AIX 5L Version 5.2

IBM

National Language Support Guide and Reference

SC23-4872-00

AIX 5L Version 5.2

National Language Support Guide and Reference

SC23-4872-00

Note

Before using this information and the product it supports, read the information in Appendix E, "Notices," on page 225.

First Edition (October 2002)

This edition applies to AIX 5L Version 5.2 and to all subsequent releases of this product until otherwise indicated in new editions.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to Information Development, Department H6DS-905-6C006, 11501 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

Contents

About This Book

This book provides information on how to provide national language support in a networked environment. Topics include locales, code sets, input methods, subroutines, and culture-specific information.

Who Should Use This Book

This book should be used by systems administrators who want to customize systems to provide national language support to administer their systems. Users should be familiar with C programming, basic system administration, and command line usage.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-Sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command is ″not found.″ Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

The following books contain information related to National Language Support:

- *Keyboard Technical Reference*
- *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*
- *The Unicode Standard* at http://unicode.org.

Chapter 1. National Language Support Overview

Many system variables are used to establish the language environment of the system. These variables and their supporting commands, files, and other tools, are referred to as National Language Support (NLS).

Topics covered in this chapter are:

- Chapter 2, "Locales," on page 7
- "Understanding Locale" on page 7
- "Understanding Locale Categories" on page 8
- "Understanding Locale Environment Variables" on page 9
- "Understanding the Locale Definition Source File" on page 11
- "Understanding the Character Set Description (charmap) Source File" on page 11
- "Converters Overview" on page 2.

NLS provides commands and Standard C Library subroutines for a single worldwide system base. An internationalized system has no built-in assumptions or dependencies on language-specific or cultural-specific conventions such as:

- Code sets
- Character classifications
- Character comparison rules
- Character collation order
- Numeric and monetary formatting
- Date and time formatting
- Message-text language.

All information pertaining to cultural conventions and language is obtained at process run time.

The following capabilities are provided by NLS to maintain a system running in an international environment:

- "Separation of Messages from Programs"
- "Conversion between Code Sets"

Separation of Messages from Programs

To facilitate translations of messages into various languages and to make the translated messages available to the program based on a user's locale, it is necessary to keep messages separate from the programs and provide them in the form of message catalogs that a program can access at run time. To aid in this task, commands and subroutines are provided by the message facility. For more information, see Chapter 7, "Message Facility," on page 151.

Conversion between Code Sets

A *character* is any symbol used for the organization, control, or representation of data. A group of such symbols used to describe a particular language make up a *character set*. A code set contains the encoding values for a character set. It is the encoding values in a code set that provide the interface between the system and its input and output devices.

Historically, the effort was directed at encoding the English alphabet. It was sufficient to use a 7-bit encoding method for this purpose because the number of English characters is not large. To support larger alphabets, such as the Asian languages, such as Chinese, Japanese, and Korean, additional code sets were developed that contained multibyte encodings.

A *character* is any symbol used for the organization, control, or representation of data. A group of such symbols for describing a particular language make up a character set. A code set contains the encoding values for a character set. The encoding values in a code set provide the interface between the system and its input and output devices.

An internationalized program must accurately read data generated in different code set environments and process the information accurately. You can use **nl_langinfo(CODESET)** to obtain the current code set in a process. The return value is a **char** pointer that is the name of the code set in the system. Because code set names are not standard, programs should not depend on any specific value for this string. Knowing the current code set can aid in code-set conversion. NLS supplies converters that translate character encoding values found in different code sets. For more information, see Chapter 5, “Converters Overview for Programming,” on page 83.

Input Method Support

The input of characters becomes complicated for languages having large character sets. For example, in Chinese, Korean, and Japanese, where the number of characters is large, it is not possible to provide one-to-one key mapping for a keystroke to a character. However, a special input method enables the user to enter phonetic or stroke characters and have them converted into native-language characters. A keyboard map associated with each keyboard matches sequences of one or more keystrokes with the appropriate character encoding. For more information, see Chapter 6, “Input Methods,” on page 123.

Converters Overview

National Language Support (NLS) provides a base for internationalization to allow data to be changed from one code set to another. You might need to convert text files or message catalogs. There are several standard converters for this purpose.

When a program sends data to another program residing on a remote host, the data can require conversion from the code set of the source machine to that of the receiver. For example, when communicating with an IBM VM system, the system converts its ISO8859-1 data to EBCDIC. Code sets define character and control function assignments to code points. These coded characters must be converted when a program receives data in one code set but displays it in another code set.

For more information on converters, see Chapter 5, “Converters Overview for Programming,” on page 83.

Using the Message Facility

To facilitate translation of messages into various languages and to make them available to a program based on a user’s locale, it is necessary to keep messages separate from the program and provide them in the form of message catalogs that a program can access at run time. To aid in this task, the Message Facility provides commands and subroutines. Message source files containing application messages are created by the programmer and converted to message catalogs. These catalogs are used by the application to retrieve and display messages, as needed. Message source files can be translated into other languages and converted to message catalogs without changing and recompiling a program.

The Message Facility includes the following commands for displaying messages with a shell script or from the command line:

dspcat Displays all or part of a message catalog

dspmsg Displays a selected message from a message catalog

These commands use the **NLSPATH** environment variable to locate the specified message catalog. The **NLSPATH** environment variable lists the directories containing message catalogs. These directories are searched in the order in which they are listed. For example:

```
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
```

The `%L` and `%N` special variables are defined as follows:

%L Specifies the locale-specific directory containing message catalogs. The value of the **LC_MESSAGES** category or the **LANG** environment variable is used for the directory name. The **LANG**, **LC_ALL**, or **LC_MESSAGES** environment variable can be set by the user to the locale for message catalogs.

%N Specifies the name of the catalog to be opened.

If the **dspcat** command cannot find the message, the default message is displayed. You must enclose the default message in single-quotation marks if the default message contains **%*n*$** format strings. If the **dspcat** command cannot find the message and you do not specify a default message, a system-generated error message is displayed.

The following example uses the **dspcat** command to display all messages in the existing `msgerrs.cat` message catalog:

```
/usr/lib/nls/msg/$LANG/msgerrs.cat:
dspcat msgerrs.cat
```

The following output is displayed:

```
1:1 Cannot open message catalog %s
Maximum number of catalogs already open
1:2 File %s not executable
2:1 Message %d, Set %d not found
```

By displaying the contents of the message catalog in this manner, you can find the message ID numbers assigned to the `msgerrs` message source file by the **mkcatdefs** command to replace the symbolic identifiers. Symbolic identifiers are not readily usable as references for the **dspmsg** command, but using the **dspcat** command as shown can give you the necessary ID numbers.

The following is a simple shell script called **runtest** that shows how to use the **dspmsg** command:

```
if [ - x ./test ]
    ./test;
else
    dspmsg  msgerrs.cat -s 1 2 '%s NOT EXECUTABLE \n' "test";
    exit;
```

Note: If you do not use a full path name, as in the preceding examples, be careful to set the **NLSPATH** environment variable so that the **dspcat** command searches the correct directory for the catalog. The **LC_MESSAGES** category or the value of the **LANG** environment variable also affects the directory search path.

Setting National Language Support for Devices

NLS uses the locale setting to define its environment. The locale setting is dependent on the user's requirements for data processing and language that determines input and output device requirements. The system administrator is responsible for configuring devices that are in agreement with user locales.

Terminals (tty Devices)

Use the **setmaps** command to set the terminal and code-set map for a given tty or pty. The **setmaps** file format defines the text of the code-set map file and the terminal map file.

The text of a code-set map file is a description of the code set, including the type (single byte or multibyte), the memory and screen widths (for multibyte code-sets), and the optional converter modules to push on the stream. The code set map file is located in the **/usr/lib/nls/csmap** directory and has the same name as the code set. For more information, see Converter Modules in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

The terminal-map-file rules associate a pattern string with a replacement string. The operating system uses an input map file to map input from the keyboard to an application and uses an output map file to map output from an application to the display.

Printers

Virtual printers inherit the default code set of incoming jobs from the **LANG** entry in the **/etc/environment** file. A printer subsystem can support several virtual printers. If more than one virtual printer is supported, each can have a different code set. The suggested printer subsystem scenarios areas follows:

- The first scenario involves several queues, several virtual printers, and one physical printer. Each virtual printer has its own code set. The print commands specify which queue to use. The queue in turn specifies the virtual printer with the appropriate code set. In this scenario, the user needs to know which queue is attached to which virtual printer and the code set that is associated with each.
- The second scenario is similar to the first, but each virtual printer is attached to a different printer.
- The third scenario involves using the **qprt** command to specify the code set. In this option, there are several queues available and one virtual printer. The virtual printer uses the inherited default code set.

Use the **qprt** command with the **-P-x** flags to specify the queue and code set. If the **-P** flag is not specified, the default queue is used. If the **-x** flag is not used, the default code set for the virtual printer is used.

Low-Function Terminals

Low-function terminals (LFTs) support single-byte code-set languages using key maps. An LFT key map translates a key stroke into a character string in the code set. A list of all available key maps is in the **/usr/lib/nls/loc** directory. LFT does not support languages that require multibyte code sets.

The default LFT keyboard setting and associated font setting are based on the language selected during installation. The possible default code sets are as follows:

- ISO8859-1
- ISO8859-2
- ISO8859-5
- ISO8859-6
- ISO8859-7
- ISO8859-8
- ISO8859-9
- ISO8859-15

You can change the default settings in the following ways:

- To change the default font for next reboot, use the **chfont** command with the **-n** flag.
- To change the default keyboard for next reboot, use the **chkbd** command with the **-n** flag.

The **lsfont** and **lskbd** commands list all the fonts and keyboard maps that are currently available to the LFT.

The LFT font libraries for all the supported code sets are in the **/usr/lpp/fonts** directory.

Changing the Language Environment

A number of system operations are affected by the language environment. Some of these operations include collation, time of day and date representation, numeric representation, monetary representation, and message translation. The language environment is determined by the value of the **LANG** environment variable, and you can change that value with the **chlang** command. The **chlang** command can be run from the command line or from SMIT.

To use the SMIT fast path to change the language environment, type `smit chlang` on the command line.

Changing the Default Keyboard Map

NLS also enables you to specify the correct keyboard for the language you want to use. The operating system provides a number of keyboard maps for this purpose. You can change the default keyboard map for LFT terminals using Web-based System Manager (type `wsm`, then select **Devices**), the SMIT fast path, **smit chkbd**, or the **chkbd** command. The change does not go into effect until you restart the system.

Note: Do not assume any particular physical keyboard is in use. Use an input method based on the locale setting to handle keyboard input.

Related Information

Chapter 2. Locales

An internationalized system has no built-in assumptions or dependencies on code set, character classification, character comparison rules, character collation order, monetary formatting, numeric punctuation, date and time formatting, or the text of messages. A *locale* is defined by these language and cultural conventions. An internationalized system processes information correctly for different locations. For example, in the United States, the date format, 9/6/2002, is interpreted to mean the sixth day of the ninth month of the year 2002. The United Kingdom interprets the same date format to mean the ninth day of the sixth month of the year 2002. The formatting of numeric and monetary data is also country-specific, for example, the U.S. dollar and the U.K. pound.

All locale information must be accessible to programs at run time so that data is processed and displayed correctly for your cultural conventions and language. This process is called localization. Localization consists of developing a database containing locale-specific rules for formatting data and an interface to obtain the rules.

Understanding Locale

A locale comprises the language, territory, and code set combination used to identify a set of language conventions. These conventions include information on collation, case conversion, and character classification, the language of message catalogs, date-and-time representation, the monetary symbol, and numeric representation.

Locale information contained in the locale definition source files must first be converted into a locale database by the **localedef** command. The **setlocale** subroutine can then access this information and set locale information for applications. To deal with locale data in a logical manner, locale definition source files are divided into six categories. Each category contains a specific aspect of the locale data. The **LC_*** environment variables and the **LANG** environment variable can be used to specify the desired locale. For more information on locale categories, see “Understanding Locale Categories” on page 8.

Typical User Scenarios

Users might encounter several NLS-related scenarios on the system. This section lists common scenarios with suggested actions to be taken.

- User keeps default code set

  The user might be satisfied with the default code set for language-territory combinations even where more than one code set is supported. The user might keep the default code set if the current user environment uses that code set, or if the user is new and has no code set preference.

  The language-territory selected at system installation time is defaulted to the appropriate locale based on the default code set. The default keyboard mappings, default font, and message catalogs are all established around the default code set. This scenario requires no special action from the user.

- User changes code set from the default code set

  Users of a Latin-1 or Japanese locale might want to migrate their data and NLS environment to a different (nondefault) code set. This can be done in the following fashion:

  – When the user has existing data that requires conversion

    Flat text files that require conversion to the preferred code set can be converted through the Users application in Web-based System Manager, the SMIT Manage the Language Environment menu, or the **iconv** utility. User-defined structured files require conversion through user-written conversion tools that use the **iconv** library functions to convert the desired text fields within the structured files.

  – When the user wants to change to the other code set

    Where more than one code set is supported for a language-territory combination, the user may change to a nondefault locale by using:

- The Users application in Web-based System Manager
- The SMIT Manage Language Environment menu
- The **chlang**, **chkbd**, and **chfont** commands.

Locale Naming Conventions

Each locale is named by its locale definition source file name. These files are named for the language, territory, and code set information they describe. The following format is used for naming a locale definition file:

```
language[_territory][.codeset][@modifier]
```

For example, the locale for the Danish language spoken in Denmark using the ISO8859-1 code set is `da_DK.ISO8859-1`. The `da` stands for the Danish language and the `DK` stands for Denmark. The short form of `da_DK` is sufficient to indicate this locale. The same language and territory using the ISO8859–15 code set is indicated by `da_DK.8859–15`.

System-defined locale definition files are provided to show the format of locale categories and their keywords. The **/usr/lib/nls/loc** directory contains the locale definition files for system-defined locales. The C, or POSIX, locale defines the ANSI C-defined standard locale inherited by all processes at startup time. To obtain a list of system-defined locale definition source files, enter the following on the command line:

```
/usr/lib/nls/lsmle -c
```

Installation Default Locale

The installation default locale refers to the locale selected at installation. For example, when prompted, a user can specify the French language as spoken in Canada during the installation process. The code set automatically defaults to the ISO8859-1 code set. With this information, the system sets the value of the default locale, specified by the **LANG** environment variable, to `fr_CA` (`fr` for ISO8859-1 French and `CA` for Canada). Every process uses this locale unless the **LC_*** or **LANG** environment variables are modified. The default locale can be changed by using the Manage Language Environment menu in SMIT. For more information see System Management Interface Tool (SMIT) Overview in *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices*.

The C or POSIX Locale

This locale refers to the ANSI C or POSIX-defined standard for the locale inherited by all processes at startup time. The C or POSIX locale assumes the 7-bit ASCII character set and defines information for the six previous categories.

Understanding Locale Categories

A *locale category* is a particular grouping of language-specific and cultural-convention-specific data. For instance, data referring to date-and-time formatting, the names of the days of the week, names of the months, and other time-specific information is grouped into the **LC_TIME** category. Each category uses a set of keywords that describe the particulars of that locale subset.

The following standard categories can be defined in a locale definition source file:

LC_COLLATE
: Defines character-collation or string-collation information.

LC_CTYPE
: Defines character classification, case conversion, and other character attributes.

LC_MESSAGES
: Defines the format for affirmative and negative responses.

LC_MONETARY
: Defines rules and symbols for formatting monetary numeric information.

LC_NUMERIC
Defines rules and symbols for formatting nonmonetary numeric information.

LC_TIME
Defines a list of rules and symbols for formatting time and date information.

Note: Locale categories can only be modified by editing the locale definition source file. Do not confuse them with the environment variables of the same name, which can be set from the command line.

Understanding Locale Environment Variables

National Language Support (NLS) uses several environment variables to influence the selection of locales. You can set the values of these variables to change search paths for locale information:

LANG Specifies the installation default locale.

Note: The **LANG** environment variable value is established at installation. (This is the locale every process uses unless the **LC_*** environment variables are set). The **LANG** environment variable can be changed by using the Manage Language Environment menu in SMIT. For more information about using SMIT, see System Management Interface Tool (SMIT) Overview in *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices*. The C and POSIX locales are designed to offer the best performance.

LC_ALL
Overrides the value of the **LANG** environment variable and the values of any other **LC_*** environment variables.

LC_COLLATE
Specifies the locale to use for **LC_COLLATE** category information. The **LC_COLLATE** category determines character-collation or string-collation rules governing the behavior of ranges, equivalence classes, and multicharacter collating elements.

LC_CTYPE
Specifies the locale to use for **LC_CTYPE** category information. The **LC_CTYPE** category determines character handling rules governing the interpretation of sequences of bytes of text data characters (that is, single-byte versus multibyte characters), the classification of characters (for example, alpha, digit, and so on), and the behavior of character classes.

LC__FASTMSG
Specifies that default messages are used for the C and POSIX locales and that **NLSPATH** are ignored when **LC__FASTMSG** is set to `true`. Otherwise, POSIX compliant message handling will be performed. The default value will be `LC__FASTMSG=true` in the **/etc/environment** file.

LC_MESSAGES
Specifies the locale to use for **LC_MESSAGES** category information. The **LC_MESSAGES** category determines rules governing affirmative and negative responses and the locale (language) for messages and menus.

LC_MONETARY
Specifies the locale to use for **LC_MONETARY** category information. The **LC_MONETARY** category determines the rules governing monetary-related formatting.

LC_NUMERIC
Specifies the locale to use for **LC_NUMERIC** category information. The **LC_NUMERIC** category determines the rules governing nonmonetary numeric formatting.

LC_TIME
Specifies the locale to use for **LC_TIME** category information. The **LC_TIME** category determines the rules governing date and time formatting.

LOCPATH
Specifies the search path for localized information, including binary locale files, input methods, and code-set converters.

Note: All **setuid** and **setgid** programs ignore the **LOCPATH** environment variable.

NLSPATH
Specifies the search path for locating message catalog files. This environment variable is used by the Message Facility component of the NLS subsystem. See the **catopen** subroutine for more information about expected format of the **NLSPATH** variable.

The environment variables that affect locale selection can be grouped into three priority classes: high, medium, and low. Environment variables in the high priority class are:

- **LC_ALL**
- **LC_COLLATE**
- **LC_CTYPE**

Environment variables in the medium priority class are:

- **LC_MESSAGES**
- **LC_MONETARY**
- **LC_NUMERIC**
- **LC_TIME**

The environment variable in the low priority class is:

- **LANG**

When a locale is requested by the **setlocale** subroutine for a particular category or for all categories, the environment variable settings are queried by their priority level in the following manner:

- If the **LC_ALL** environment variable is set, all six categories use the locale it specified. For example, if the **LC_ALL** environment variable is equal to `en_US` and the **LANG** environment variable is equal to `fr_FR`, a call to the **setlocale** subroutine sets each of the six categories to the `en_US` locale.
- If the **LC_ALL** environment variable is not set, each individual category uses the locale specified by its corresponding environment variable. For example, if the **LC_ALL** environment variable is not set, the **LC_COLLATE** environment variable is set to `de_DE`, and the **LC_TIME** environment variable is set to `fr_CA`, then a call to the **setlocale** subroutine sets the **LC_COLLATE** category to `de_DE` and the **LC_TIME** category to `fr_CA`. Neither environment variable has precedence over the other in this situation.
- If the **LC_ALL** environment variable is not set, and a value for a particular **LC_*** environment variable is not set, the value of the **LANG** environment variable determines the setting for that specific category. For example, if the **LC_ALL** environment variable is not set, the **LC_CTYPE** environment variable is set to `en_US`, the **LC_NUMERIC** environment variable is not set, and the **LANG** environment variable is set to `is_IS`, then a call to the **setlocale** subroutine sets the **LC_CTYPE** category to `en_US` and the **LC_NUMERIC** category to `is_IS`. The **LANG** environment variable specifies the locale for only those categories not previously determined by an **LC_*** environment variable.
- If the **LC_ALL** environment variable is not set, a value for a particular **LC_*** environment variable is not set, and the value of the **LANG** environment variable is not set, the locale for that specific category defaults to the `C` locale. For example, if the **LC_ALL** environment variable is not set, the **LC_MONETARY** environment variable is set to `sv_SE`, the **LC_TIME** environment variable is not set, and the **LANG** environment variable is not set, then a call to the **setlocale** subroutine sets the **LC_MONETARY** category to `sv_SE` and the **LC_TIME** category to `C`.

