedes == '1' ||
      number < 0 && lcs->n_cs_precedes == '1' ) {
      cs_p = lcs->currency_symbol;
      cs_f = "";
      }
  else {
      cs_p = "";
      cs_f = lcs->currency_symbol;
      }
  printf("%s %6.2f %s\n", cs_p, number, cs_f);
      :
```

Other information in the `lconv` structure describes decimal point, thousands separator, spaces used with the currency symbol, etc.

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

For some applications, a performance improvement may be obtained 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.

Conversion of Existing Programs

Conversion of existing programs is necessarily an ad hoc process. The `grep` command can be used on existing source code to find calls to routines, such as `ctime()` and `strcmp()`, which may require changes.

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:

- 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.
- Create a message text source file for localization. This file contains messages that would ordinarily appear as quoted strings in the source program.
- 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 `gencat` command, which 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

Message catalogs are opened by the `catopen()` routine:

```
#include <nl_types.h>
    :
nl_catd catd;
    :
catd = catopen("name", 0);
```

where the *name* argument identifies the catalog to be opened. If `catopen()` can successfully open the identified catalog, it returns a message catalog descriptor. Otherwise it returns `(nl_catd)-1`. The program can test this return value and take an appropriate action if the requested catalog cannot be opened.

Note

- The catalog descriptor `catd` is used by `catgets()` and consequently it must be accessible to every `catgets()` call.
- It is recommended that the program name be used as the *name* argument.

Search Path and Naming Conventions

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.

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. Message catalog naming conventions can be defined by means of substitution field descriptors that permit the use of run-time information. For example:

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

This specifies two paths, separated by `:`, to be searched for a message catalog. The meta character, `%`, 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 4-1. Summary of NLSPATH Replacement Specifiers

Replacement Specifiers	Expansion by NLS
`%L`	replaced by the value of LANG
`%N`	replaced by the name of the application
`%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.*

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" below.

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 arranged 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 will return `(nl_catd)-1`, an invalid file descriptor. This will subsequently cause `catgets()` to fail, and it will 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 a 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.

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:

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

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("hello", 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

- 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);
  ```

  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. For further information, see *catgets(3C)* and other references in *HP-UX Reference*.
- The message system makes no provision to ensure 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 the following:

```
    :
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;
    }
    :
```

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 Program 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" below 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 are restricted to the routine so they do not conflict with 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.

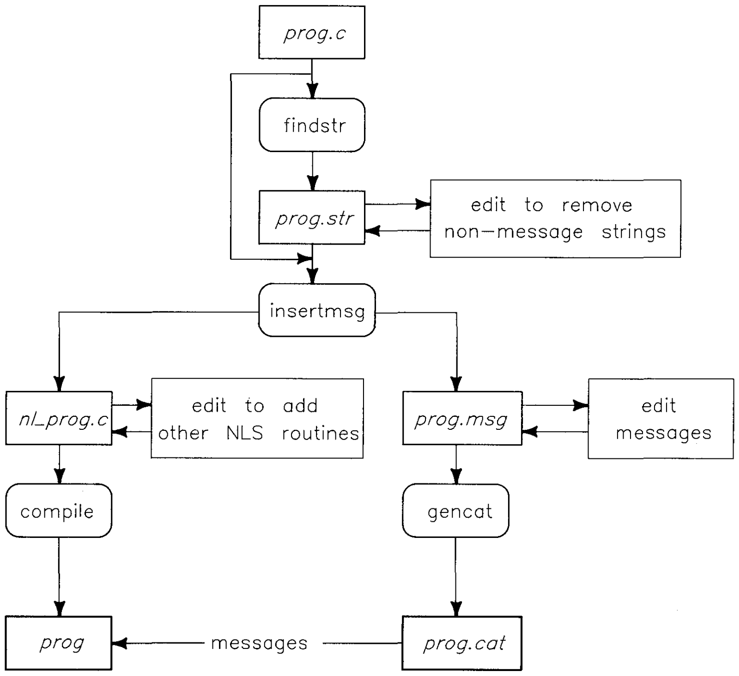

Figure 4-1. Converting a Non-Internationalized Program

Conversion of Existing Programs for NLS Messaging

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.