Environment Variables Precedence Example

The following table shows the current setting of the environment variables and the effect of calling **setlocale**(**LC_ALL**,″″). The last column indicates the locale setting after **setlocale**(**LC_ALL**,″″) is called.

Environment Variable and Category Names	Value of Environment Variables	Value of Category After Call To setlocale(LC_ALL,″″)
LC_COLLATE	de_DE	de_DE
LC_CTYPE	de_DE	de_DE
LC_MONETARY	en_US	en_US
LC_NUMERIC	(unset)	da_DK
LC_TIME	(unset)	da_DK
LC_MESSAGES	(unset)	da_DK
LC_ALL	(unset)	(not applicable)
LANG	da_DK	(not applicable)

Understanding the Character Set Description (charmap) Source File

Using the character set description (**charmap**) source file, you can assign symbolic names to character encodings.

Developers of character set description (**charmap**) source files can choose their own symbolic names, provided that these names do not conflict with the standardized symbolic names that describe the portable character set.

The **charmap** file resolves problems with the portability of sources, especially locale definition sources. The standardized portable character set is constant across all locales. The **charmap** file provides the capability to define a common locale definition for multiple code sets. That is, the same locale definition source can be used for code sets with different encodings of the same extended characters.

A **charmap** file defines a set of symbols that are used by the locale definition source file to refer to character encodings. The characters in the portable character set can optionally be included in the **charmap** file, but the encodings for these characters should not differ from their default encodings.

The **charmap** files are located in the **/usr/lib/nls/charmap** directory.

Understanding the Locale Definition Source File

Unlike environment variables, which can be set from the command line, locales can only be modified by editing and compiling a locale definition source file.

If a desired locale is not part of the library, a binary version of the locale can be compiled by the **localedef** command. Locale behavior of programs is not affected by a locale definition source file unless the file is first converted by the **localedef** command, and the locale object is made available to the program. The **localedef** command converts source files containing definitions of locales into a run-time format and copies the run-time version to the file specified on the command line, which usually is a locale name. Internationalized commands and subroutines can then access the locale information. For information on preparing source files to be converted by the **localedef** command, see Locale Definition Source File Format in *AIX 5L Version 5.2 Files Reference*.

Multibyte Subroutines

Multibyte subroutines process characters in file-code form. The names of these subroutines usually start with the prefix **mb**. However, some multibyte subroutines do not have this prefix. For example, the **strcoll** and **strxfrm** subroutines process characters in their multibyte form but do not have the **mb** prefix. The following standard C subroutines operate on bytes and can be used to handle multibyte data: **strcmp**, **strcpy**, **strncmp**, **strncpy**, **strcat**, and **strncat**. The standard C search subroutines **strchr**, **strrchr**, **strpbrk**, **strcspn**, **strrchr**, **strspn**, **strstr**, and **strtok** can be used in the following cases:

- Searching or scanning for characters in single-byte code sets
- Searching or scanning for unique code-point range characters in multibyte strings

For more information about multibyte character subroutines, see Chapter 3, “Subroutines for National Language Support,” on page 15.

Wide Character Subroutines

Wide character subroutines process characters in process-code form. Wide character subroutines usually start with a **wc** prefix. However, there are exceptions to this rule. For example, the wide character classification functions use an **isw** prefix. To determine if a subroutine is a wide character subroutine, check if the subroutine prototype defines characters as **wchar_t** data type or **wchar_t** data pointer, or else check whether the subroutine returns a **wchar_t** data type. There are some exceptions to this rule. For example, the wide character classification subroutines accept **wint_t** data type values.

For more information about wide character subroutines, see Chapter 3, “Subroutines for National Language Support,” on page 15.

Bidirectionality and Character Shaping

An internationalized program may need to handle bidirectionality of text and character shaping.

Bidirectionality (BIDI) occurs when texts of different direction orientation appear together. For example, English text is read from left to right. Hebrew text is read from right to left. If both English and Hebrew texts appear on the same line, the text is bidirectional.

Character shaping occurs when the shape of a character is dependent on its position in a line of text. In some languages, such as Arabic, characters have different shapes depending on their position in a string and on the surrounding characters.

For more information about bidirectionality and character shaping, see “Layout (Bidirectional Text and Character Shaping) Overview” on page 165.

Code Set Independence

The system needs certain information about code sets to communicate with the external environment. This information is hidden by the code set-independent library subroutines (NLS library). These subroutines pass information to the code set-dependent functions. Because NLS subroutines handle the necessary code set information, you do not need explicit knowledge of any code set when you write programs that process characters. This programming technique is called *code set independence*.

To see a sample program that illustrates internationalization through code-set independent programming, see Appendix C, “NLS Sample Program,” on page 217.

Determining Maximum Number of Bytes in Code Sets

You can use the **MB_CUR_MAX** macro to determine the maximum number of bytes in a multibyte character for the code set in the current locale. The value of this macro is dependent on the current setting of the **LC_CTYPE** category. Because the locale can differ between processes, running the **MB_CUR_MAX** macro in different processes or at different times may produce different results. The **MB_CUR_MAX** macro is defined in the **stdlib.h** header file.

You can use the **MB_LEN_MAX** macro to determine the maximum number of bytes in any code set that is supported by the system. This macro is defined in the **limits.h** header file.

Determining Character and String Display Widths

The **_max_disp_width** macro is operating-system-specific, and its use should be avoided in portable applications. If portability is not important, you can use the **_max_disp_width** macro to determine the maximum number of display columns required by a single character in the code set in the current locale. The value of this macro is dependent on the current setting of the **LC_CTYPE** category. If the value of this is 1 (one), all characters in the current code set require only one display column width on output.

When both **MB_CUR_MAX** and **_max_disp_width** are set to 1 (one), you can use the **strlen** subroutine to determine the display column width needed for a string. When **MB_CUR_MAX** is greater than one, use the **wcswidth** subroutine to find the display column width of the string.

The **wcswidth** and **wcwidth** wide-character display-width subroutines do not have corresponding multibyte functions. The **wcswidth** subroutine does not indicate how many characters can be displayed in the space available on a display. The **wcwidth** subroutine is useful for this purpose. This subroutine must be called repeatedly on a wide-character string to find out how many characters can be displayed in the available positions on the display.

Exceptions to Code Set Knowledge: Unique Code-Point Range

Because of the way the supported code sets are organized, there is one exception to the statement: ″No knowledge of the underlying code set can be assumed in a program.″

When a multibyte character string is searched for any character within the unique code-point range (for example, the . (period) character), it is not necessary to convert the string to process code form. It is sufficient to just look for that character (.) by examining each byte. This exception enables the kernel and utilities to search for the special characters . and / while parsing file names. If a program searches for any of the characters in the unique code-point range, the standard string functions that operate on bytes (such as the **strchr** subroutine), should be used. For a list of the characters in the unique code-point range, see “ASCII Characters” on page 53.

File Name Matching

POSIX.2 defines the **fnmatch** subroutine to be used for file name matching. An application can use the **fnmatch** subroutine to read a directory and apply a pattern against each entry. For example, the **find** utility can use the **fnmatch** subroutine. The **pax** utility can use the **fnmatch** subroutine to process its pattern operands. Applications that must match strings in a similar fashion can use the **fnmatch** subroutine.

Radix Character Handling

Note that the radix character, as obtained by **nl_langinfo(**RADIXCHAR**),** is a pointer to a string. It is possible that a locale may specify this as a multibyte character or as a string of characters. However, in AIX, a simplifying assumption is made that the RADIXCHAR is a single-byte character.

Programming Model

The programming model presented here highlights changes you need to make when an existing program is internationalized or when a new program is developed:

- Provide complete internationalization. Do not assume that characters have any specific properties. Determine the properties dynamically by using the appropriate interfaces. Do not assume properties of code sets, except for the ASCII characters with code points in the unique code-point range.
- Make programs code set-independent. Programs should not assume single-byte, double-byte, or multibyte encoding of any sort. Data can be processed in either process-code or file-code form by using the appropriate subroutines.
- Provide interaction with the kernel in file-code form only. The kernel does not handle process codes.
- The NLS subroutine library can handle processing based on file-code as well as processing based on process-code.

  Note: Several subroutines based on process-code form do not have corresponding subroutines based on file-code form. Due to this asymmetry, it may be necessary to convert strings to process-code form and invoke the appropriate process-code subroutines.
- Some libraries may not provide processing in process-code form. An application needing these libraries must use file-codes when invoking functions from them.
- Programs can process characters either in process-code form or file-code form. It is possible to write code set-independent programs using both methods.

Chapter 3. Subroutines for National Language Support

This chapter guides programmers in using subroutines when developing portable internationalized programs. Use standard Open Group, ISO/ANSI C, and POSIX functions to maximize portability.

The following topics are covered in this chapter:

Note: Do not use the layout subroutines in the **libi18n.a** library unless the application is doing presentation types of services. Most applications deal with logically ordered text.

Locale Subroutines

Programs that perform locale-dependent processing, including user messages, must call the **setlocale** subroutine at the beginning of the program. This call is the first executable statement in the **main** program. Programs that do not call the **setlocale** subroutine in this way inherit the C or POSIX locale. Such programs perform as in the C locale, regardless of the setting of the **LC_*** and **LANG** environment variables.

Other subroutines are provided to determine the current settings for locale data formatting. For more information about these subroutines, see “Setting the Locale.”

The locale of a process determines the way that character collation, character classification, date and time formatting, numeric punctuation, monetary punctuation, and message output are handled. The following section describes how to set and access information about the current locale in a program using National Language Support (NLS).

Setting the Locale

Every internationalized program must set the current locale using the **setlocale** subroutine. This subroutine allows a process to change or query the current locale by accessing locale databases.

When a process is started, its current locale is set to the C or POSIX locale. A program that depends on locale data not defined in the C or POSIX locale must invoke the **setlocale** subroutine in the following manner before using any of the locale-specific information:

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

Accessing Locale Information

The following subroutines provide access to information defined in the current locale as determined by the most recent call to the **setlocale** subroutine:

localeconv
: Provides access to locale information defined in the **LC_MONETARY** and **LC_NUMERIC** categories of the current locale. The **localeconv** subroutine retrieves information about these categories, places the information in a structure of type **lconv** as defined in the **locale.h** file, and returns a pointer to this structure.

nl_langinfo
: Returns a pointer to a null-terminated string containing information defined in the **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** categories of the current locale.

rpmatch
: Tests for positive and negative responses, which are specified in the **LC_MESSAGES** category of the current locale. Responses can be regular expressions, as well as simple strings. The **rpmatch** subroutine is not an industry-standard subroutine, portable applications should not assume that this subroutine is available.

The **localeconv** and **nl_langinfo** subroutines do not provide access to all **LC_*** categories.

The current locale setting for a category can be obtained by: **setlocale**(*Category*, `(char*)0`). The return value is a string specifying the current locale for *Category*. The following example determines the current locale setting for the **LC_CTYPE** category:

```
char *ctype_locale; ctype_locale = setlocale(LC_CTYPE, (char*)0);
```

Examples

- The following example uses the **setlocale** subroutine to change the locale from the default C locale to the locale specified by the environment variables, consistent with the hierarchy of the locale environment variables:

```
#include <locale.h>
main()
{

    char *p;

    p = setlocale(LC_ALL, "");

   /*
    **  The program will have the locale as set by the
    **  LC_* and LANG variables.
    */
 }
```

- The following example uses the **setlocale** subroutine to obtain the current locale setting for the **LC_COLLATE** category:

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

main()
{
    char *p;

    /*  set the current locale to what is specified  */
    p = setlocale(LC_ALL, "");
    /*  The current locale settings for all the
    **  categories is pointed to by p
    */

    /*
    **  Find the current setting for the
    **  LC_COLLATE category
    */
    p = setlocale(LC_COLLATE, NULL);
    /*
    **  p points to a string containing the current locale
    **  setting for the LC_COLLATE category.
    */

 }
```

- The following example uses the **setlocale** subroutine to obtain the current locale setting and saves it for later use. This action allows the program to temporarily change the locale to a new locale. After processing is complete, the locale can be returned to its original state.

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

#define NEW_LOCALE "MY_LOCALE"

main()
{
    char *p, *save_locale;

    p = setlocale(LC_ALL, "");
    /*
    **  Initiate locale. p points to the current locale
    **  setting for all the categories
    */

    save_locale = (char *)malloc(strlen(p) +1);
    strcpy(save_locale, p);
        /*  Save the current locale setting */
    p = setlocale(LC_ALL, NEW_LOCALE);
        /* Change to new locale */

    /*
    **  Do processing  ...
    */

     /*  Change back to old locale  */
    p = setlocale(LC_ALL, save_locale);  /* Restore old locale */

     free(save_locale);    /* Free the memory */
}
```

- The following example uses the **setlocale** subroutine to set the **LC_MESSAGES** category to the locale determined by the environment variables. All other categories remain set to the C locale.

```
#include <locale.h>

main()
{
    char *p;

    /*
    **  The program starts in the C locale for all categories.
    */

    p = setlocale(LC_MESSAGES, "");

     /*
    **  At this time the LC_COLLATE, LC_CTYPE, LC_NUMERIC,
    **  LC_MONETARY, LC_TIME will be in the C locale.
    **  LC_MESSAGES will be set to the current locale setting
    **  as determined by the environment variables.
    */
}
```

- The following example uses the **localeconv** subroutine to obtain the decimal-point setting for the current locale:

```
#include <locale.h>

main()
{
        struct lconv *ptr;
        char *decimal;

        (void)setlocale(LC_ALL, "");
        ptr = localeconv();
        /*
        ** Access the data obtained. For example,
```

```
        ** obtain the current decimal point setting.
        */
        decimal = ptr->decimal_point;
}
```

- The following example uses the **nl_langinfo** subroutine to obtain the date and time format for the current locale:

```
#include <langinfo.h>
#include <locale.h>
main()
{
    char *ptr;
    (void)setlocale(LC_ALL, "");
    ptr = nl_langinfo(D_T_FMT);
}
```

- The following example uses the **nl_langinfo** subroutine to obtain the radix character for the current locale:

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

main()
{
    char *ptr;
    (void)setlocale(LC_ALL, "");  /* Set the program's locale  */
    ptr = nl_langinfo(RADIXCHAR); /* Obtain the radix character*/
}
```

- The following example uses the **nl_langinfo** subroutine to obtain the setting of the currency symbol for the current locale:

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

main()
{
    char *ptr;
    (void)setlocale(LC_ALL, "");  /* Set the program's locale  */
    ptr = nl_langinfo(CRNCYSTR);  /* Obtain the currency string*/
    /* The currency string will be "-$" in the U. S. locale. */
}
```

- The following example uses the **rpmatch** subroutine to obtain the setting of affirmative and negative response strings for the current locale:

  The affirmative and negative responses as specified in the locale database are no longer simple strings; they can be regular expressions. For example, the yesexpr can be the following regular expression, which will accept an upper or lower case letter `y`, followed by zero or more alphabetic characters; or the character `O` followed by `K`. Thus, yesexpr may be the following regular expression:

  `([yY][:alpha:]*|OK)`

  The standards do not contain a subroutine to retrieve and compare this information. You can use the AIX-specific **rpmatch(const char *response)** subroutine.

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

int rpmatch(const char *);
    /*
    **  Returns 1 if yes response, 0 if no  response,
    ** -1 otherwise
    */

main()
{
    int  ret;
    char *resp;
```

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

    do {
        /*
        ** Obtain the response to the query for yes/no strings.
        ** The string pointer resp points to this response.
        ** Check if the string is yes.
        */
    ret = rpmatch(resp);

    if(ret == 1){
        /* Response was yes. */
        /* Process accordingly. */
        }else if(ret == 0){
        /* Response was negative. */
        /* Process negative response. */
        }else if(ret<0){
        /* No match with yes/no occurred. */
        continue;
        }
    }while(ret <0);
}
```

- The following example provides a method of implementing the **rpmatch** subroutine. Note that most applications should use the **rpmatch** subroutine in **libc**. The following implementation of the **rpmatch** subroutine is for illustration purposes only.

  Note that **nl_langinfo(YESEXPR)** and **nl_langinfo(NOEXPR)** are used to obtain the regular expressions for the affirmative and negative responses respectively.

```
#include <langinfo.h>
#include <regex.h>
/*
** rpmatch() performs comparison of a string to a regular expression
** using the POSIX.2 defined regular expression compile and match
** functions. The first argument is the response from the user and the
** second string is the current locale setting of the regular expression.
*/
int rpmatch( const char *string)


{
     int status;
     int retval;
     regex_t re;
     char *pattern;

     pattern = nl_langinfo(YESEXPR);
     /* Compile the regular expression pointed to by pattern. */
     if( ( status = regcomp( &re, pattern, REG_EXTENDED | REG_NOSUB )) != 0 ){
          retval = -2; /*-2 indicates yes expr compile error */
          return(retval);
     }
     /* Match the string with the compiled regular expression. */
     status = regexec( &re, string, (size_t)0, (regmatch_t *)NULL, 0);
     if(status == 0){
          retval = 1;   /* Yes match found */
     }else{    /* Check for negative response */
          pattern = nl_langinfo(NOEXPR);
          if( ( status = regcomp( &re, pattern,
               REG_EXTENDED | REG_NOSUB )) != 0 ){
                    retval = -3;/*-3 indicates no compile error */
                    return(retval);
               }
          status = regexec( &re, string, (size_t)0,
                         (regmatch_t *)NULL, 0);
          if(status == 0)
```

```
            retval = 0;/* Negative response match found */
    }else
            retval = -1; /* The string did not match yes or no
                           response */
    regfree(&re);
    return(retval);
}
```

Time Formatting Subroutines

Programs that need to format time into wide character code strings can use the **wcsftime** subroutine. Programs that need to convert multibyte strings into an internal time format can use the **strptime** subroutine.

In addition to the **strftime** subroutine defined in the C programming language standard, X/Open Portability Guide Issue 4 defines the following time formatting subroutines:

wcsftime
: Formats time into wide character code strings

strptime
: Converts a multibyte string into an internal time format

Examples

- The following example uses the **wcsftime** subroutine to format time into a wide character string:

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

main()
{
        wchar_t timebuf[BUFSIZE];
        time_t clock = time( (time_t*) NULL);
        struct tim *tmptr = localetime(&clock);

        (void)setlocale(LC_ALL, "");

        wcsftime(
                timebuf,       /* Time string output buffer */
                BUFSIZ,        /*Maximum size of output string */
                nl_langinfo(D_T_FMT),      /* Date/time format */
                tmptr          /* Pointer to tm structure */
        );

        printf("%S\n", timebuf);
}
```

- The following example uses the **strptime** subroutine to convert a formatted time string to internal format:

```
#include <langinfo.h>
#include <locale.h>
#include <time.h>

main(int argc, char **argv)
{
        struct tm tm;

        (void)setlocale(LC_ALL, "");

        if (argc != 2) {
                ...               /* Error handling */
        }
```

```
        if (strptime(
                argv[1],          /* Formatted time string */
                nl_langinfo(D_T_FMT),      /* Date/time format */
                &tm               /* Address of tm structure */
        ) == NULL) {
                ...               /* Error handling */
        }
        else {
                ...              /* Other Processing */
        }
}
```

Monetary Formatting Subroutines

Programs that need to specify or access monetary quantities can call the **strfmon** subroutine.

Although the C programming language standard in conjunction with POSIX provides a means of specifying and accessing monetary information, these standards do not define a subroutine that formats monetary quantities. The XPG4 **strfmon** subroutine provides the facilities to format monetary quantities. No defined subroutine converts a formatted monetary string into a numeric quantity suitable for arithmetic. Applications that need to do arithmetic on monetary quantities may do so after processing the locale-dependent monetary string into a number. The culture-specific monetary formatting information is specified by the **LC_MONETARY** category. An application can obtain information pertaining to the monetary format and the currency symbol by calling the **localeconv** subroutine.

Euro Currency Support

The **strfmon** subroutine uses the information from the locale's **LC_MONETARY** category to determine the correct monetary format for the given language/territory. Locales can handle both the traditional national currencies by using the @preeuro modifier, as well as the common European currency (euro). Each European country that uses the euro will have an additional **LC_MONETARY** definition with the @preeuro modifier appended. This alternate format is invoked when specified through the locale environment variables, or with the **setlocale** subroutine.

To use the French locale, UTF-8 codeset environment, and euro as the monetary unit, set:

```
LANG=FR_FR
```

To use the French locale, UTF-8 codeset environment, and French francs as the monetary unit, set:

```
LANG=FR_FR
LC_MONETARY=FR_FR@preeuro
```

Users should *not* attempt to set LANG=FR_FR@preeuro, because the @preeuro variant for locale categories other than **LC_MONETARY** is undefined.

Examples

- The following example uses the **strfmon** subroutine and accepts a format specification and an input value. The input value is formatted according to the input format specification.

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

main(int argc, char **argv)
{
        char bfr[256], format[256];
        int match; ssize_t size;
        float value;

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

```
        if (argc != 3){
                ...       /* Error handling */
        }
        match = sscanf(argv[1], "%f", &value);
        if (!match) {
                ...       /* Error handling */
        }
        match = sscanf(argv[2], "%s", format);
        if (!match) {
                ...       /*Error handling */
        }
        size = strfmon(bfr, 256, format, value);
        if (size == -1) {
                ...       /* Error handling */
        }
        printf ("Formatted monetary value is: %s\n", bfr);
}
```

The following table provides examples of other possible conversion specifications and the outputs for 12345.67 and -12345.67 in a U.S. English locale:

Conversion Specification	Output	Description
%n	$12,345.67 -$12,345.67	Default formatting
%15n	$12,345.67 -$12,345.67	Right justifies within a 15-character field.
%#6n	$ 12,345.67 -$ 12,345.67	Aligns columns for values up to 999,999.
%=*#8n	$****12,345.67 -$****12,345.67	Specifies a fill character.
%=0#8n	$000012,345.67 -$000012,345.67	Fill characters do not use grouping.
%^#6n	$ 12345.67 -$ 12345.67	Disables the thousands separator.
%^#6.0n	$ 12346 -$ 12346	Rounds off to whole units.
%^#6.3n	$ 12345.670 -$ 12345.670	Increases the precision.
%(#6n	$ 12,345.67 ($ 12,345.67)	Uses an alternate positive or negative style.
%!(#6n	12,345.67 (12,345.67)	Disables the currency symbol.

- The following example converts a monetary value into a numeric value. The monetary string is pointed to by `input`, and the result of converting it into numeric form is stored in the string pointed to by `output`. Assume that `input` and `output` are initialized.

```
char *input;   /* the input multibyte string containing the monetary string */
char *output;  /* the numeric string obtained from the input string */
wchar_t src_string[SIZE], dest_string[SIZE];
wchar_t *monetary, *numeric;
wchar_t mon_decimal_point, radixchar;
wchar_t wc;
localeconv *lc;

/* Initialize input and output to point to valid buffers as appropriate. */
/* Convert the input string to process code form*/
retval = mbstowcs(src_string, input, SIZE);
/* Handle error returns */

monetary = src_string;
numeric = dest_string;
lc = localeconv();
        /* obtain the LC_MONETARY and LC_NUMERIC info */

/* Convert the monetary decimal point to wide char form */
retval = mbtowc( &mon_decimal_point, lc->mon_decimal_point,
                MB_CUR_MAX);
```

```
/* Handle any error case */

/* Convert the numeric decimal point to wide char form */
retval = mbtowc( &radixchar, lc->decimal_point, MB_CUR_MAX);
/* Handle error case */
/* Assuming the string is converted first into wide character
** code form via mbstowcs, monetary points to this string.
*/
/* Pick up the numeric information from the wide character
** string and copy it into a temp buffer.
*/
        while(wc = *monetary++){
                if(iswdigit(wc))
                        *numeric++ = wc;
                else if( wc == mon_decimal_point)
                        *numeric++=radixchar;
        }
        *numeric = 0;
/* dest_string has the numeric value of the monetary quantity. */
/* Convert the numeric quantity into multibyte form */
retval = wcstombs( output, dest_string, SIZE);
/* Handle any error returns */
/* Output contains a numeric value suitable for atof conversion. */
```

Multibyte and Wide Character Subroutines

The external representation of data is referred to as the *file code* representation of a character. When file code data is created in files or transferred between a computer and its I/O devices, a single character may be represented by one or several bytes. For processing strings of such characters, it is more efficient to convert these codes into a uniform-length representation. This converted form is intended for internal processing of characters. The internal representation of data is referred to as the *process code* or *wide character code* representation of the character.

NLS internationalization of programs is a blend of multibyte and wide character subroutines. A *multibyte* subroutine uses multibyte character sets. A *wide character* subroutine uses wide character sets. Multibyte subroutines have an **mb** prefix. Wide character subroutines have a **wc** prefix. The corresponding string-handling subroutines are indicated by the **mbs** and **wcs** prefixes, respectively. Deciding when to use multibyte or wide character subroutines can be made only after careful analysis.

This section contains the following major subsections that discuss multibyte and wide character code subroutines:

- "Wide Character Classification Subroutines" on page 28
- "Multibyte and Wide Character String Collation Subroutines" on page 32
- "Multibyte and Wide Character String Comparison Subroutines" on page 34
- "Multibyte and Wide Character String Collation Subroutines" on page 32
- "Wide Character String Search Subroutines" on page 37
- "Working with the Wide Character Constant" on page 45

Multibyte Code and Wide Character Code Conversion Subroutines

The internationalized environment of NLS blends multibyte and wide character subroutines. The decision of when to use wide character or multibyte subroutines can be made only after careful analysis.

If a program primarily uses multibyte subroutines, it may be necessary to convert the multibyte character codes to wide character codes before certain wide character subroutines can be used. If a program uses wide character subroutines, data may need to be converted to multibyte form when invoking subroutines. Both methods have drawbacks, depending on the program in use and the availability of standard subroutines to perform the required processing. For instance, the wide character display-column-width subroutine has no corresponding standard multibyte subroutine.

If a program can process its characters in multibyte form, this method should be used instead of converting the characters to wide character form.

Attention: The conversion between multibyte and wide character code depends on the current locale setting. Do not exchange wide character codes between two processes, unless you have knowledge that each locale that might be used handles wide character codes in a consistent fashion. With the exception of locales based on the IBM-eucTW codeset, AIX locales use the Unicode character value as a wide character code.

Multibyte Code to Wide Character Code Conversion Subroutines

The following subroutines are used when converting from multibyte code to wide character code:

mblen
: Determines the length of a multibyte character. Do not use p++ to increment a pointer in a multibyte string. Use the **mblen** subroutine to determine the number of bytes that compose a character.

mbstowcs
: Converts a multibyte string to a wide character string.

mbtowc
: Converts a multibyte character to a wide character.

Wide Character Code to Multibyte Code Conversion Subroutines

The following subroutines are used when converting from wide character code to multibyte character code:

wcslen
: Determines the number of wide characters in a wide character string.

wcstombs
: Converts a wide character string to a multibyte character string.

wctomb
: Converts a wide character to a multibyte character.

Examples

- The following example uses the **mbtowc** subroutine to convert a character in multibyte character code to wide character code:

```
main()
{
    char     *s;
    wchar_t  wc;
    int      n;

    (void)setlocale(LC_ALL,"");

    /*
    **  s points to the character string that needs to be
    **  converted to a wide character to be stored in wc.
    */
    n = mbtowc(&wc, s, MB_CUR_MAX);

    if (n == -1){
        /*  Error handle  */
    }
    if (n == 0){
        /*  case of name pointing to null  */
    }

    /*
```

```
    **  wc contains the process code for the multibyte character
    **  pointed to by s.
    */
}
```

- The following example uses the **wctomb** subroutine to convert a character in wide character code to multibyte character code:

```
#include <stdlib.h>
#include <limits.h>              /* for MB_LEN_MAX  */
#include <stdlib.h>              /* for wchar_t     */

main()
{
    char    s[MB_LEN_MAX};       /*  system wide maximum number of
                                 **  bytes in a multibyte character r. */
    wchar_t wc;
    int     n;

    (void)setlocale(LC_ALL,"");

    /*
    **  wc is the wide character code to be converted to
    **  multibyte character code.
    */
    n = wctomb(s, wc);

    if(n == -1){
        /* pwcs does not point to a valid wide character */
    }
    /*
    ** n has the number of bytes contained in the multibyte
    ** character stored in s.
    */
}
```

- The following example uses the **mblen** subroutine to find the byte length of a character in multibyte character code:

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

main
{
    char *name = "h";
    int  n;

    (void)setlocale(LC_ALL,"");

    n = mblen(name, MB_CUR_MAX);
    /*
    **  The count returned in n is the multibyte length.
    **  It is always less than or equal to the value of
    **  MB_CUR_MAX in stdlib.h
    */
    if(n == -1){
        /* Error Handling */
    }
}
```

- The following example obtains a previous character position in a multibyte string. If you need to determine the previous character position, starting from a current character position (not a random byte position), step through the buffer starting at the beginning. Use the **mblen** subroutine until the current character position is reached, and save the previous character position to obtain the needed character position.

```
char buf[];     /* contains the multibyte string */
char *cur,      /* points to the current character position */
char *prev,     /* points to previous multibyte character */
```

```
char *p;        /* moving pointer */

/* initialize the buffer and pointers as needed */
/* loop through the buffer until the moving pointer reaches
** the current character position in the buffer, always
** saving the last character position in prev pointer */
p = prev = buf;

/* cur points to a valid character somewhere in buf */
while(p< cur){
        prev = p;
        if( (i=mblen(p, mbcurmax))<=0){
                /* invalid multibyte character or null */
                /* You can have a different error handling
                ** strategy */
                p++;    /* skip it */
        }else {
                p += i;
        }
}
/* prev will point to the previous character position */

/* Note that if( prev == cur), then it means that there was
** no previous character. Also, if all bytes up to the
** current character are invalid, it will treat them as
** all valid single-byte characters and this may not be what
** you want. One may change this to handle another method of
** error recovery. */
```

- The following example uses of the **mbstowcs** subroutine to convert a multibyte string to wide character string:

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

main()
{
     char    *s;
     wchar_t *pwcs;
     size_t  retval, n;

     (void)setlocale(LC_ALL, "");

     n = strlen(s) + 1;          /*string length + terminating null */

     /*  Allocate required wchar array    */
     pwcs = (wchar_t *)malloc(n * sizeof(wchar_t) );
     retval = mbstowcs(pwcs, s, n);
     if(retval == -1){

     /*  Error handle  */
          }
          /*
          ** pwcs contains the wide character string.
          */
}
```

- The following example illustrates the problems with using the **mbstowcs** subroutine on a large block of data for conversion to wide character form. When it encounters a multibyte that is not valid, the **mbstowcs** subroutine returns a value of -1 but does not specify where the error occurred. Therefore, the **mbtowc** subroutine must be used repeatedly to convert one character at a time to wide character code.

  Note: Processing in this manner can considerably slow program performance.
  During the conversion of single-byte code sets, there is no possibility for partial multibytes. However, during the conversion of multibyte code sets, partial multibytes are copied to a save buffer. During the next call to the **read** subroutine, the partial multibyte is prefixed to the rest of the byte sequence.

Note: A null-terminated wide character string is obtained. Optional error handling can be done if an instance of an invalid byte sequence is found.

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

main(int argc, char *argv[])
{

    char     *curp, *cure;
    int      bytesread, bytestoconvert, leftover;
    int      invalid_multibyte, mbcnt, wcnt;
    wchar_t  *pwcs;
    wchar_t  wbuf[BUFSIZ+1];
    char     buf[BUFSIZ+1];
    char     savebuf[MB_LEN_MAX];
    size_t   mb_cur_max;
    int      fd;
        /*
        **  MB_LEN_MAX specifies the system wide constant for
        **  the maximum number of bytes in a multibyte character.
        */

    (void)setlocale(LC_ALL, "");
    mb_cur_max = MB_CUR_MAX;

    fd = open(argv[1], 0);
    if(fd < 0){
       /*  error handle  */
    }

    leftover = 0;
    if(mb_cur_max==1){    /*  Single byte code sets case  */
        for(;;){
            bytesread = read(fd, buf, BUSIZ);
            if(bytesread <= 0)
                break;
            mbstowcs(wbuf, buf, bytesread+1);
            /*  Process using the wide character buffer   */
        }
                        /*  File processed ...   */
        exit(0);        /*  End of program       */

    }else{              /*  Multibyte code sets  */
        leftover = 0;

        for(;;) {
            if(leftover)
                strncpy(buf, savebuf ,leftover);
            bytesread=read(fd,buf+leftover, BUFSIZ-leftover);
            if(bytesread <= 0)
                break;

            buf[leftover+bytesread] = '\0';
                     /* Null terminate string */
            invalid_multibyte = 0;
            bytestoconvert = leftover+bytesread;
            cure= buf+bytestoconvert;
            leftover=0;
            pwcs = wbuf;
                /* Stop processing when invalid mbyte found. */
            curp= buf;

            for(;curp<cure;){
                mbcnt = mbtowc(pwcs,curp, mb_cur_max);
                if(mbcnt>0){
```

```
                    curp += mbcnt;
                    pwcs++;
                    continue;

                }else{
                    /* More data needed on next read*/
                    if ( cure-curp<mb_cur_max){
                        leftover=cure-curp;
                        strncpy(savebuf,curp,leftover);
                        /* Null terminate before partial mbyte */
                        *curp=0;
                        break;

                    }else{
                            /*Invalid multibyte found */
                        invalid_multibyte =1;
                        break;
                    }
                }
            }
            if(invalid_multibyte){          /*error handle */
            }
            /* Process the wide char buffer */
        }
    }
}
```

- The following example uses the **wcstombs** and **wcslen** subroutines to convert a wide character string to multibyte form:

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

main()
{
    wchar_t *pwcs; /* Source wide character string */
    char *s;       /* Destination multibyte character string */
    size_t n;
    size_t retval;

    (void)setlocale(LC_ALL, "");
    /*
    ** Calculate the maximum number of bytes needed to
    ** store the wide character buffer in multibyte form in the
    ** current code page and malloc() the appropriate storage,
    ** including the terminating null.
    */
    s = (char *) malloc( wcslen(pwcs) * MB_CUR_MAX + 1 );
    retval= wcstombs( s, pwcs, n);
    if( retval == -1) {
        /* Error handle */
        /* s points to the multibyte character string. */
}
```

Wide Character Classification Subroutines

The majority of wide character classification subroutines are similar to traditional character classification subroutines, except that wide character classification subroutines operate on a **wchar_t** data type argument passed as a **wint_t** data type argument.

Generic Wide Character Classification Subroutines

In the internationalized environment of National Language Support, you need the ability to create new character class properties. For example, several properties are defined for Japanese characters that are not applicable to the English language. As more languages are supported, a framework enabling applications to deal with a varying number of character properties is needed. The **wctype** and **iswctype** subroutines allow handling of character classes in a general fashion. These subroutines are used to allow for both user-defined and language-specific character classes.

The action of wide character classification subroutines is affected by the definitions in the **LC_CTYPE** category for the current locale.

To create new character classifications for use with the **wctype** and **iswctype** subroutines, create a new character class in the **LC_CTYPE** category and generate the locale using the **localedef** command. A user application obtains this locale data with the **setlocale** subroutine. The program can then access the new classification subroutines by using the **wctype** subroutine to get the **wctype_t** property handle. It then passes to the **iswctype** subroutine both the property handle and the wide character code of the character to be tested.

The following subroutines are used for wide character classification:

wctype
Obtains handle for character property classification.

iswctype
Tests for character property.

Standard Wide Character Classification Subroutines

The **isw*** subroutines determine various aspects of a standard wide character classification. The **isw*** subroutines also work with single-byte code sets. Use the **isw*** subroutines in preference to the **wctype** and **iswctype** subroutines. Use the **wctype** and **iswctype** subroutines only for extended character class properties (for example, Japanese language properties).

When using the wide character functions to convert the case in several blocks of data, the application must convert characters from multibyte to wide character code form. Because this can affect performance in single-byte code set locales, consider providing two conversion paths in your application. The traditional path for single-byte code set locales would convert case using the **isupper**,**islower**, **toupper**, and **tolower** subroutines. The alternate path for multibyte code set locales would convert multibyte characters to wide character code form and convert case using the **iswupper**, **iswlower**, **towupper** and **towlower** subroutines. When converting multibyte characters to wide character code form, an application needs to handle special cases where a multibyte character may split across successive blocks.

The following is a list of standard wide character classification subroutines:

iswalnum
Tests for alphanumeric character classification.

iswalpha
Tests for alphabetic character classification.

iswcntrl
Tests for control character classification.

iswdigit
Tests for digit character classification.

iswgraph
Tests for graphic character classification.

iswlower
Tests for lowercase character classification.

iswprint
Tests for printable character classification.

iswpunct
Tests for punctuation character classification.

iswspace
Tests for space character classification.

iswupper
Tests for uppercase character classification.

iswxdigit
Tests for hexadecimal-digit character classification.

Wide Character Case Conversion Subroutines

The following subroutines convert cases for wide characters. The action of wide character case conversion subroutines is affected by the definition in the **LC_CTYPE** category for the current locale.

towlower
Converts an uppercase wide character to a lowercase wide character.

towupper
Converts a lowercase wide character to an uppercase wide character.

Example

The following example uses the **wctype** subroutine to test for the **NEW_CLASS** character classification:

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

main()
{
    wint_t    wc;
    int       retval;
    wctype_t  chandle;

    (void)setlocale(LC_ALL,"");
    /*
    ** Obtain the character property handle for the NEW_CLASS
    ** property.
    */
    chandle = wctype("NEW_CLASS") ;
    if(chandle == (wctype_t)0){
        /* Invalid property. Error handle. */
    }
    /* Let wc be the wide character code for a character */
    /* Test if wc has the property of NEW_CLASS */
    retval = iswctype( wc, chandle );
    if( retval > 0 ) {
        /*
        ** wc has the property NEW_CLASS.
        */
    }else if(retval == 0) {
        /*
        ** The character represented by wc does not have the
        ** property NEW_CLASS.
        */
    }
}
```

Wide Character Display Column Width Subroutines

When characters are displayed or printed, the number of columns occupied by a character may differ. For example, a Kanji character (Japanese language) may occupy more than one column position. The number of display columns required by each character is part of the National Language Support locale database. The **LC_CTYPE** category defines the number of columns needed to display a character.