1. Finding Strings in a Program

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.

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.

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`.

Note The `findstr` or 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.

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 "Initializing NLS" in this chapter for additional information.

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

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.

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 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 *HP-UX Reference, Section 3C*.

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, set_num, 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

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 “Example 2”, in “Appendix A” of this manual for more 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.
- 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.
- 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 the program to continue to operate with a default language (such as `C`).

5

Administering International Software

Read this chapter if you are:

- A Systems 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.

Both the information and the tasks are minimal since your local NLS Coordinator should have already determined the required configuration and initialization of the system with respect to NLS.

Finding NLS Files

The NLS information used by HP-UX commands and libraries is located in the following 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 `buildlang`. 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.def*	These files contain locale-dependent processing information.
`/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.8859
german_swiss
german_swiss.8859
    :
japanese
japanese.ujis
    :
```

Then, you should expect to find the following directories:

```
/usr/lib/nls/german/8859
/usr/lib/nls/german/swiss
/usr/lib/nls/german/swiss/8859
/usr/lib/nls/japanese
/usr/lib/nls/japanese/ujis
```

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 local NLS Coordinator. 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.

Terminal Configuration

Users running internationalized commands will be using the following setting:

```
stty -istrip -parity
```

The `/etc/gettydefs` file for these users should be set properly for their terminal.

Installing Message Catalogs

Localized message catalogs would ordinarily be delivered to you by the local NLS Coordinator. You should install these catalogs in the appropriate location.

Message catalogs for HP-UX commands and libraries are 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 local NLS Coordinator. This location and naming may require a change to the system default value of `NLSPATH`. If it does, the NLS Coordinator will determine the new value. You will need to make the required change to the `NLSPATH` setting in `/etc/profile` and `/etc/csh.login` and you will want to notify users 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 *UP-UX System Administration Tasks.*

HP-UX is shipped with the default locale, `C`. For specific locations, other locales may also be shipped. If you install other products, however, you must order the specific locales for them as an additional option. Not all character sets are supported on all peripherals, 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 character sets that your peripheral(s) will need to support. Hewlett-Packard provides printers, plotters and terminals which support HP single- and multi-byte character sets, as well as non-HP standards (such as the ISO8859-1 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.

Because of these considerations, the information below is provided to help in understanding the special characteristics of non-ASCII peripherals. For further information, you can contact your local HP sales representative for assistance.

European Character Sets

For European languages, many HP peripherals support the ROMAN8 character set. ROMAN8 is a full superset of ASCII and offers 88 additional local language symbols. Older HP peripherals may use the HP Roman Extension set, which is a subset of ROMAN8. Roman Extension is missing ROMAN8 characters À through Ï, Ù, Û, Ç, ¥, §, *f*, and Á through ±.

ROMAN8 terminals can simultaneously display any characters in the set. The keyboards have keycaps only for the specified local language, but, in the 8-bit mode, you can enter any ROMAN8 character by use of the (Extend char) key. You can also use most 8-bit terminals in ISO7 mode.

Katakana Character Sets

Many HP peripherals support a base 8-bit character set known as KANA8. The first 128 codes in the KANA8 set are JASCII (the same as ASCII except that the set substitutes "¥" for "\"), and the last 128 codes are available for Katakana.

Other 8-bit HP Character Sets

As with KANA8, the other 8-bit character sets supported by HP have ASCII as the first 128 codes, with the last 128 codes used by other characters. Some Arabic printers are capable of context-sensitive letters, so some character shapes may vary on these devices.

16-bit HP Character Sets

For Asian languages, many HP peripherals support one of five HP-16 character sets. These character sets are compatible with the five HP-15 character sets (PRC15, ROC15, JAPAN15, UJIS, and KOREA15). NLIO is required for converting between HP-15 and HP-16 during input and output. NLIO is also necessary with some Asian terminals to provide the "input method" by which a user can input multibyte characters using a conventional keyboard. Certain peripherals, such as PC's used as terminals, can generate and display HP-15 multibyte characters directly and need no additional software.

Non-HP 7-Bit Character Sets

The ISO7 (International Standards Organization 7-bit character substitution) and similar character sets have certain infrequently-used ASCII codes, such as those for "|" and "{", designated to generate local-language symbols. Examples are the ø or æ in Danish. Unfortunately, the designated ASCII codes also represent special characters often used in HP-UX (and all other UNIX and UNIX-like systems). For this reason, the use of ISO 7-bit, and similar non-HP international character sets is neither recommended nor supported.

Limited support for non-HP 8-bit character sets may be provided through appropriate language definitions. Currently this definition must be provided by the user. The `buildlang` utility described in the chapter "Localizing Internationalized Software", in this manual, provides help in defining your own language and locale characteristics.

6

Localizing International Software

Read this chapter if you are:

- A local NLS Coordinator.

The chapter covers information and tasks for localizing commands that have been internationalized. It will also help you in determining local NLS needs which you may need to communicate to your System Administrator.

Localizing the User Environment

HP-UX does not automatically set NLS environment variables. HP-UX commands, when run with NLS environment variables not 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, you will need to specify the desired default values for the NLS environment variables:

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

The chosen values should be those most commonly used. The default values should be set in `/etc/profile` and `/etc/csh.login`. You should arrange with your system administrator 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.
- Arranging for translation of 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.

Preparing for Translating Messages

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.

The developer should have provided a translator's "cookbook". Lacking this, here are some possible translation problems you might encounter:

- The meaning of a message 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 possible solutions you might try:

- Experiment with the program to see if you can determine the intended behavior.
- Communicate with the developer of the program.
- Communicate with 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. 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 delivered to your System Administrator 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. In the event that none of the existing locales is appropriate, it is possible to 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 `buildlang` 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 `american` except that it is to have 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 `buildlang` script of the `american` locale by executing:

```
buildlang -d american > new_locale
```

You can now modify the buildlang script `new_locale` 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
    :
```

To get the desired formatting, you need to 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
    :
```

You also 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. If you want to replace an existing locale with a new definition, these must be the *locale_name* and *locale_id* of the locale that is to be replaced.

After you have changed `new_locale`, the *locale_name* locale can be installed in the system by executing:

```
buildlang 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.def` exists.
- Set `LANG` to the *locale_name* locale and verify that `date` formats the date as desired.

7

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 conversion tables and special treatment that may be needed in the conversion. See *iconv(3C)* in the *HP-UX Reference* for details.

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 keystrokes 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
```

Order and mode information is specified by the `LANGOPTS` environment variable. To set LANGOPTS using Bourne Shell or Korn Shell:

```
    LANGOPTS=mode_order
    export LANGOPTS
```

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 A, "Examples of Internationalized Software", in this manual.

Locale Information

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

Programmatic access to information about the currently active locale is provided by three library routines. The `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:

```
10/31/87
```

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

```
31/10/87
```

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

Examples of Internationalized Software

Example 1: Rtlcat

The following is the first of two example programs given to illustrate the usage of NLS routines.

Program Description and Comments:

```
/*
** This program is used to illustrate several Internationalization
** features including:
** - message catalogs
** - setlocale(3c)
** - right-to-left processing
** - some multi-byte in get_basename()
** Syntax:
** rtlcat [options] [files ... ]
** Options:
** -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
** Description:
** Do a right-to-left cat.
**
** Rtlcat 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/order is gotten from the LANGOPTS
** environment variable (environ(5)).  The terminal mode/order
** is 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 hp150 and hp2392
** terminals.  Rtlcat assumes the terminal is the stdout device.
**
** If the input file mode/order and the terminal mode/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 sucessful 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;
}
```

Example 2: Makefile