No standard multibyte display-column-width subroutines exist. For portability, convert multibyte codes to wide character codes and use the required wide character display-width subroutines. However, if the __max_disp_width** macro (defined in the **stdlib.h** file) is set to 1 and a single-byte code set is in use, then the display-column widths of all characters (except tabs) in the code set are the same, and are equal to 1. In this case, the **strlen (***string***)** subroutine gives the display column width of the specified string, as shown in the following example:

```
#include <stdlib.h>
        int display_column_width;  /* number  of  display  columns  */
        char  *s;                  /*  character  string            */
        ....
        if((MB_CUR_MAX  ==  1)  &&  (__max_disp_width  ==  1)){
                display_column_width  =  strlen(s);
                                   /*  s  is  a  string  pointer    */
        }
```

The following subroutines find the display widths for wide character strings:

wcswidth
Determines the display width of a wide character string.

wcwidth
Determines the display width of a wide character.

Examples

- The following example uses the **wcwidth** subroutine to find the display column width of a wide character:

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

main()
{
    wint_t  wc;
    int     retval;

    (void)setlocale(LC_ALL,  "");

    /*
    **    Let wc be the wide character whose display width is
    **    to be found.
    */
    retval  =  wcwidth(wc);
    if(retval  ==  -1){
        /*
        **  Error handling. Invalid or nonprintable
        **  wide character in wc.
        */
    }
}
```

- The following example uses the **wcswidth** subroutine to find the display column width of a wide character string:

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

main()
{
    wchar_t  *pwcs;
    int      retval;
    size_t   n;

    (void)setlocale(LC_ALL,  "");
    /*
    **    Let pwcs point to a wide character null
    **    terminated string.
```

```
        **    Let n be the number of wide characters
        **    whose display column width is to be determined.
        */
        retval  =  wcswidth(pwcs,  n);
        if(retval  ==  -1){
            /*
            **  Error handling. Invalid wide or nonprintable
            **  character  ode encountered in the wide
            **  character string pwcs.
            */
        }
    }
```

Multibyte and Wide Character String Collation Subroutines

Strings can be compared in the following ways:

- Using the ordinal (binary) values of the characters.
- Using the weights associated with the characters for each locale, as determined by the **LC_COLLATE** category.

National Language Support (NLS) uses the second method.

Collation is a locale-specific property of characters. A weight is assigned to each character to indicate its relative order for sorting. A character may be assigned more than one weight. Weights are prioritized as primary, secondary, tertiary, and so forth. The maximum number of weights assigned each character is system-defined.

A process inherits the C locale or POSIX locale at its startup time. When the **setlocale (LC_ALL, ″ ″)** subroutine is called, a process obtains its locale based on the **LC_*** and **LANG** environment variables. The following subroutines are affected by the **LC_COLLATE** category and determine how two strings will be sorted in any given locale.

Note: Collation-based string comparisons take a long time because of the processing involved in obtaining the collation values. Perform such comparisons only when necessary. If you need to determine whether two wide character strings are equal, do not use the **wcscoll** and **wcsxfrm** subroutines; use the **wcscmp** subroutine instead.

The following subroutines compare multibyte character strings:

strcoll
Compares the collation weights of multibyte character strings.

strxfrm
Converts a multibyte character string to values representing character collation weights.

The following subroutines compare wide character strings:

wcscoll
Compares the collation weights of wide character strings.

wcsxfrm
Converts a wide character string to values representing character collation weights.

Examples

- The following example uses the **wcscoll** subroutine to compare two wide character strings based on their collation weights:

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

```
extern  int  errno;

main()
{
    wchar_t  *pwcs1, *pwcs2;
    size_t   n;

    (void)setlocale(LC_ALL,  "");

    /*    set it to zero for checking errors on wcscoll    */
    errno  =  0;
    /*
    **    Let pwcs1 and pwcs2 be two wide character strings to
    **    compare.
    */
    n  =  wcscoll(pwcs1, pwcs2);
        /*
        **    If errno is set then it indicates some
        **    collation error.
        */
    if(errno  !=  0){
        /*  error has occurred... handle error ...*/
    }
}
```

- The following example uses the **wcsxfrm** subroutine to compare two wide character strings based on collation weights:

  Note: Determining the size *n* (where *n* is a number) of the transformed string, when using the **wcsxfrm** subroutine, can be accomplished in one of the following ways:

  – For each character in the wide character string, the number of bytes for possible collation values cannot exceed the **COLL_WEIGHTS_MAX * sizeof(wchar_t)** value. This value, multiplied by the number of wide character codes, gives the buffer length needed. To the buffer length add 1 for the terminating wide character null. This strategy may slow down performance.

  – Estimate the byte-length needed. If the previously obtained value is not enough, increase it. This may not satisfy all strings but gives maximum performance.

  – Call the **wcsxfrm** subroutine twice: first to find the value of *n*, and a second time to transform the string using this *n* value. This strategy slows down performance because the **wcsxfrm** subroutine is called twice. However, it yields a precise value for the buffer size needed to store the transformed string.

  The method you choose depends on the characteristics of the strings used in the program and the performance objectives of the program.

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

main()
{
    wchar_t  *pwcs1, *pwcs2, *pwcs3, *pwcs4;
    size_t   n, retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let the string pointed to by pwcs1 and pwcs3 be the
    **  wide character arrays to store the transformed wide
    **  character strings. Let the strings pointed to by pwcs2
    **  and pwcs4 be the wide character strings to compare based
    **  on the collation values of the wide characters in these
    **  strings.
    **  Let n be large enough (say,BUFSIZ) to transform the two
    **  wide character strings specified by pwcs2 and pwcs4.
```

```
    **
    **  Note:
    **  In practice, it is best to call wcsxfrm if the wide
    **  character string is to be compared several times to
    **  different wide character strings.
    */

    do {
        retval = wcsxfrm(pwcs1, pwcs2, n);
        if(retval == (size_t)-1){
            /*  error has occurred.  */
            /*  Process the error if needed  */
            break;
        }

        if(retval >= n ){
        /*
        ** Increase the value of n and use a bigger buffer pwcs1.
        */
        }
    }while (retval >= n);

    do {
        retval = wcsxfrm(pwcs3, pwcs4, n);
        if (retval == (size_t)-1){
            /*  error has occurred.  */
            /*  Process the error if needed  */
            break;

        if(retval >= n){
        /*Increase the value of n and use a bigger buffer pwcs3.*/
        }
    }while (retval >= n);
    retval = wcscmp(pwcs1, pwcs3);
    /*  retval has the result  */
}
```

Multibyte and Wide Character String Comparison Subroutines

The **strcmp** and **strncmp** subroutines determine if the contents of two multibyte strings are equivalent. If your application needs to know how the two strings differ lexically, use the multibyte and wide character string collation subroutines.

The following NLS subroutines compare wide character strings:

wcscmp Compares two wide character strings.
wcsncmp Compares a specific number of wide character strings.

Example

The following example uses the **wcscmp** subroutine to compare two wide character strings:

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

main()
{
    wchar_t *pwcs1, *pwcs2;
    int retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  pwcs1 and pwcs2 point to two wide character
    **  strings to compare.
    */
```

```
    retval = wcscmp(pwcs1, pwcs2);
    /*  pwcs1 contains a copy of the wide character string
    **  in pwcs2
    */
}
```

Wide Character String Conversion Subroutines

The following NLS subroutines convert wide character strings to double, long, and unsigned long integers:

wcstod Converts a wide character string to a double-precision floating point.
wcstol Converts a wide character string to a signed long integer.
wcstoul Converts a wide character string to an unsigned long integer.

Before calling the **wcstod**, **wcstoul**, or **wcstol** subroutine, the **errno** global variable must be set to 0. Any error that occurs as a result of calling these subroutines can then be handled correctly.

Examples

- The following example uses the **wcstod** subroutine to convert a wide character string to a double-precision floating point:

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

extern int errno;

main()
{
    wchar_t  *pwcs, *endptr;
    double   retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs point to a wide character null terminated
    **  string containing a floating point value.
    */
    errno = 0;   /*  set errno to  zero  */
    retval = wcstod(pwcs, &endptr);

    if(errno != 0){
        /*  errno has changed, so error has occurred */

        if(errno == ERANGE){
            /*  correct value is outside range of
            **  representable values. Case of overflow
            **  error
            */

             if((retval == HUGE_VAL) ||
                (retval == -HUGE_VAL)){
                /*  Error case. Handle accordingly.  */
            }else if(retval == 0){
                /*  correct value causes underflow   */
                /*  Handle appropriately             */
            }
        }
    }
    /*  retval contains the double.  */
}
```

- The following example uses the **wcstol** subroutine to convert a wide character string to a signed long integer:

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

extern int errno;

main()
{
    wchar_t   *pwcs, *endptr;
    long int  retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs point to a wide character null terminated
    **  string containing a signed long integer value.
    */
    errno = 0;  /*  set errno to  zero  */
    retval = wcstol(pwcs, &endptr, 0);

    if(errno != 0){
        /*  errno has changed, so error has occurred */

        if(errno == ERANGE){
            /*  correct value is outside range of
            **  representable values. Case of overflow
            **  error
            */

             if((retval == LONG_MAX) || (retval == LONG_MIN)){
                /*  Error case. Handle accordingly.     */
            }else if(errno == EINVAL){
                /*  The value of base is not supported  */
                /*  Handle appropriately                */
            }
        }
    }
    /*  retval contains the long integer.  */
}
```

- The following example uses the **wcstoul** subroutine to convert a wide character string to an unsigned long integer:

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

extern int errno;

main()
{
    wchar_t    *pwcs, *endptr;
    unsigned long int  retval;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs point to a wide character null terminated
    **  string containing an unsigned long integer value.
    */
    errno = 0;   /*  set errno to  zero  */
    retval = wcstoul(pwcs, &endptr, 0);

    if(errno != 0){
        /*  error has occurred */
        if(retval == ULONG_MAX || errno == ERANGE){
            /*
            **  Correct value is outside of
```

```
            **  representable value. Handle appropriately
            */
        }else if(errno == EINVAL){
            /*  The value of base is not representable  */
            /*  Handle appropriately                    */
        }
    }
    /*  retval contains the unsigned long integer.  */
}
```

Wide Character String Copy Subroutines

The following NLS subroutines copy wide character strings:

wcscpy Copies a wide character string to another wide character string.
wcsncpy Copies a specific number of characters from a wide character string to another wide character string.
wcscat Appends a wide character string to another wide character string.
wcsncat Appends a specific number of characters from a wide character string to another wide character string.

Example

The following example uses the **wcscpy** subroutine to copy a wide character string into a wide character array:

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

main()
{
    wchar_t *pwcs1, *pwcs2;
    size_t  n;

    (void)setlocale(LC_ALL, "");
    /*
    **  Allocate the required wide character array.
    */
    pwcs1 = (wchar_t *)malloc( (wcslen(pwcs2) +1)*sizeof(wchar_t));
    wcscpy(pwcs1, pwcs2);
    /*
    **  pwcs1 contains a copy of the wide character string in pwcs2
    */
}
```

Wide Character String Search Subroutines

The following NLS subroutines are used to search for wide character strings:

wcschr Searches for the first occurrence of a wide character in a wide character string.
wcsrchr Searches for the last occurrence of a wide character in a wide character string.
wcspbrk Searches for the first occurrence of a several wide characters in a wide character string.
wcsspn Determines the number of wide characters in the initial segment of a wide character string.
wcscspn Searches for a wide character string.
wcswcs Searches for the first occurrence of a wide character string within another wide character string.
wcstok Breaks a wide character string into a sequence of separate wide character strings.

Examples

- The following example uses the **wcschr** subroutine to locate the first occurrence of a wide character in a wide character string:

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

```
main()
{
    wchar_t *pwcs1, wc, *pws;
    int     retval;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let wc point to the wide character to search for.
    **
    */
    pws = wcschr(pwcs1, wc);
    if (pws == (wchar_t )NULL ){
        /*  wc does not occur in pwcs1  */
    }else{
        /*  pws points to the location where wc is found  */
    }
}
```

- The following example uses the **wcsrchr** subroutine to locate the last occurrence of a wide character in a wide character string:

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

main()
{
    wchar_t *pwcs1, wc, *pws;
    int     retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let wc point to the wide character to search for.
    **
    */
    pws = wcsrchr(pwcs1, wc);
    if (pws == (wchar_t )NULL ){
        /*  wc does not occur in pwcs1  */
    }else{
        /*  pws points to the location where wc is found  */
    }
}
```

- The following example uses the **wcspbrk** subroutine to locate the first occurrence of several wide characters in a wide character string:

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

main()
{
    wchar_t *pwcs1, *pwcs2, *pws;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
    **  that contains wide characters to search for.
    */
    pws = wcspbrk(pwcs1, pwcs2);

    if (pws == (wchar_t )NULL ){
        /* No wide character from pwcs2 is found in pwcs1    */
```

```
    }else{
        /* pws points to the location where a match is found */
    }
}
```

- The following example uses the **wcsspn** subroutine to determine the number of wide characters in the initial segment of a wide character string segment:

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

main()
{
    wchar_t *pwcs1, *pwcs2;
    size_t count;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
    **  that contains wide characters to search for.
    */
    count = wcsspn(pwcs1, pwcs2);
    /*
    **  count contains the length of the segment.
    */
 }
```

- The following example uses the **wcscspn** subroutine to determine the number of wide characters not in a wide character string segment:

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

main()
{
    wchar_t *pwcs1, *pwcs2;
    size_t count;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
    **  that contains wide characters to search for.
    */
    count = wcscspn(pwcs1, pwcs2);
    /*
    **  count contains the length of the segment consisting
    **  of characters not in pwcs2.
    */
}
```

- The following example uses the **wcswcs** subroutine to locate the first occurrence of a wide character string within another wide character string:

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

main()
{
    wchar_t *pwcs1, *pwcs2, *pws;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
```

```
    **  that contains wide characters sequence to locate.
    */
    pws = wcswcs(pwcs1, pwcs2);
    if (pws == (wchar_t)NULL){
       /*  wide character sequence pwcs2 is not found in pwcs1 */
    }else{
        /*
        **  pws points to the first occurrence of the sequence
        **  specified by pwcs2 in pwcs1.
        */
    }
}
```

- The following example uses the **wcstok** subroutine to tokenize a wide character string:

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

main()
{
    wchar_t *pwcs1 = L"?a???b,,,#c";
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");
    pwcs = wcstok(pwcs1, L"?");
    /*  pws points to the token:  L"a"  */
    pwcs = wcstok((wchar_t *)NULL, L",");
    /*  pws points to the token:  L"??b"  */
    pwcs = wcstok((wchar_t *)NULL, L"#,");
    /*  pws points to the token:  L"c"  */
}
```

Wide Character Input/Output Subroutines

NLS provides subroutines for both formatted and unformatted I/O.

Formatted Wide Character I/O

The **printf** and **scanf** subroutines allow for the formatting of wide characters. The **printf** and **scanf** subroutines have two additional format specifiers for wide character handling: %C and %S. The %C and %S format specifiers allow I/O on a wide character and a wide character string, respectively. They are similar to the %c and %s format specifiers, which allow I/O on a multibyte character and string.

The multibyte subroutines accept a multibyte array and output a multibyte array. To convert multibyte output from a multibyte subroutine to a wide character string, use the **mbstowcs** subroutine.

Unformatted Wide Character I/O

Unformatted wide character I/O subroutines are used when a program requires code set-independent I/O for characters from multibyte code sets. For example, use the **fgetwc** or **getwc** subroutine to input a multibyte character. If the program uses the **getc** subroutine to input a multibyte character, the program must call the **getc** subroutine once for each byte in the multibyte character.

Wide character input subroutines read multibyte characters from a stream and convert them to wide characters. The conversion is done as if the subroutines call the **mbtowc** and **mbstowcs** subroutines.

Wide character output subroutines convert wide characters to multibyte characters and write the result to the stream. The conversion is done as if the subroutines call the **wctomb** and **wcstombs** subroutines.

The **LC_CTYPE** category of the current locale affects the behavior of wide character I/O subroutines.

Reading and Processing an Entire File: If a program must go through an entire file that must be handled in wide character code form, use one of the following ways:

- In the case of multibyte characters, use either the **read** or **fread** subroutine to convert a block of text data into a buffer. Convert one character at a time in this buffer using the **mbtowc** subroutine. Handle special cases of multibyte characters crossing block boundaries. For multibyte code sets, do not use the **mbstowcs** subroutine on this buffer. On an invalid or a partial multibyte character sequence, the **mbstowcs** subroutine returns -1 without indicating how far it successfully converted the data. You can use the **mbstowcs** subroutine with single-byte code sets because you will not run into a partial-byte sequence problem with single-byte code sets.
- Use the **fgetws** subroutine to obtain a line from the file. If the returned wide character string contains a wide character <new-line>, then a complete line is obtained. If there is no <new-line> wide character, the line is longer than expected, and more calls to the **fgetws** subroutine are needed to obtain the complete line. If the program can efficiently process one line at a time, this approach is recommended.
- If the **fgets** subroutine is used to read a multibyte file to obtain one line at a time, a split multibyte character may result. Handle this condition just as in the case of the **read** subroutine breaking up a multibyte character across successive reads. If you can guarantee that the input line length is not more than a set limit, a buffer of that size (plus 1 for null) can be used, thereby avoiding the possibility of a split multibyte character. If the program can efficiently process one line at a time, this approach may be used. Because of the possibility of split bytes in the buffer, use the **fgetws** subroutine in preference to the **fgets** subroutine for multibyte characters.
- Use the **fgetwc** subroutine on the file to read one wide character code at a time. If a file is large, the function call overhead becomes large and reduces the value of this method.

The decision of which of these methods to use should be made on a per program basis. The **fgetsw** subroutine option is recommended, as it is capable of optimum performance and the program does not have to handle the special cases.

Input Subroutines: The **wint_t** data type is required to represent the wide character code value as well as the end-of-file (EOF) marker. For example, consider the case of the **fgetwc** subroutine, which returns a wide character code value:

wchar_t fgetwc();
: If the **wchar_t** data type is defined as a **char** value, the y-umlaut symbol cannot be distinguished from the end-of-file (EOF) marker in the ISO8859-1 code set. The 0xFF code point is a valid character (y umlaut). Hence, the return value cannot be the **wchar_t** data type. A data type is needed that can hold both the EOF marker and all the code points in a code set.

int fgetwc();
: On some machines, the **int** data type is defined to be 16 bits. When the **wchar_t** data type is larger than 16 bits, the **int** value cannot represent all the return values.

The **wint_t** data type is therefore needed to represent the **fgetwc** subroutine return value. The **wint_t** data type is defined in the **wchar.h** file.