```
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
```

B

NLS References

Following is a list of current NLS documentation in the *HP-UX Reference.*

BUILDLANG(1M)	buildlang - generate and display locale.def file
CATGETS(3C)	catgets - get a program message
CATOPEN(3C)	catopen, catclose - open and close a message catalog for reading
ENVIRON(5)	environ - user environment
FINDMSG(1)	findmsg, dumpmsg - create message catalog file for modification
FINDSTR(1)	findstr - find strings for inclusion in message catalogs
FORDER(1)	forder - convert file data order
GENCAT(1)	gencat - generate a formatted message catalog file
HPNLS(5)	hpnls - HP Native Language Support (NLS) Model
ICONV(1)	iconv - code set conversion
ICONV(3C)	iconvsize, iconvopen, iconvclose, iconvlock, ICONV, ICONV1, ICONV2 - code set conversion routines
INSERTMSG(1)	insertmsg - use findstr(1) output to insert calls to catgets(3C)

LANG(5)	lang - description of supported languages
LANGINFO(5)	langinfo - language information constants
LOCALECONV(3C)	localeconv - query the numeric formatting conventions of the current locale
NL_LANGINFO(3C)	nl_langinfo - language information
NLJUST(1)	nljust - justify lines, left or right, for printing
NLSINFO(1)	nlsinfo - display native language support information
SETLOCALE(3C)	setlocale, getlocale - set and get the locale of a program
STRORD(3C)	strord - convert string data order
FINDMSG(1)	findmsg, dumpmsg - create message catalog file for modification

C

Previous Usage

The items identified under PREVIOUS have been superseded by the corresponding item under CURRENT. They are supported but will be withdrawn at some time. Continued use is not recommended.

Previous	Current	Reference	Notes
BYTE_STATUS	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
byte_status	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
catgetmsg	catgets	catgets(3C)	Withdrawn by X/Open
catread	catgets	catgets(3C)	
CHARAT	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
FIRSTOF2	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
firstof2	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
fprintmsg	fprintf	printf(3C)	
idtolang	(none)		
langid(5)	(none)		
langinfo	nl_langinfo	nl_langinfo(3C)	

Previous	Current	Reference	Notes
langinit	setlocale	setlocale(3C)	
n-computer	C	lang(5)	See discussion in *lang*(5)
nl_asctime	nl_ascxtime	ctime(3C)	
nl_asctime	strftime	strftime(3C)	
nl_atof	atof	strtod(3C)	
nl_ctime	nl_cxtime	ctime(3C)	
nl_fprintf	fprintf	string(3C)	
nl_fscanf	fscanf	string(3C)	
nl_gcvt	gcvt	ecvt(3C)	
nl_isalpha	isalpha	ctype(3C)	
nl_isctrl	isctrl	ctype(3C)	
nl_isgraph	isgraph	ctype(3C)	
nl_isprint	isprint	ctype(3C)	
nl_isspace	isspace	ctype(3C)	
nl_isupper	isupper	ctype(3C)	
nl_sprintf	sprintf	string(3C)	
nl_sscanf	sscanf	string(3C)	
nl_strcmp	strcoll	string(3C)	
nl_strtod	strtod	strtod(3C)	
PCHARADV	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
PCHARADV	WCHARADV	nl_tools_16(3C)	Use of multi-byte routines recommended for portability
PCHAR	WCHAR	nl_tools_16(3C)	Use of multi-byte routines recommended for portability

Previous	Current	Reference	Notes
printmsg	printf	printf(3C)	
SECOF2	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
secof2	mbtowc	multibyte(3C)	Use of multi-byte routines recommended for portability
sprintmsg	sprintf	printf(3C)	
strcmp[8|16]	strcoll	string(3C)	
strncmp[8|16]	nl_strncmp	string(3C)	
WCHARADV	mbtowc	multibyte(3C)	

D

Languages and Codesets

Following are native languages and the HP codesets that support them.

Language	Codeset
American	ROMAN8
Arabic	ARABIC8
Canadian French)	ROMAN8
Chinese-s	PRC15
Chinese-t	ROC15
Danish	ROMAN8
Dutch	ROMAN8
English	ROMAN8
Finnish	ROMAN8
French	ROMAN8
German	ROMAN8
Greek	GREEK8
Icelandic	ROMAN8
Italian	ROMAN8
Japanese	JAPAN15
Japanese.ujis	UJIS
Katakana	KANA8
Korean	KOREA15
Norwegian	ROMAN8
Portuguese	ROMAN8
Spanish	ROMAN8
Swedish	ROMAN8
Turkish	TURKISH8
Western Arabic	ARABIC8

Glossary

Note For additional information on terms used with HP-UX, please see the Glossary section of *HP-UX Reference, vol. 1.*

adaptation
As used in this document, adaptation is the process of making a product and all that goes with it (including documentation, training, distribution, support, etc.) suitable for, and available to, markets outside the country of its origin. Adaptation includes, but is not limited to, internationalization and localization.

alternate character set
A codeset 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 codeset for the Arabic language.

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 set

A set of symbols required to write a language. Different languages often have different character sets.

coded character set

See **codeset.**

codeset

A set of unambiguous rules that establishes a one-to-one relationship between each character of a character set and the numeric representation for that character.