The following subroutines are used for wide character input:

fgetwc Gets next wide character from a stream.
fgetws Gets a string of wide characters from a stream.
getwc Gets next wide character from a stream.
getwchar Gets next wide character from standard input.
getws Gets a string of wide characters from a standard input.
ungetwc Pushes a wide character onto a stream.

Output Subroutines: The following subroutines are used for wide character output:

fputwc Writes a wide character to an output stream.
fputws Writes a wide character string to an output stream.
putwc Writes a wide character to an output stream.
putwchar Writes a wide character to standard output.

putws Writes a wide character string to standard output.

Examples

- The following example uses the **fgetwc** subroutine to read wide character codes from a file:

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

main()
{
    wint_t  retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    /*
    **  Open a stream.
    */
    fp = fopen("file", "r");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */
    }else{
        /*
        **  pwcs points to a wide character buffer of BUFSIZ.
        */
        while((retval = fgetwc(fp)) != WEOF){
            *pwcs++ = (wchar_t)retval;
               /*  break when buffer is full  */
        }
    }
    /*  Process the wide characters in the buffer  */
}
```

- The following example uses the **getwchar** subroutine to read wide characters from standard input:

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

 main()
{
    wint_t  retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    index = 0;
    while((retval = getwchar()) != WEOF){
        /*  pwcs points to a wide character buffer of BUFSIZ.  */
        *pwcs++ = (wchar_t)retval;
        /*  break on buffer full  */
    }
    /*  Process the wide characters in the buffer  */
}
```

- The following example uses the **ungetwc** subroutine to push a wide character onto an input stream:

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

```
main()
{
    wint_t  retval;
    FILE    *fp;

    (void)setlocale(LC_ALL, "");
    /*
    **  Open a stream.
    */
    fp = fopen("file", "r");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */

    else{
        retval = fgetwc(fp);
        if(retval != WEOF){
            /*
            **  Peek at the character and return it to the stream.
            */
            retval = ungetwc(retval, fp);
            if(retval == EOF){
                /*  Error on ungetwc  */
            }
        }
    }
}
```

- The following example uses the **fgetws** subroutine to read a file, one line at a time:

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

main()
{
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    /*
    **  Open a stream.
    */
    fp = fopen("file", "r");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */
    }else{
        /*  pwcs points to wide character buffer of BUFSIZ.  */
        while(fgetws(pwcs, BUFSIZ, fp) != (wchar_t *)NULL){
            /*
            **  pwcs contains wide characters with null
            **  termination.
            */
        }
    }
}
```

- The following example uses the **fputwc** subroutine to write wide characters to an output stream:

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

main()
{
    int     index, len;
    wint_t  retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    /*
    **  Open a stream.
    */
    fp = fopen("file", "w");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */
    }else{
        /*  Let len indicate number of wide chars to output.
        **  pwcs points to a wide character buffer of BUFSIZ.
        */
        for(index=0; index < len; index++){
            retval = fputwc(*pwcs++, fp);
            if(retval == WEOF)
                break;  /*  write error occurred  */
                        /*  errno is set to indicate the error. */
        }
    }
}
```

- The following example uses the **fputws** subroutine to write a wide character string to a file:

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

main()
{
    int     retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    /*
    **  Open a stream.
    */
    fp = fopen("file", "w");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */

    }else{
        /*
        **  pwcs points to a wide character string
        **  to output to fp.
        */
        retval = fputws(pwcs, fp);
        if(retval == -1){
```

```
            /*  Write error occurred                */
            /*  errno is set to indicate the error  */
        }
    }
}
```

Working with the Wide Character Constant

Use the **L** constant for ASCII characters only. For ASCII characters, the **L** constant value is numerically the same as the code point value of the character. For example, `L'a'` is same as `a`. The **L** constant obtains the **wchar_t** value of an ASCII character for assignment purposes. A wide character constant is introduced by the **L** specifier. For example:

```
wchar_t wc = L'x' ;
```

A wide character code corresponding to the character `x` is stored in `wc`. The C compiler converts the character `x` using the **mbtowc** or **mbstowcs** subroutine as appropriate. This conversion to wide characters is based on the current locale setting at compile time. Because ASCII characters are part of all supported code sets and the wide character representation of all ASCII characters is the same in all locales, `L'x'` results in the same value across all code sets. However, if the character `x` is non-ASCII, the program may not work when it is run on a different code set than used at compile time. This limitation impacts some programs that use switch statements using the wide character constant representation.

wchar.h Header File

The **wchar.h** header file declares information that is necessary for programming with multibyte and wide character subroutines. The **wchar.h** header file declares the **wchar_t**, **wctype_t**, and **wint_t** data types, as well as several functions for testing wide characters. Because the number of characters implemented as wide characters exceeds that of basic characters, it is not possible to classify all wide characters into the existing classes used for basic characters. Therefore, it is necessary to provide a way of defining additional classes specific to some locale. The action of these subroutines is affected by the current locale.

The **wchar.h** header file also declares subroutines for manipulating wide character strings (that is, **wchar_t** data type arrays). Array length is always determined in terms of the number of **wchar_t** elements in an array. A null wide character code ends an array. A pointer to a **wchar_t** data type array or void array always points to the initial element of the array.

> **Note:** If the number of **wchar_t** elements in an array exceeds the defined array length, unpredictable results can occur.

Internationalized Regular Expression Subroutines

Programs that contain internationalized regular expressions can use the **regcomp**, **regexec**, **regerror**, **regfree**, and **fnmatch** subroutines.

The following subroutines are available for use with internationalized regular expressions.

regcomp
: Compiles a specified basic or extended regular expression into an executable string.

regexec
: Compares a null-terminated string with a compiled basic or extended regular expression that must have been previously compiled by a call to the **regcomp** subroutine.

regerror
: Provides a mapping from error codes returned by the **regcomp** and **regexec** subroutines to printable strings.

regfree
: Frees any memory allocated by the **regcomp** subroutine associated with the compiled basic or

extended regular expression. The expression is no longer treated as a compiled basic or extended regular expression after it is given to the **regfree** subroutine.

fnmatch
Checks a specified string to see if it matches a specified pattern. You can use the **fnmatch** subroutine in an application that reads a dictionary to find which entries match a given pattern. You also can use the **fnmatch** subroutine to match path names to patterns.

Examples

- The following example compiles an internationalized regular expression and matches a string using this compiled expression. A match is found for the first pattern, but no match is found for the second pattern.

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

#define                 BUFSIZE    256


main()
{
        char    *p;

        char    *pattern[] = {
                "hello[0-9]*",
                "1234"
        };


        char     *string = "this is a test string hello112 and this is test";
                /* This is the source string for matching */

        int     retval;
        regex_t   re;
        char    buf[BUFSIZE];

        int     i;

        setlocale(LC_ALL, "");

        for(i = 0;i <2; i++){
                retval = match(string, pattern[i], &re);
                if(retval == 0){
                        printf("Match found \n");
                }else{
                        regerror(retval, &re, buf, BUFSIZE);
                        printf("error = %s\n", buf);
                }
        }
        regfree( &re);
}


int match(char *string, char *pattern, regex_t *re)
{
        int     status;

        if((status=regcomp( re, pattern, REG_EXTENDED))!= 0)
                return(status);
        status = regexec( re, string, 0, NULL, 0);
        return(status);
}
```

- The following example finds all substrings in a line that match a pattern. The numbers `11` and `2001` are matched. Every digit that is matched counts as one match. There are six such matches corresponding to the six digits supplied in the string.

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

#define                BUFSIZE   256


main()
{
        char    *p;

        char    *pattern = "[0-9]";
        char    *string = "Today is 11 Feb 2001 ";

        int     retval;
        regex_t re;
        char    buf[BUFSIZE];
        regmatch_t pmatch[100];
        int      status;
        char     *ps;

        int      eflag;

        setlocale(LC_ALL, "");

        /* Compile the pattern */

        if((status = regcomp( &re, pattern, REG_EXTENDED))!= 0){
                regerror(status, &re, buf, 120);
                exit(2);
        }

        ps = string;
        printf("String to match=%s\n", ps);
        eflag = 0;

        /* extract all the matches */
        while( status = regexec( &re, ps, 1, pmatch, eflag)== 0){
                printf("match found at: %d, string=%s\n",
                        pmatch[0].rm_so, ps +pmatch[0].rm_so);
                ps += pmatch[0].rm_eo;
                printf("\nNEXTString to match=%s\n", ps);
                eflag = REG_NOTBOL;

        }
        regfree( &re);
}
```

- The following example uses the **fnmatch** subroutine to read a directory and match file names with a pattern.

```
#include                 <locale.h>
#include                 <fnmatch.h>
#include                 <sys/dir.h>

main(int argc, char *argv[] )
{
        char    *pattern;
        DIR     *dir;
        struct dirent    *entry;
        int     ret;


                setlocale(LC_ALL, "");

        dir = opendir(".");

        pattern = argv[1];
```

```
        if(dir != NULL){
                while( (entry = readdir(dir)) != NULL){
                        ret = fnmatch(pattern, entry->d_name,
                                FNM_PATHNAME|FNM_PERIOD);
                        if(ret == 0){
                                printf("%s\n", entry->d_name);
                        }else if(ret == FNM_NOMATCH){
                                continue ;
                        }else{
                                printf("error file=%s\n",
                                        entry->d_name);
                        }
                }
                closedir(dir);
        }
}
```

Related Information

Chapter 3, “Subroutines for National Language Support,” on page 15 provides information about wide character and multibyte subroutines.

For more information about using locales, see Chapter 2, “Locales,” on page 7.

Character Set Description (charmap) source file format, Locale Definition source file format.

For specific information about locale categories and their keywords, see the **LC_COLLATE** category, **LC_CTYPE** category, **LC_MESSAGES** category, **LC_MONETARY** category, **LC_NUMERIC** category, and **LC_TIME** category.

Chapter 3, “Subroutines for National Language Support,” on page 15 provides information about wide character and multibyte subroutines.

The **strfmon** subroutine.

Chapter 3, “Subroutines for National Language Support,” on page 15 provides information about wide character and multibyte subroutines.

The **LC_COLLATE** category of the locale definition file in *AIX 5L Version 5.2 Files Reference*.

The **LC_CTYPE** category of the locale definition file in *AIX 5L Version 5.2 Files Reference*.

The **localedef** command in *AIX 5L Version 5.2 Commands Reference, Volume 3*

“List of Wide Character Subroutines” on page 175 and “List of Multibyte Character Subroutines” on page 175

The **getc** subroutine, **printf** subroutines, in *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1*; and **read** subroutine, **scanf** subroutines, **setlocale** subroutine, **strlen** subroutine in *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2*.

Chapter 4. Code Sets for National Language Support

The internationalization of AIX is based on the assumption that all code sets can be divided into any number of character sets.

The following topics are covered in this section:

To understand code sets, it is necessary to first understand *character sets*. A character set is a collection of predefined characters based on the specific needs of one or more languages without regard to the encoding values used to represent the characters. The choice of which code set to use depends on the user's data processing requirements. A particular character set can be encoded using different encoding schemes. For example, the ASCII character set defines the set of characters found in the English language. The Japanese Industrial Standard (JIS) character set defines the set of characters used in the Japanese language. Both the English and Japanese character sets can be encoded using different code sets.

The ISO2022 standard defines a *coded character set* as a set of precise rules that defines a character set and the one-to-one relationship between each character and its bit pattern. A *code set* defines the bit patterns that the system uses to identify characters.

A *code page* is similar to a code set with the limitation that a code-page specification is based on a 16-column by 16-row matrix. The intersection of each column and row defines a coded character.

Consider the following when working with code sets:

- Do not assume the size of all characters to be 8 bits, or 1 byte. Characters may be 1, 2, 3, 4 or more bytes.
- Do not assume the encoding of any code set.
- Do not hard code names of code sets, locales, or fonts because it can impact portability.

The following code sets are supported:

- Support for industry-standard code sets is provided. The **ISO8859** family of code sets provides a range of single-byte code set support that includes:
  - Latin-1
  - Latin-2
  - Cyrillic
  - Arabic
  - Greek
  - Hebrew
  - Turkish

  The following industry-standard code sets are available:
  - The IBM-eucJP code set is the industry-standard code set used to support the Japanese locale.
  - The IBM-eucKR code set is the industry-standard code set used to support Korean countries.
  - The IBM-eucTW code set is the industry-standard code set used to support Traditional Chinese countries.

– The IBM-eucCN code set is the industry-standard code set used to support countries using Simplified Chinese.
– The UTF-8 code set is a Universal Transformation Format of Unicode/ISO10646 used to support multiple languages at once (including Simplified Chinese, Traditional Chinese, and Chinese characters used in Japanese and Korean).

- **ISO8859-15** standard codeset is a replacement standard for the existing ISO8859-1 codeset that is currently in use by the western European locales, the United States, and Canada. The need for another codeset resulted from the introduction of the euro currency unit and the need for European countries to be able to do business transactions using the euro. In addition, ISO8859-15 contains 7 additional characters for the French and Finnish languages.
- Support is also provided for the personal computer (PC) based code sets **IBM-856**, **IBM-943**, **IBM-932**, and **IBM-1046**. IBM-856 is a single-byte code set used to support Hebrew countries. IBM-943 and IBM-932 are multibyte code set used to support the Japanese locale. IBM-1046 is a single-byte code set used to support Arabic countries.
- **IBM-1129** is a single-byte code set used to support Vietnamese.
- **TIS-620** is a single-byte code set used to support Thai.
- **IBM-1124** is a single-byte code set used to support Ukrainian.
- Full Unicode support is provided by the **UTF-8** code set for *all* languages and territories supported by AIX. The UTF-8 code set is a Universal Transformation Format of Unicode/ISO10646 used to support multiple languages at once. The UTF-8 code set provides the most complete solution for use in environments where multiple languages and alphabets must be processed. The Unicode/UTF-8 codeset also provides full support for the common European currency (euro).
- **IBM-1252** codeset support is provided as a compatibility option for users who require a single byte codeset environment containing the euro currency symbol. The structure of the IBM-1252 codeset is identical to the industry-standard codeset ISO8859-1, except that additional graphic characters are added in the ISO control character range from 0x80 through 0x9F. The euro currency symbol is located at hexadecimal value 0x80 in the IBM-1252 codeset.

Single-Byte and Multibyte Code Sets

A single-byte encoding method is sufficient for representing the English character set because the number of characters is not large. To support larger alphabets, such as Japanese and Chinese, additional code sets containing multibyte encodings are necessary. All supported single-byte and multibyte code sets contain the single-byte ASCII character set. Therefore, programs that handle multibyte code sets must handle character encodings of one or more bytes.

An example of a single-byte code set is the ISO 8859 family of code sets. Examples of multibyte character sets are the IBM-eucJP and the IBM-943 code sets. The single-byte code sets have at most 256 characters and the multibyte code sets have more than 256 (without any theoretical limit).

Unique Code-Point Range

None of the supported code sets have bytes 0x00 through 0x3F in any byte of a multibyte character. This group of code points is called the *unique code-point range*. Furthermore, these code points always refer to the same characters as specified for 7-bit ASCII. This is a special property governing all supported code sets. ASCII Characters in the Unique Code-Point Range (“ASCII Characters” on page 53) lists the characters in the unique code-point range.

Data Representation

Because the encoding for some characters requires more than one byte, a single character may be represented by one or several bytes when data is created in files or transferred between a computer and its I/O devices. This external representation of data is referred to as the *file code* or *multibyte character code* representation of a character.

For processing strings of such characters, it is more efficient to convert file codes into a uniform representation. This converted form is intended for internal processing of characters. This internal representation of data is referred to as the *process code* or *wide character code* representation of the character. An understanding of multibyte character and wide character codes is essential to the overall internationalization strategy.

Multibyte Character Code Data Representation

A multibyte character code is an external representation of data, regardless of whether it is character input from a keyboard or a file on a disk. Within the same code set, the number of bytes that represent the multibyte code of a character can vary. You must use NLS functions for character processing to ensure code set independence.

For example, a code set may specify the following character encodings:

```
C  = 0x43
*  = 0x81 0x43
*C = 0x81 0x43& 0x43
```

A program searching for `C`, not accounting for multibyte characters, finds the second byte of the `*C` string and assumes it found `C` when in fact it found the second byte of the `*` (asterisk) character.

Wide Character Code Data Representation

The wide character code was developed so that multibyte characters could be processed more efficiently internally in the system. A multibyte character representation is converted into a uniform internal representation (wide character code) so that internally all characters have the same length. Using this internal form, character processing can be done in a code set-independent fashion. The wide character code refers to this internal representation of characters.

The **wchar_t** data type is used to represent the wide character code of a character. The size of the **wchar_t** data type is implementation-specific. It is a **typedef** definition and can be found in the **ctype.h**, **stddef.h**, and **stdlib.h** files. No program should assume a particular size for the **wchar_t** data type, enabling programs to run under implementations that use different sizes for the **wchar_t** data type.

On AIX 4.3, the **wchar_t** datatype is implemented as an unsigned short value (16 bits). On AIX 5.1 and later, the **wchar_t** datatype is 32–bit in the 64–bit environment and 16–bit in the 32–bit environment. . The locale methods have been standardized such that in most locales, the value stored in the **wchar_t** for a particular character will always be its Unicode data value. For applications which are intended to run only on AIX, this allows certain applications handle the **wchar_t** datatype in a consistent fashion, even if the underlying codeset is unknown. All locales use Unicode for their wide character code values (process code), except the IBM-eucTW codeset. The IBM-eucTW codeset (LANG =**zh_TW**) contains many characters that are not contained in the Unicode standard. Because of this, it is impossible to represent these characters with a Unicode wide character value. Applications that need to have Unicode based **wchar_t** data for Traditional Chinese should use the **Zh_TW** locale (big5 codeset) instead.

Do not assume that the **char** data type is either signed or unsigned. This is platform-specific. If the particular system that is used defines **char** to be **signed**, comparisons with full 8-bit quantity will yield incorrect results. As all the 8-bits are used in encoding a character, be sure to declare **char** as **unsigned char** wherever necessary. Also, note that if a **signed char** value is used to index an array, it may yield incorrect results. To make programs portable, define 8-bit characters as **unsigned char**.

Character Properties

Every character has several language-dependent attributes or properties. These properties are called *class properties*. For example, the lowercase letter a in U.S. English has the following properties:

- alphabetic
- hexadecimal digit
- printable
- lowercase
- graphic

Character class properties are specified by the **LC_CTYPE** category.

Collation-Order Properties

Character ordering or *collation* refers to the culture-specific ordering of characters. This ordering differs from that based on the ordinal value of a character in a code set. Collation-based ordering is dependent on the language. Character collation is specified by the **LC_COLLATE** category. The term *collating element* refers to one or more characters that have a collation value in a specific locale. The Spanish ll character is an example of a multicharacter collating element.

To sort the characters in any given language in the proper order, a weight is assigned to each character so that the characters sort as expected. However, a character's sort value and code-point value are not necessarily related.

One set of weights is not sufficient to sort strings for all languages. For example, in the case of the German words b<*a-umlaut*>ch and bane, if there is only one set of weights, and the weight of the letter a is less than that of <*a-umlaut*>, then bane sorts before b<*a-umlaut*>ch. However, the opposite result is correct. To satisfy the requirement of this example, two sets of weights, the Primary and Secondary Weights, are given to each character in the language. In the case of the characters a and <*a-umlaut*>, they have the same Primary Weights, but differ in their Secondary Weights. In the German locale, the Secondary Weight of a is less than that of <*a-umlaut*>.

The sorting algorithm first compares the two strings based on the Primary Weights of each character. If the Primary Weight values are the same, the two strings are compared again based on their Secondary Weights. In this example, the Primary Weights of the first two characters ba and b<*a-umlaut*> are the same, but the Primary Weights of the characters that follow (c and n, respectively) differ. As a result of this comparison, b<*a-umlaut*>ch is sorted before bane.

Here, the Secondary Weights are not used to collate the strings. However, as in the case of the strings bach and b<*a-umlaut*>ch, Secondary Weights must be used to get the proper order. When compared using Primary Weight values, these two strings are found to be equivalent. To break the tie, the Secondary Weights of a and <*a-umlaut*> are used. Because the Secondary Weight of a is less than that of <*a-umlaut*>, the string bach sorts before b<*a-umlaut*>ch.

Characters having the same Primary Weights belong to the same *equivalence class*. In this example, the characters a and <*a-umlaut*> are said to be members of the same equivalence class.

In string collation, each pair of strings is first compared based on Primary Weight. If the two strings are equal, they are compared again based on their Secondary Weights. If still equal, they are compared again based on Tertiary Weights up to the limit set by the **COLL_WEIGHTS_MAX** collating weight limit specified in the **sys/limits.h** file.

Code-Set Width

Code-set width refers to the maximum number of bytes required to represent a character as a file code. This information is specified by the **LC_CTYPE** category.

Code-Set Display Width

Code-set display width refers to the maximum number of columns required to display a character on a terminal. This information is specified by the **LC_CTYPE** category.

ASCII Characters

ASCII is a code set containing 128 code points (0x00 through 0x7F). The ASCII character set contains control characters, punctuation marks, digits, and the uppercase and lowercase English alphabet. Several 8-bit code sets incorporate ASCII as a proper subset. However, throughout this document, ASCII refers to 7-bit-only code sets. To emphasize this, it is referred to as 7-bit ASCII. The 7-bit ASCII code set is a proper subset of all supported code sets and is referred to as the *portable character set*. For more information, see Chapter 4, "Code Sets for National Language Support," on page 49.

ASCII Characters in the Unique Code-Point Range

The following table lists the ASCII characters in the unique code-point range. These characters are in the range 0x00 through 0x3F.

ASCII Characters in the Unique Code-Point Range					
Symbolic Name	**Hex Value**	**Glyph**	**Symbolic Name**	**Hex Value**	**Glyph**
nul	00		space	20	blank
soh	01		exclamation-mark	21	!
stx	02		quotation-mark	22	"
etx	03		number-sign	23	#
eot	04		dollar-sign	24	$
enq	05		percent	25	%
ack	06		ampersand	26	&
alert	07		apostrophe	27	'
backspace	08		left-parenthesis	28	(
tab	09		right-parenthesis	29	)
newline	0A		asterisk	2A	*
vertical-tab	0B		plus-sign	2B	+
form-feed	0C		comma	2C	,
carriage-return	0D		hyphen	2D	-
so	0E		period	2E	.
si	0F		slash	2F	/
dle	10		zero	30	0
dc1	11		one	31	1
dc2	12		two	32	2
dc3	13		three	33	3
dc4	14		four	34	4
nak	15		five	35	5
syn	16		six	36	6
etb	17		seven	37	7
can	18		eight	38	8
em	19		nine	39	9

ASCII Characters in the Unique Code-Point Range					
Symbolic Name	**Hex Value**	**Glyph**	**Symbolic Name**	**Hex Value**	**Glyph**
sub	1A		colon	3A	:
esc	1B		semicolon	3B	;
is1	1C		less-than	3C	<
is2	1D		equal-sign	3D	=
is3	1E		greater-than	3E	>
is4	1F		question-mark	3F	?

Other ASCII Characters

The following table lists the 7-bit ASCII characters that are not in the unique code-point range. These characters are in the range 0x40 through 0x7F.

Other ASCII Characters					
Symbolic Name	**Hex Value**	**Glyph**	**Symbolic Name**	**Hex Value**	**Glyph**
commercial-at	40	@	grave-accent	60	`
A	41	A	a	61	a
B	42	B	b	62	b
C	43	C	c	63	c
D	44	D	d	64	d
E	45	E	e	65	e
F	46	F	f	66	f
G	47	G	g	67	g
H	48	H	h	68	h
I	49	I	i	69	i
J	4A	J	j	6A	j
K	4B	K	k	6B	k
L	4C	L	l	6C	l
M	4D	M	m	6D	m
N	4E	N	n	6E	n
O	4F	O	o	6F	o
P	50	P	p	70	p
Q	51	Q	q	71	q
R	52	R	r	72	r
S	53	S	s	73	s
T	54	T	t	74	t
U	55	U	u	75	u
V	56	V	v	76	v
W	57	W	w	77	w
X	58	X	x	78	x
Y	59	Y	y	79	y
Z	5A	Z	z	7A	z

Other ASCII Characters					
Symbolic Name	**Hex Value**	**Glyph**	**Symbolic Name**	**Hex Value**	**Glyph**
left-bracket	5B	[	left-brace	7B	{
backslash	5C	\	vertical-line	7C	|
right-bracket	5D	]	right-brace	7D	}
circumflex	5E	^	tilde	7E	~
underscore	5F	_	del	7F	

Code Set Strategy

Each locale in the system defines which code set it uses and how the characters within the code set are manipulated. Because multiple locales can be installed on the system, multiple code sets can be used by different users on the system. While the system can be configured with locales using different code sets, all system utilities assume that the system is running under a single code set.

Most commands have no knowledge of the underlying code set being used by the locale. The knowledge of code sets is hidden by the code set-independent library subroutines (NLS library), which pass information to the code set-dependent subroutines.

Because many programs rely on ASCII, all code sets include the 7-bit ASCII code set as a proper subset. Because the 7-bit ASCII code set is common to all supported code sets, its characters are sometimes referred to as the *portable character set*. The 7-bit ASCII code set is based on the ISO646 definition and contains the control characters, punctuation characters, digits (0-9), and the English alphabet in uppercase and lowercase.

Code Set Structure

Each code set is divided into the following principal areas:

Graphic Left (GL) Columns 0-7
Graphic Right (GR) Columns 8-F

The first two columns of each code set are reserved by International Organization for Standardization (ISO) standards for control characters. The terms C0 and C1 are used to denote the control characters for the Graphic Left and Graphic Right areas, respectively.

Note: The IBM PC code sets use the C1 control area to encode graphic characters.

The remaining six columns are used to encode graphic characters. Graphic characters are considered to be printable characters, while the control characters are used by devices and applications to indicate some special function.

Control Characters

Based on the ISO definition, a control character initiates, modifies, or stops a control operation. A control character is not a graphic character, but can have graphic representation in some instances. The control characters in the following table are present in all supported code sets and the encoded values of the control characters are consistent throughout the code sets.

Name	Value	Description
NUL	00	Null

Name	Value	Description
SOH	01	Start of header
STX	02	Start of text
ETX	03	End of text
EOT	04	End of transmission
ENQ	05	Enquiry
ACK	06	Acknowledge
BEL	07	Bell
BS	08	Backspace
HT	09	Horizontal tab
LF	0A	Line feed
VT	0B	Vertical tab
FF	0C	Form feed
CR	0D	Carrier return
SO	0E	Shift Out
SI	0F	Shift In
DLE	10	Data link escape
DC1	11	Device control 1
DC2	12	Device control 2
DC3	13	Device control 3
DC4	14	Device control 4
NAK	15	Not acknowledge
SYN	16	Synchrous idle
ETB	17	End of transmission block
CAN	18	Cancel
EM	1	End of media
SUB	1A	Substitute character
ESC	1B	Escape character
IS4	1C	Info Separator Four
IS3	1D	Info Separator Three
IS2	1E	Info Separator Two
IS1	1F	Info Separator One

Graphic Characters

Each code set can be considered to be divided into one or more character sets, with each character having a unique coded value. The ISO standard reserves six columns for encoding characters and does not allow graphic characters to be encoded in the control character columns.

Single-Byte and Multibyte Code Sets

Code sets that use all 8 bits of a byte can support European, Middle Eastern, and other alphabetic languages. Such code sets are called single-byte code sets. Single-byte code sets have a limit of encoding 191 characters, not including control characters.

Languages that require more than 191 characters use a mixture of single-byte characters (8 bits) and multibyte characters (more than 8 bits). The system can support any number of bits to encode a character.

ISO Code Sets

The code sets listed in the following topics are based on definitions set by the International Organization for Standardization (ISO).

ISO646-IRV

The ″ISO646-IRV code set″ below defines the code set used for information processing based on a 7-bit encoding. The character set associated with this code set is derived from the ASCII characters.

First Hexadecimal Digit

Second Hexadecimal Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	BLANK (SPACE)	0	@	P	`	p								
1	SOH	DC1	!	1	A	Q	a	q								
2	STX	DC2	"	2	B	R	b	r								
3	ETX	DC3	#	3	C	S	c	s								
4	EOT	DC4	$	4	D	T	d	t								
5	ENQ	NAK	%	5	E	U	e	u								
6	ACK	SYN	&	6	F	V	f	v								
7	BEL	ETB	'	7	G	W	g	w								
8	BS	CAN	(	8	H	X	h	x								
9	HT	EM	)	9	I	Y	i	y								
A	LF	SUB	*	:	J	Z	j	z								
B	VT	ESC	+	;	K	[	k	{								
C	FF	IS4	,	<	L	\	l	|								
D	CR	IS3	—	=	M	]	m	}								
E	SO	IS2	.	>	N	^	n	~								
F	S1	IS1	/	?	O	_	o	△								

ISO8859 Family

ISO8859 is a family of single-byte encodings based on and compatible with other ISO, American National Standards Institute (ANSI), and European Computer Manufacturer's Association (ECMA) code extension techniques. The ISO8859 encoding defines a family of code sets with each member containing its own unique character sets. The 7-bit ASCII code set is a proper subset of each of the code sets in the ISO8859 family.

While the ASCII code set defines an order for the English alphabet, the Graphic Right (GR) characters are not ordered according to any specific language. The locale defines the language-specific ordering.

Each code set includes the ASCII character set plus its own unique character set. The ISO8859 encoding figure shows the ISO8859 general encoding scheme.

Character Encoding	Code Point	Description	Count
000xxxxx	00–1F	Controls	32
00100000	20	Space	1
0xxxxxxx	21–7E	7-bit	94
01111111	7F	Delete	1
100xxxxx	80–9F	Controls	32
10100000	A0	No-break Space	1
1xxxxxxx	A1–F	8-bit	96

Code Set ISO8859-1

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-1. For a textual representation of this code set, see “ISO8859–1” on page 179.

First Hexadecimal Digit

Second Hexadecimal Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	‘	p			NBSP	°	À	Đ	à	ð
1			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
2			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
3			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
4			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
6			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
7			'	7	G	W	g	w			§	·	Ç	×	ç	÷
8			(	8	H	X	h	x			¨	¸	È	Ø	è	ø
9			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
A			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
B			+	;	K	[	k	{			«	»	Ë	Û	ë	û
C			,	<	L	\	l	|			¬	1/4	Ì	Ü	ì	ü
D			–	=	M	]	m	}			SHY	1/2	Í	Ý	í	ý
E			.	>	N	^	n	~			®	3/4	Î	Þ	î	þ
F			/	?	O	_	o				—	¿	Ï	β	ï	ÿ

Code Set ISO8859-2

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-2. For a textual representation of this code set, see “ISO8859–2” on page 182.

First Hexadecimal Digit

Second Hexadecimal Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p			RSP	°	Ŕ	Đ	ŕ	đ
1			!	1	A	Q	a	q			Ą	ą	Á	Ń	á	ń
2			"	2	B	R	b	r			˘	˛	Â	Ň	â	ň
3			#	3	C	S	c	s			Ł	ł	Ă	Ó	ă	ó
4			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5			%	5	E	U	e	u			Ľ	ľ	Ĺ	Ő	ĺ	ő
6			&	6	F	V	f	v			Ś	ś	Ć	Ö	ć	ö
7			'	7	G	W	g	w			§	ˇ	Ç	×	ç	÷
8			(	8	H	X	h	x			¨	¸	Č	Ř	č	ř
9			)	9	I	Y	i	y			Š	š	É	Ů	é	ů
A			*	:	J	Z	j	z			Ş	ş	Ę	Ú	ę	ú
B			+	;	K	[	k	{			Ť	ť	Ë	Ű	ë	ű
C			,	<	L	\	l	|			Ź	ź	Ě	Ü	ě	ü
D			-	=	M	]	m	}			SHY	˝	Í	Ý	í	ý
E			.	>	N	^	n	~			Ž	ž	Î	Ţ	î	ţ
F			/	?	O	_	o				Ż	ż	Ď	ß	ď	˙

Code Set ISO8859-5

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-5. For a textual representation of this code set, see “ISO8859–5” on page 184.

First Hexadecimal Digit (columns); Second Hexadecimal Digit (rows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p			RSP	А	Р	а	р	№
1			!	1	A	Q	a	q			Ё	Б	С	б	с	ё
2			"	2	B	R	b	r			Ђ	В	Т	в	т	ђ
3			#	3	C	S	c	s			Ѓ	Г	У	г	у	ѓ
4			$	4	D	T	d	t			Є	Д	Ф	д	ф	є
5			%	5	E	U	e	u			Ѕ	Е	Х	е	х	ѕ
6			&	6	F	V	f	v			І	Ж	Ц	ж	ц	і
7			'	7	G	W	g	w			Ї	З	Ч	з	ч	ї
8			(	8	H	X	h	x			Ј	И	Ш	и	ш	ј
9			)	9	I	Y	i	y			Љ	Й	Ш	й	ш	љ
A			*	:	J	Z	j	z			Њ	К	Ъ	к	ъ	њ
B			+	;	K	[	k	{			Ћ	Л	Ы	л	ы	ћ
C			,	<	L	\	l					Ќ	М	Ь	м	ь	ќ
D			-	=	M	]	m	}			SHY	Н	Э	н	э	§
E			.	>	N	^	n	~			Ў	О	Ю	о	ю	ў
F			/	?	O	_	o				Џ	П	Я	п	я	џ

Code Set ISO8859-6

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-6. For a textual representation of this code set, see “ISO8859–6” on page 187.

First Hexadecimal Digit / Second Hexadecimal Digit

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	‘	p			RSP			ذ	ـ	ِ
1			!	1	A	Q	a	q					ء	ر	ف	ّ
2			"	2	B	R	b	r					آ	ز	ق	ْ
3			#	3	C	S	c	s					أ	س	ك	
4			$	4	D	T	d	t			¤		ؤ	ش	ل	
5			%	5	E	U	e	u					إ	ص	م	
6			&	6	F	V	f	v					ئ	ض	ن	
7			'	7	G	W	g	w					ا	ط	ه	
8			(	8	H	X	h	x					ب	ظ	و	
9			)	9	I	Y	i	y					ة	ع	ى	
A			*	:	J	Z	j	z					ت	غ	ي	
B			+	;	K	[	k	{				؛	ث		ً	
C			,	<	L	\	l	|			،		ج		ٌ	
D			-	=	M	]	m	}			SHY		ح		ٍ	
E			.	>	N	^	n	~					خ		َ	
F			/	?	O	_	o					؟	د		ُ	

Code Set ISO8859-7

The following figure summarizes the available symbols and layout of Code Set ISO8859-7. This code set is made up of an ASCII character set plus its own unique character set. For a textual representation of this code set, see “ISO8859–7” on page 188.

First Hexadecimal Digit (columns); Second Hexadecimal Digit (rows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p			NBSP	°	ΐ	Π	ΰ	π
1			!	1	A	Q	a	q			‘	±	Α	Ρ	α	ρ
2			"	2	B	R	b	r			’	²	Β		β	φ
3			#	3	C	S	c	s			£	³	Γ	Σ	γ	σ
4			$	4	D	T	d	t				΄	Δ	Τ	δ	τ
5			%	5	E	U	e	u				΅	Ε	Υ	ε	ϑ
6			&	6	F	V	f	v			¦	Ά	Ζ	Φ	ζ	ϕ
7			'	7	G	W	g	w			§	·	Η	Χ	η	χ
8			(	8	H	X	h	x			¨	Έ	Θ	Ψ	θ	ψ
9			)	9	I	Y	i	y			©	Ή	Ι	Ω	ι	ω
A			*	:	J	Z	j	z				Ί	Κ	Ϊ	κ	ϊ
B			+	;	K	[	k	{			«	»	Λ	Ϋ	λ	ϋ
C			,	<	L	\	l	|			¬	Ό	Μ	ά	μ	ό
D			-	=	M	]	m	}			SHY	1/2	Ν	έ	ν	ύ
E			.	>	N	^	n	~				Ύ	Ξ	ή	ξ	ώ
F			/	?	O	_	o				―	Ώ	Ο	ί	ο	

Code Set ISO8859-8

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-8. For a textual representation of this code set, see “ISO8859–8” on page 190.

Second Hexadecimal Digit / First Hexadecimal Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p			RSP	°			א	נ
1			!	1	A	Q	a	q				±			ב	ס
2			"	2	B	R	b	r			¢	²			ג	ע
3			#	3	C	S	c	s			£	³			ד	ף
4			$	4	D	T	d	t			¤	´			ה	פ
5			%	5	E	U	e	u			¥	µ			ו	ץ
6			&	6	F	V	f	v			¦	¶			ז	צ
7			'	7	G	W	g	w			§	●			ח	ק
8			(	8	H	X	h	x			¨	¸			ט	ר
9			)	9	I	Y	i	y			©	¹			י	ש
A			*	:	J	Z	j	z			×	÷			ך	ת
B			+	;	K	[	k	{			«	»			כ	
C			,	<	L	\	l	|			¬	¼			ל	
D			-	=	M	]	m	}			SHY	½			ם	
E			.	>	N	^	n	~			®	¾			מ	
F			/	?	O	_	o				¯			‗	ן	

Code Set ISO8859-9

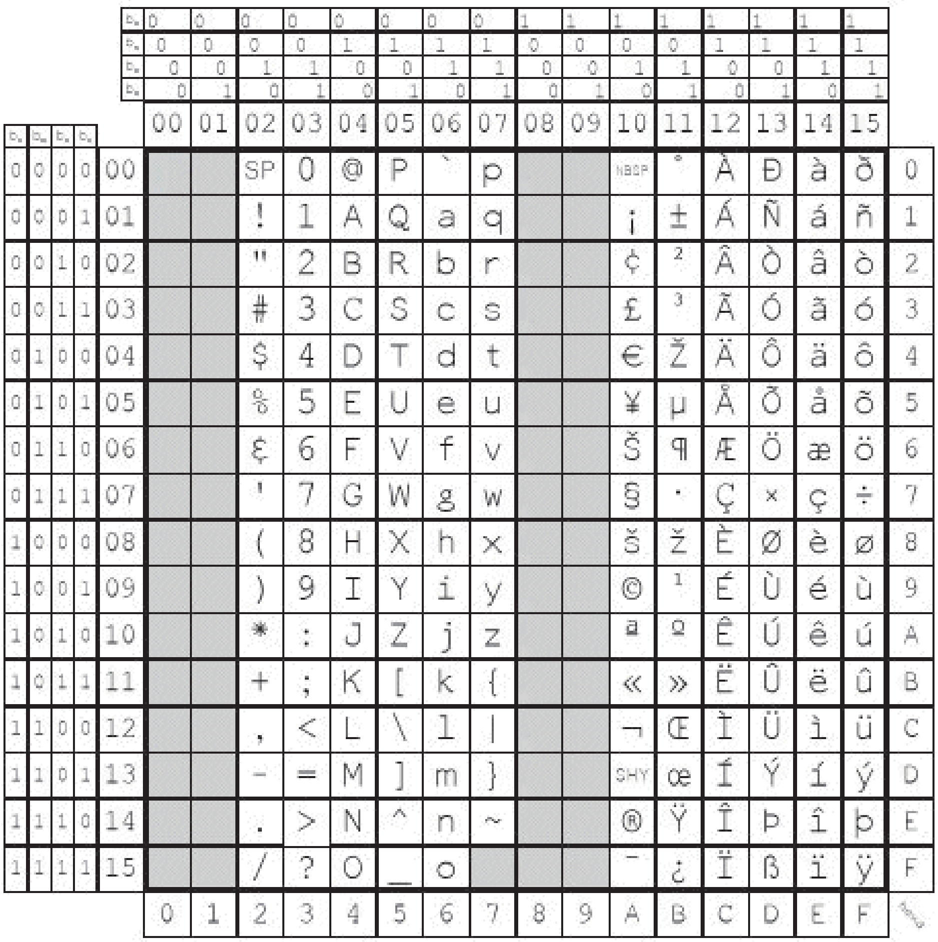
The following figure summarizes the available symbols and layout of Code Set ISO8859-9. This code set is made up of an ASCII character set plus its own unique character set. For a textual representation of this code set, see “ISO8859–9” on page 192.

First Hexadecimal Digit (columns) / Second Hexadecimal Digit (rows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	‘	p			NBSP	°	À	Ğ	à	ğ
1			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
2			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
3			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
4			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
6			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
7			'	7	G	W	g	w			§	·	Ç	×	ç	÷
8			(	8	H	X	h	x			¨	¸	È	Ø	è	ø
9			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
A			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
B			+	;	K	[	k	{			«	»	Ë	Û	ë	û
C			,	<	L	\	l	|			¬	1/4	Ì	Ü	ì	ü
D			-	=	M	]	m	}			SHY	1/2	Í	İ	í	ı
E			.	>	N	^	n	~			®	3/4	Î	Ş	î	ş
F			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ

Code Set ISO8859-15

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-15. For a textual representation of this code set, see "ISO8859–15" on page 194.



					0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
					0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
					0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
					0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
					00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			€	Ž	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			Š	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			š	ž	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	Œ	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SHY	œ	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	Ÿ	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ	F
					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Extended UNIX Code (EUC) Encoding Scheme

The EUC encoding scheme defines a set of encoding rules that can support one to four character sets. The encoding rules are based on the ISO2022 definition for the encoding of 7-bit and 8-bit data. The EUC encoding scheme uses control characters to identify some of the character sets. The following table shows the basic structure of all EUC encoding.

EUC	Character Encoding
CS0	0xxxxxxx
CS1	1xxxxxxx 1xxxxxxx 1xxxxxxxx 1xxxxxxx 1xxxxxxxx 1xxxxxxx ...
CS2	10001110 1xxxxxxx 10001110 1xxxxxxx 1xxxxxxxx 10001110 1xxxxxxx 1xxxxxxxx 1xxxxxxxx ...
CS3	10001111 1xxxxxxx 10001111 1xxxxxxx 1xxxxxxxx 10001111 1xxxxxxx 1xxxxxxxx 1xxxxxxxx ...

The term *EUC* denotes these general encoding rules. A code set based on EUC conforms to the EUC encoding rules but also identifies the specific character sets associated with the specific instances. For example, IBM-eucJP for Japanese refers to the encoding of the Japanese Industrial Standard characters according to the EUC encoding rules.

The first set (CS0) always contains an ISO646 character set. All of the other sets must have the most significant bit (MSB) set to 1 and can use any number of bytes to encode the characters. In addition, all characters within a set must have the following:

- Same number of bytes to encode all characters
- Same column display width (number of columns on a fixed-width terminal)

All characters in the third set (CS2) are always preceded with the control character SS2 (single-shift 2, 0x8e). Code sets that conform to EUC do not use the SS2 control character other than to identify the third set.

All characters in the fourth set (CS3) are always preceded with the control character SS3 (single-shift 3, 0x8f). Code sets that conform to EUC do not use the SS3 control character other than to identify the fourth set.

IBM-eucJP

The EUC for Japanese is an encoding consisting of single-byte and multibyte characters. The encoding is based on ISO2022, Japanese Industrial Standard (JIS), and EUC definitions.

The IBM-eucJP code set consists of the following character sets:

JISCII	JISX0201 Graphic Left character set
JISX0201.1976	Katakana/Hiragana Graphic Right character set
JISX0208.1983	Kanji level 1 and 2 character sets
IBM-udcJP	IBM-user definable characters

The IBM-eucJP code set is also capable of supporting the following:

JISX0212.1990 Supplemental Kanji

The IBM-eucJP code set is encoded as follows:

- CS0 maps JISX0201 Graphic Left characters starting at the 0x00 position.
- CS1 maps the JISX0208 character set starting at the 0xa1xa1 position. The positions 0xf5a1 through 0xfefe (940 characters) in CS1 are reserved as primary user-definable character areas.
- CS2 maps the JISX0201 Graphic Right starting at the 0x8ea1 position.
- CS3 is capable of mapping JISX0212 starting at the 0x8fa1a1 position. The positions 0x8ff5a1 through 0x8ffefe in CS3 (940 characters) are reserved as secondary user-definable character areas. The positions 0x8feea1 through 0x8ff4fe in CS3 (658 characters) are reserved for future system use. Therefore, users should not use this area.

IBM-eucCN

The EUC for the Simplified Chinese language is an encoding consisting of characters that contain 1 or 2 bytes. The EUC encoding is based on ISO2022, GB2312 as defined by the People's Republic of China, and multibyte character definitions unique to the manufacturer.

The current GB2312 defines 6,763 Simplified Chinese characters and 682 symbols. The IBM-eucCN is based upon a concept of one plane containing up to 94x94 characters. The encoding values of these characters range from 0xa1a1 to 0xfefe.

The GB2312 is mapped into the CS1 of EUC. Specifically, the IBM-eucCN consists of the following character sets:

ISO0646-IRV 7-bit ASCII character set, Graphic Left.

GB2312.1980 Contains 7445 characters. It occupies positions 0xa1a1 to 0xfedf (some user-defined characters scattered in 0xa1a1 to 0xfedf).

IBM-udcCN Scattered in GB. It occupies positions Oxa1a1 to Oxfedf. The actual values are:

```
a2a1 -- a2b0    a1e3 -- a2e4  a1ef -- a2f0
a2fd -- a1fe    a4f4 -- a4fe  a5f7 -- a5fe
a6b9 -- a6c0    a6d9 -- a6fe  a7c2 -- a7d0
a7f2 -- a7fe    a8bb -- a8c4  a8ea -- a9a3
a9f0 -- affe    a7fa -- d7fe  f8a1 -- fedf
```

IBM-sbdCN Scattered in GB. It occupies positions 0xfee0 to 0xfefe.

GB18030

GBK stands for Guo (national) Biao (Standard) Kuo (Extension). GB18030 expands the national ″Industry GB″ definition to contain all 20, 902 Han Characters defined in Unicode and additional DBCS symbols defined in Big-5 code (Traditional Chinese PC defacto standard). GB18030 defines all DBCS characters and symbols in use in the People's Republic of China and in Taiwan.

Locale	Code Set	Description
Zh_CN	GB18030	Simplified Chinese, GB18030 Locale

Code Range	Words	Marks
A1A1-A9FE	846	GB2312, GB12345 (GBK/1)
A840-A9A0	192	Big5, Symbols (GBK/5)
B0A1-F7FE	6768	GB2312 (GBK/2)

Code Range	Words	Marks
8140-A0FE	6080	GB13000 (GBK/3)
AA40-FEA0	8160	GB13000 (GBK/4)
AAA1-AFFE	564	User defined 1
F8A1-FEFE	658	User defined 2
A140-A7A0	672	User defined 3

IBM-eucTW

The EUC for the Traditional Chinese language is an encoding consisting of characters that contain 1, 2 and 4 bytes. The EUC encoding is based on ISO2022, the Chinese National Standard (CNS) as defined by Taiwan, and multibyte character definitions unique to the manufacturer.

The current CNS defines 13,501 Chinese characters and 684 symbols. The IBM-eucTW is based upon a concept of 15 planes, each containing up to 8836 (94x94) characters. The encoding values of these characters range from 0xa1a1 to 0xfefe. Characters have presently been defined for only 4 of the planes, with the other planes being reserved for future expansion.

The 15 planes are mapped into the CS1 and CS2 of EUC, with the CS2 of EUC consisting of 14 planes. Specifically, the IBM-eucTW consists of the following character sets:

ISO646-IRV
7-bit ASCII character set, Graphic Left.

CNS11643.1986-1
Plane 1, containing 6085 characters (5401+684). This plane uses positions 0ax1a1-0xc2c1 and 0xc4a1-0xfdcb.

CNS11643.1986-2
Plane 2, containing 7650 characters. This plane occupies positions 0x8ea2a1a1-0x8ea2f2c4.

CNS11643.1992-3
Plane 4, containing 7298 characters. This plane occupies positions 0x8ea4a1a1-0x8ea4eedc.

IBM-udcTW
Plane 12, containing 6204 characters. This plane is reserved for the User Defined Characters (udc) areas. It occupies the positions 0x8eaca1a1-0x8ea2f2c4.

IBM-sbdTW
Plane 13, containing 325 characters. This plane is reserved for symbols unique to the manufacturer. It occupies positions 0xeada1a1-0x8eada4cb.

Planes 3-11 are expected to occupy positions 0x8ea3xxxx to 0x8eabxxxx. Planes 14-15 are expected to occupy positions 0x8eaexxxx to 0x8eafxxxx.

Big5

The Traditional Chinese big5 locale, **Zh_TW**, code set is the most commonly used code set in the PC field that is used to support countries using Traditional Chinese.

Big5 code set defines 13056 characters and 1004 symbols. It includes 684 symbols in CNS11643.192, as well as 325 symbols unique to IBM.

Locale	Code Set	Description
Zh_TW	Big5 (IBM-950)	Traditional Chinese, Big5 Locale

Code Range for Big5 Locale:

Plan	Code Range	Description
1	A140H - A3E0H	Symbol and Chinese Control Code

Plan	Code Range	Description
1	A440H - C67EH	Commonly Used Characters
2	C940H - F9D5H	Less Commonly Used Characters
UDF	FA40H - FEFE	User-Defined Characters
	8E40H - A0FEH	User-Defined Characters
	8140H - 8DFEH	User-Defined Characters
	8181H - 8C82H	User-Defined Characters
	F9D6H - F9F1H	User-Defined Characters

Code Set	Words	Code Range	Marks
Commonly Used Area	5841	A140-C67E	
Less Commonly Used Area	7652	C940-F9D5	
ET Unique Area (1)	308	C6A1-C878	
ET Unique Area (2)	7	C8CD-C8D3	
IBM Unique Area	251	F286-F9A0	Low-Byte Range 81-A0
User-Defined Area (1)	785	FA40-FEFE	
User-Defined Area (2)	2983	8E40-A0FE	
User-Defined Area (3)	2041	8140-8DFE	
User-Defined Area (4)	354	8181-8C82	Low-Byte Range 81-AQ
User-Defined Area (5)	41	F9D6-F9FE	

IBM-eucKR

The EUC for the Korean language is an encoding consisting of single-byte and multibyte characters. The encoding is based on ISO2022, Korean Standard Code set, and EUC definitions.

The Korean EUC code set consists of the following main character groups:

- ASCII (English)
- Hangul (Korean characters)

The Hangul code set includes Hangul and Hanja (Chinese) characters. One Hangul character can comprise several consonants and vowels. However, most Hangul words can be expressed in Hanja. Each Hanja character has its own meaning and is more specific than Hangul.

The IBM-eucKR consists of the following character sets:

ISO646-IRV 7-bit ASCII character set, Graphic Left
KSC5601.1987-0 Korean Graphic Character Set, Graphic Right

IBM PC Code Sets

IBM PC code sets are the code sets originally supported on the IBM PC systems and AIX. The IBM PC code sets assign graphic characters to the Control One (C1) control area. Applications that depend on these control characters cannot support these code sets.

The ASCII characters are encoded with the most significant bit (MSB) zero in positions 0x20-0x7e. The extended Latin 1, combined with the IBM PC unique character sets, make up the extended set of characters which are encoded in positions 0x80-0xfe. The following table shows the location of the control, ASCII, and extended characters for the IBM-850 code set.

Character Encoding	Code Point	Description	Count
000xxxxx	00–1F	Controls	32
00100000	20	Space	1
0xxxxxxx	21–7E	7-bit	94
01111111	7F	Delete	1
1xxxxxxx	80–FE	8-bit	17
11111111	FF	All ones	1

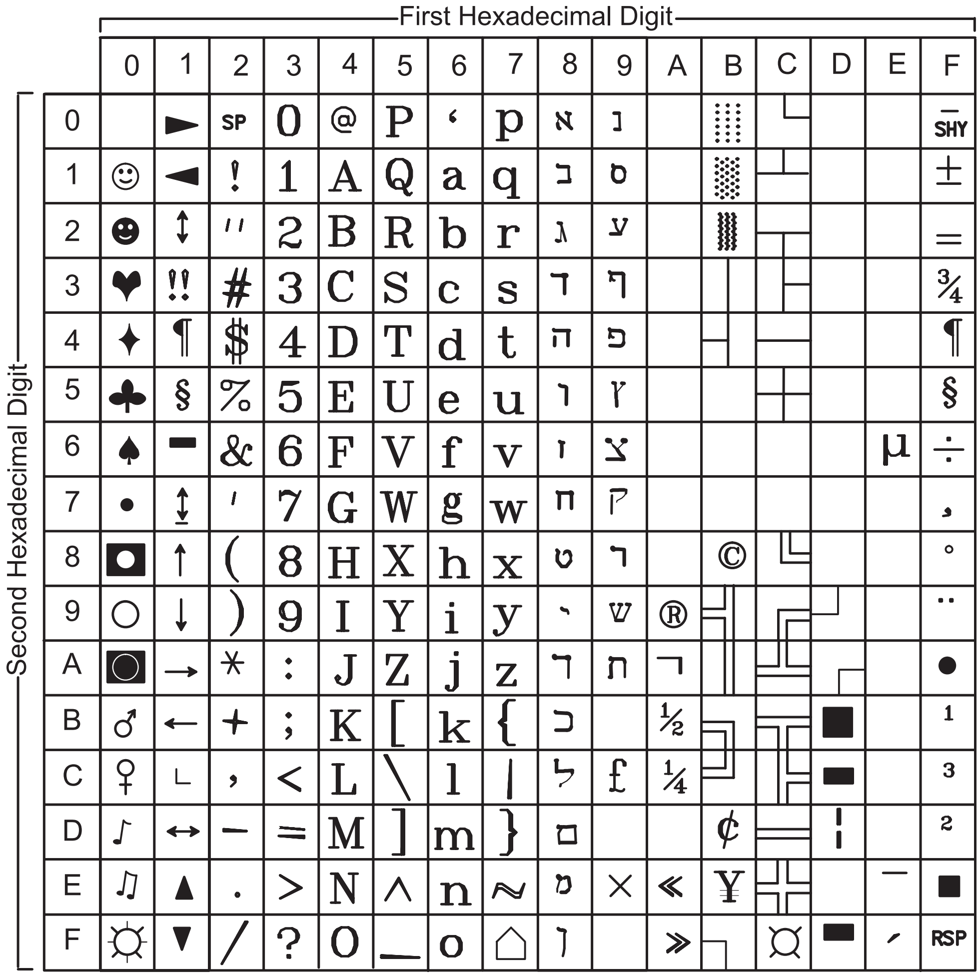
The IBM PC unique character set includes the following:

IBM PC Unique Character Set	
Symbol	**Return Code**
Florin sign	0x9f
Quarter-hashed	0xb0
Half-hashed	0xb1
Full-hashed	0xb2
Vertical bar	0xb3
Right-side middle	0xb4
Double right-side middle	0xb9
Double vertical bar	0xba
Double upper right-corner box	0xbb
Double lower right-corner box	0xbc
Upper right-corner box	0xbf
Lower left-corner box	0xc0
Bottom-side middle	0xc1
Top-side middle	0xc2
Left-side middle	0xc3
Center-box bar	0xc4
Intersection	0xc5
Double lower left-corner box	0xc8
Double upper left-corner box	0xc9
Double bottom-side middle	0xca
Double top-side middle	0xcb
Double left-side middle	0xcc
Double center-box bar	0xcd
Double intersection	0xce
Small i dotless	0xd5
Lower right-corner box	0xd9
Upper left-corner box	0xda
Bright character cell	0xdb

IBM PC Unique Character Set	
Symbol	**Return Code**
Bright character cell - lower half	0xde
Bright character cell - upper half	0xdf
Overbar	0xee
Middle dot, Product dot	0xfa
Vertical solid rectangle	0xfe

IBM-856

The following figure summarizes the available symbols and shows the layout of Code Set IBM-856. For a textual representation of this code set, see “IBM-856” on page 197.



IBM-921

The following figure summarizes the available symbols and shows the layout of Code Set IBM-921. For a textual representation of this code set, see "IBM-921" on page 200.

First Hexadecimal Digit (columns) / Second Hexadecimal Digit (rows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	‘	p			RSP	°	Ą	Š	ą	š
1			!	1	A	Q	a	q			º	±	Į	Ń	į	ń
2			"	2	B	R	b	r			¢	²	Ā	Ņ	ā	ņ
3			#	3	C	S	c	s			£	³	Ć	Ó	ć	ó
4			$	4	D	T	d	t			¤	ō̂	Ä	Ō	ä	ō
5			%	5	E	U	e	u			º	µ	Å	Õ	å	õ
6			&	6	F	V	f	v			¦	¶	Ę	Ö	ę	ö
7			'	7	G	W	g	w			§	·	Ē	×	ē	÷
8			(	8	H	X	h	x			Ø	ø	Č	Ų	č	ų
9			)	9	I	Y	i	y			©	¹	É	Ł	é	ł
A			*	:	J	Z	j	z			Ŗ	ŗ	Ź	Ś	ź	ś
B			+	;	K	[	k	{			«	»	Ė	Ū	ė	ū
C			,	<	L	\	l	|			¬	1/4	Ģ	Ü	ģ	ü
D			-	=	M	]	m	}			SHY	1/2	Ķ	Ż	ķ	ż
E			.	>	N	^	n	~			®	3/4	Ī	Ž	ī	ž
F			/	?	O	_	o				Æ	æ	Ļ	β	ļ	’

IBM-922

The following figure summarizes the available symbols and shows the layout of Code Set IBM-922. For a textual representation of this code set, see “IBM-922” on page 202.

First Hexadecimal Digit (columns) / Second Hexadecimal Digit (rows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	‘	p			RSP	°	À	Š	à	š
1			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
2			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
3			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
4			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
6			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
7			'	7	G	W	g	w			§	·	Ç	×	ç	÷
8			(	8	H	X	h	x			¨	¸	È	Ø	è	ø
9			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
A			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
B			+	;	K	[	k	{			«	»	Ë	Û	ë	û
C			,	<	L	\	l					¬	1/4	Ì	Ü	ì	ü
D			-	=	M	]	m	}			SHY	1/2	Í	Ý	í	ý
E			.	>	N	^	n	~			®	3/4	Î	Ž	î	ž
F			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ

IBM-943 and IBM-932

Each of the Japanese IBM PC code sets are an encoding consisting of single-byte and multibyte coded characters. The encoding is based on the IBM PC code set and places the JIS characters in shifted positions. This is referred to as *Shift-JIS* or SJIS.

IBM-943 is a newer code set for the Japanese locale than IBM-932. IBM-943 is a compatible code set for the Japanese Microsoft Windows environment. This code set is known as **1983 ordered shift-JIS**. The differences between IBM-932 and IBM-943 are as follows:

- Previous JIS sequence (1978 ordered) is applied for IBM-932 while newer JIS sequence (1983 ordered) is applied for IBM-943.
- NEC selected characters are added to IBM-943.
- NEC's IBM selected characters are added to IBM-943.

The IBM-932 code set consists of the following character sets:

JISCII
JISX0201 Graphic Left character set

JISX0201.1976
Katakana/Hiragana Graphic Right character set

JISX0208.1983
Kanji level 1 and 2 character sets

IBM-udcJP
IBM user-definable characters

The IBM-943 code set consists of the following character sets:

JISCII
JISX0201 Graphic Left character set

JISX0201.1976
Katakana/Hiragana Graphic Right character set

JISX0208.1990
Kanji level 1 and 2 character sets

IBM-udcJP
IBM user-definable characters and NEC's IBM selected characters and NEC selected characters