7-bit: A codeset that uses seven bits to represent a collection of characters, control codes, and the space character. A 7-bit codeset allows a maximum of 128 characters which does not accomodate international languages. ASCII is an example of a 7-bit codeset.

8-bit: A codeset that uses all eight bits of a single byte to encode each character in the codeset. These codesets 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 codeset 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. A full 16 bits (2-bytes) per character allows definition of 65,536 unique character codes. The HP-15 encoding scheme limits practical use of all 16 bits, thus limiting the size of the codeset to 49,284 characters. The HP-16 encoding scheme limits the size to 35,344 characters. 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 codeset.

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. 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.

ECMA
The European Computer Manufacturers Association standards organization.

GREEK8
The Hewlett-Packard supported 8-bit codeset for the Greek language.

Hindi digits
An alternate representation of numbers used in some Arabic countries. Other Arabic countries use the Latin representation of numbers.

HP-8
The HP implementation of the ISO (International Standard Organization) 8-bit character codeset.

HP-15
The HP encoding scheme for internal operating system representation of 16-bit data that uses only 15 bits for characters.

HP-16
The HP encoding scheme for 16-bit codesets used for communicating 8- and 16-bit data between a peripheral and a computer. This is derived from the ISO (International Standards Organization) multi-byte character processing standard. By using 16-bit data 35,344 characters can be represented.

ideogram or ideograph
A pictographic symbol used to represent whole words or syllables.

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.

JAPAN15
The HP-supported 16-bit codeset for the Japanese language.

KANA8
The HP-supported 8-bit codeset for support of phonetic Japanese (Katakana).

Kanji
The Japanese ideographic codeset based on Chinese characters. The set consists of roughly 50,000 characters.

Katakana
The Japanese phonetic codeset typically used in formal 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 codeset for the Korean (Hangul) 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:

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.

local customs
The standard way dates, times, currency, numeric quantities, and collation are written in a particular region or country. Also known as country or local conventions.

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. Localization centers are responsible for the construction and/or translation of these files.

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 provides 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 in the 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).

n-computer (native-computer)
An invented, artificial computer locale which specifies the minimal environment for C translation. Now replaced by the `C` locale.

Native Language Support (NLS)
The HP set of software facilities within the HP-UX system which supports proper handling of native language data, including character data, country formatting conventions, and other local customs.

non-Latin mode
The mode where the terminal is configured so that the text display order is from right to left.

NLS Coordinator
This person handles the responsibility for localization of software and may also participate in installing and administering NLS aspects of a system.

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 codeset for Simplified Chinese, the language of the People's Republic of China.

prelocalize; prelocalization

See **internationalization**

programming language

See **language.**

radix character

The actual or implied character that separates the integer portion of a number from the fractional portion.

ROC

The HP-supported 16-bit codeset for Traditional Chinese, the language of the Republic of China.

ROMAN8
The HP-supported 8-bit codeset for Europe.

routine
See **library routine.**

screen order
The order in which characters appear on the screen.

syntax
The rules governing sentence structure in a spoken language, or statement structure in a computer language such as that of a compiler program.

TURKISH8
The HP-supported 8-bit codeset for the Turkish language.

upshifting
The means by which the peripheral produces uppercase letters by using the (Shift) key.

USASCII
A less common name for ASCII, the American Standard Code for Information Interchange.

X/Open
An international standards group dedicated to creating a free and open market. The group is concerned with standards selection and adoption, using International Standards where they exist.

Index

HEWLETT
PACKARD

HP Part Number
97089-90058

Microfiche No. 97089-99058
Printed in U.S.A. E0989

97089-90661
For Internal Use Only