The first byte of each character is used to determine the number of bytes for a given character. The values 0x20-0x7e and 0xa1-oxdf are used to encode JISX0201 characters, with exceptions. The positions 0x81-0x9f and 0xe0-0xfc are reserved for use as the first byte of a multibyte character. The JISX0208 characters are mapped to the multibyte values starting at 0x8140. The second byte of a multibyte character can have any value. The Shift-JIS table shows where these characters are located on the code set.

Character Encoding	Code Point	Description	Count
000xxxxx	00–1f	Controls	32
00100000	20	Space	1
0xxxxxxx	21–7E	7-bit ASCII	94
01111111	7F	Delete	1
10000000	80	Undefined	1
100xxxxx 01xxxxxx	[81–9F] [40–7E]	Double byte	1953
100xxxxx 1xxxxxxx	[81–9F] [80–FC]	Double byte	3975
10100000	A0	Undefined	1
1xxxxxxx	A1–DF	7-bit single byte	63
111xxxxx 01xxxxxx	[E0–FC] [40–7E]	Double byte	1827
111xxxxx 1xxxxxxx	[E0–FC] [80–FC]	Double byte	3625
11111101	FD	Undefined	1
11111110	FE	Undefined	1

Character Encoding	Code Point	Description	Count
11111111	FF	Undefined	1

The following table shows the DBCS portion of IBM-943.

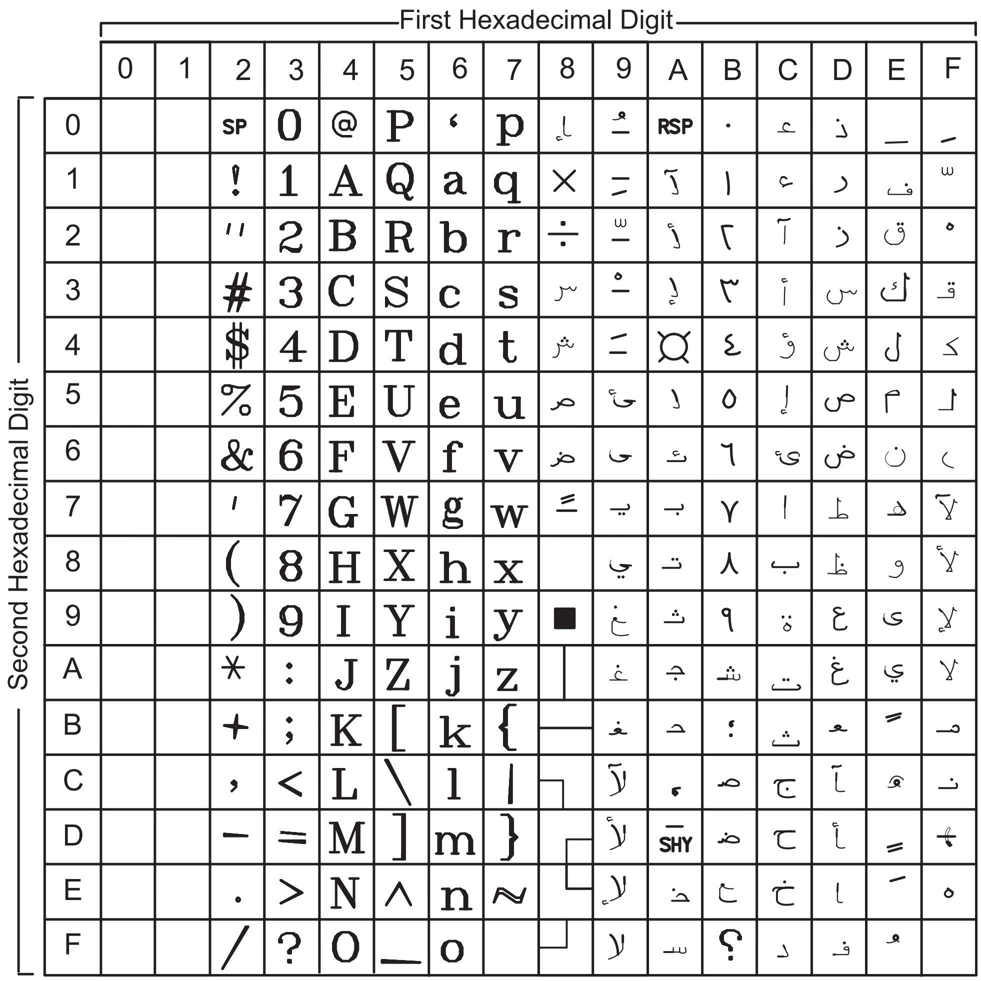
Code Point	Description
[81–84] [40–7E] and [81–84] [80–F0]	JIS X 0208 (Non-Kanji)
[87] [40–7E] and [87] [80–F0]	NEC selected characters
[89–98] [40–7E] and [88] [9F-F0], [89–97] [80–F0], [98] [80–9F]	JIS X0208 (Level-1 Kanji)
[99–9F] [40–7E] and [98] [9F-F0], [99–9F] [80–F0]	JIS X0208 (Level-2 Kanji)
[E0–EA] [40–7E] and [E0–EA] [80–F0]	JIS X0208 (Level-2 Kanji)
[ED–EE] [40–7E] and [ED–EE] [80–F0]	NEC IBM selected characters
[F0–F9] [40–7E] and [F0–F9] [80–F0]	User-defined characters
[FA] [40–5C]	IBM selected characters (non-Kanji)
[FA] [5C-7E], [FB-FC] [40–7E] and [FA-FC] [80–F0]	IBM selected characters (Kanji)

The following table shows the DBCS portion of IBM-932.

Code Point	Description
[81–98] [40–7E] and [81–97] [80–FC], [98] [80–9F]	JIS X 0208 (Level-1 Kanji)
[99–9F] [40–7E] and [98] [9F-FC], [99–9F] [80–FC]	JIS X 0208 (Level-2 Kanji)
[E0–EF] [40–7E] and [E0–EF] [80–FC]	JIS X 0208 (Level-2 Kanji)
[F0–F9] [40–7E] and [F0–F9] [80–FC]	User-defined characters
[FA–FC] [40–7E] and [FA–FC] [80–FC]	IBM selected characters

IBM-1046

The following figure summarizes the available symbols and shows the layout of Code Set IBM-1046. For a textual representation of this code set, see “IBM-1046” on page 205.



IBM-1124

The following figure summarizes the available symbols and shows the layout of Code Set IBM-1124. For a textual representation of this code set, see “IBM-1124” on page 208.

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0			(SP)	0	@	P	`	p			(RSP)	А	Р	а	р	№
-1			!	1	A	Q	a	q			Ё	Б	С	б	с	ё
-2			"	2	B	R	b	r			Ђ	В	Т	в	т	ђ
-3			#	3	C	S	c	s			Ґ	Г	У	г	у	ґ
-4			$	4	D	T	d	t			Є	Д	Ф	д	ф	є
-5			%	5	E	U	e	u			Ѕ	Е	Х	е	х	ѕ
-6			&	6	F	V	f	v			І	Ж	Ц	ж	ц	і
-7			'	7	G	W	g	w			Ї	З	Ч	з	ч	ї
-8			(	8	H	X	h	x			Ј	И	Ш	и	ш	ј
-9			)	9	I	Y	i	y			Љ	Й	Щ	й	щ	љ
-A			*	:	J	Z	j	z			Њ	К	Ъ	к	ъ	њ
-B			+	;	K	[	k	{			Ћ	Л	Ы	л	ы	ћ
-C			,	<	L	\	l	|			Ќ	М	Ь	м	ь	ќ
-D			-	=	M	]	m	}			(SHY)	Н	Э	н	э	§
-E			.	>	N	^	n	~			Ў	О	Ю	о	ю	ў
-F			/	?	O	_	o				Џ	П	Я	п	я	џ

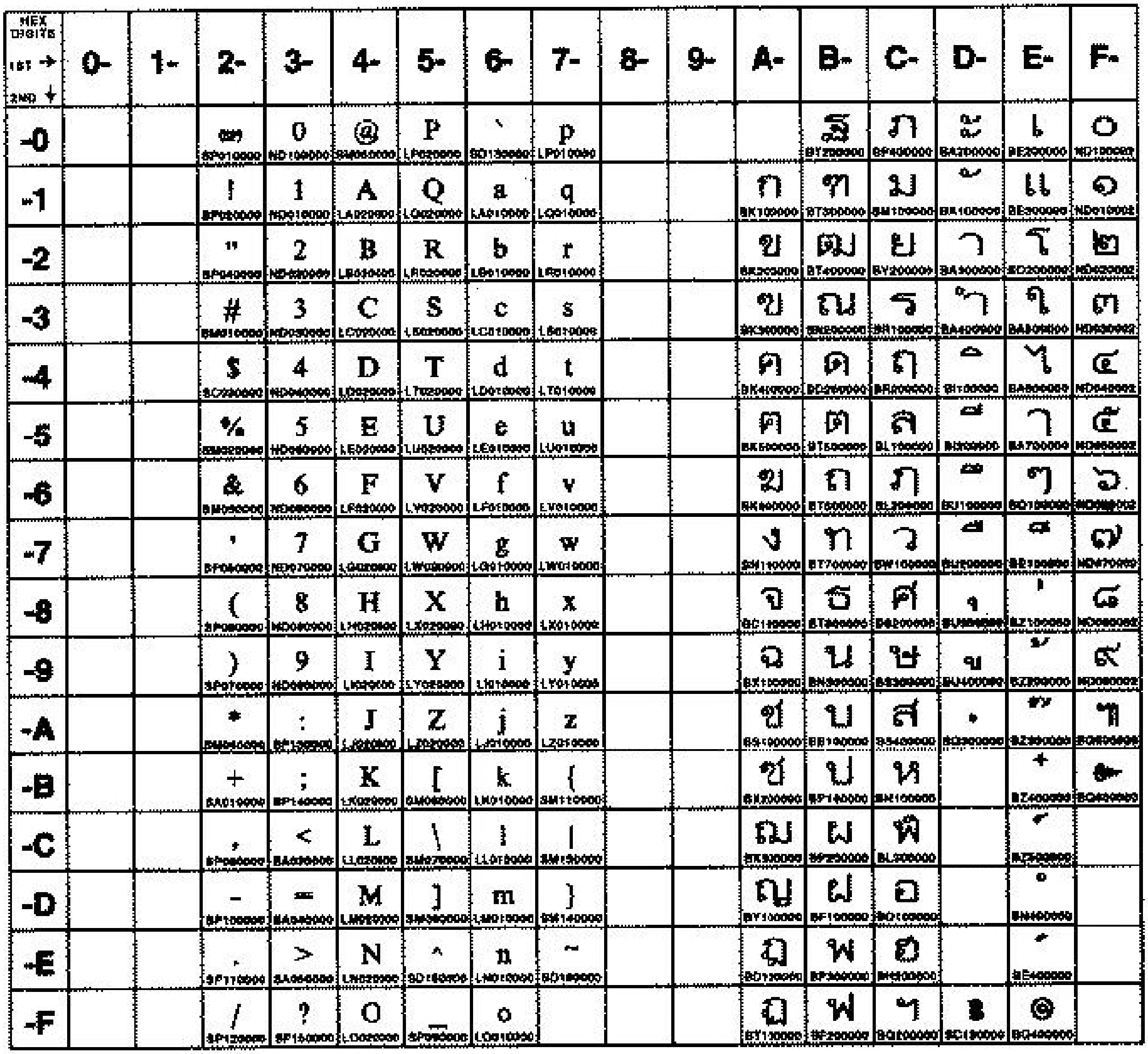
IBM-1129

The following figure summarizes the available symbols and shows the layout of Code Set IBM-1129. For a textual representation of this code set, see "IBM-1129" on page 210.

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0			(SP)	0	@	P	\`	p			(RSP)	°	À	Đ	à	đ
-1			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
-2			"	2	B	R	b	r			¢	²	Â	̉	â	̣
-3			#	3	C	S	c	s			£	³	Ă	Ó	ă	ó
-4			$	4	D	T	d	t			¤	Ÿ	Ä	Ô	ä	ô
-5			%	5	E	U	e	u			¥	µ	Å	Ơ	å	ơ
-6			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
-7			'	7	G	W	g	w			§	·	Ç	×	ç	÷
-8			(	8	H	X	h	x			œ	Œ	È	Ø	è	ø
-9			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
-A			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
-B			+	;	K	[	k	{			«	»	Ë	Û	ë	û
-C			,	<	L	\	l	|			¬	¼	̀	Ü	́	ü
-D			-	=	M	]	m	}			(SHY)	½	Í	Ư	í	ư
-E			.	>	N	^	n	~			®	¾	Î	̃	î	₫
-F			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ

TIS-620

The following figure summarizes the available symbols and shows the layout of Code Set TIS-620. For a textual representation of this code set, see “TIS-620” on page 213.



UCS-2 and UTF-8

AIX provides a set of codesets that address the needs of a particular language or a language group. None of the codesets represented in the ISO8859 family of codesets, the PC codesets, nor the Extended UNIX Code (EUC) codesets allow the mixing of characters from different scripts. With ISO8859-1, you can mix and represent the Latin 1 characters (languages principally spoken in the U.S., Canada, Western Europe, and Latin America). ISO8859-2 covers Eastern European languages; ISO8859-5 covers Cyrillic, ISO8859-6 covers Arabic, ISO8859-7 covers Greek, ISO8859-8 covers Hebrew, ISO8859-9 covers Turkish, IBM-eucJP covers Japanese, IBM-eucKR covers Korean, IBM-eucTW covers Traditional Chinese. The point is that none of the above codesets covers all of the languages.

The International Organization for Standardization (ISO) addressed the limited language coverage by code sets by adopting Unicode as the encoding for the 2-octet form of the ISO10646 Universal Multiple-Octet Coded Character Set (UCS-2). The 32-bit form of ISO10646 is known as UCS-4 for 4-octet form. AIX uses the 16-bit form of ISO10646 and uses the standard label *UCS-2* to describe this encoding.

Although UCS-2 is ideal for an internal process code, it is not suitable for encoding plain text on traditional byte-oriented systems, such as AIX. Therefore, the external file code is The Open Group's File System Safe UCS Transformation Format (FSS-UTF). This transformation format encoding is also known as UTF-8, and *UTF-8* is the label that is used for this encoding on AIX.

ISO10646 UCS-2 (Unicode)

Universal Coded Character Set (UCS) is the name of the ISO10646 standard that defines a single code for the representation, interchange, processing, storage, entry, and presentation of the written form of all the major languages of the world.

ISO10646 defines canonical character codes with a length of 32 bits, which provides code numbers for over 4 billion characters. When used in canonical form to represent text, the coding is referred to as UCS-4 for Universal Coded Character Set 4-byte form.

The code values from 0x0000 through 0xFFFF of ISO 10646 can be represented by a uniform character encoding of 16 bits. When used in this form to represent text, these codes are referred to as UCS-2, for Universal Character Set 2-octet form. This range is also called the Basic Multilingual Plane (BMP) of ISO10646. The standard is arranged so that the most useful characters, covering all major existing standards worldwide, are assigned within this range.

The character code values of UCS-2 are identical to those of the Unicode character encoding standard published by the Unicode Consortium. UCS-2 defines codes for characters used in all major written languages. In addition to a set of scientific, mathematic, and publishing symbols, UCS-2 covers the following scripts:

- Arabic
- Armenian
- Bengali
- Bopomofo
- Cyrillic
- Devanagari
- Georgian
- Greek
- Gujarati
- Gurmukhi
- Hangul
- Chinese Hanzi

- Hebrew
- Hiragana
- International Phonetic Alphabet (IPA)
- Katakana
- Japanese Kanji
- Kannada
- Korean Hanja
- Laotian
- Latin
- Malayalam
- Oriya
- Tamil
- Teluga
- Thai
- Tibetan

The ability of AIX to display characters in the scripts mentioned above is limited to the availability of fonts. AIX provides bitmap fonts for most of the major languages of the world, as well as a Unicode-based scalable TrueType font. Use of this font requires the TrueType font rasterizer for AIX, which is a separately installable feature.

UCS-2 encodes a number of combining characters, also known as non-spacing marks for floating diacritics. These characters are necessary in several scripts including Indic, Thai, Arabic, and Hebrew. The combining characters are used for generating characters in Latin, Cyrillic, and Greek scripts. However, the presence of combining characters creates the possibility for an alternative coding for the same text. Although the coding is unambiguous and data integrity is preserved, the processing of text that contains combining characters is more complex. To provide conformance for applications that choose not to deal with the combining characters, ISO10646 defines the following implementation levels:

Level 1
: Does not allow combining characters.

Level 2
: Allows combining marks from Thai, Indic, Hebrew, and Arabic scripts.

Level 3
: Allows combining marks, including ones for Latin, Cyrillic, and Greek.

UCS-4 and UTF-32

The Unicode standard is used to define standard character encodings for most of the commonly used languages in the world. The 2-byte form of this standard is commonly referred to as *UCS-2*. However, UCS-2 is only capable of representing a maximum of 65,536 characters as a 2-byte quantity. The 4-byte form of Unicode is referred to as *UCS-4* or *UTF-32*, and is capable of defining the complete extensions of Unicode, with a maximum of over 1,000,000 unique characters definable.

UTF-8 (UCS Transformation Format)

The Open Group has developed a transformation format for UCS designed for use in existing file systems. The intent is that UCS will be the process code for the transformation format, which is usable as a file code.

UTF-8 has the following properties:

- It is a superset of ASCII, in which the ASCII characters are encoded as single-byte characters with the same numeric value.

- No ASCII code values occur in multibyte characters, other than those that represent the ASCII characters.
- The first byte of a character indicates the number of bytes to follow in the multibyte character sequence and cannot occur anywhere else in the sequence.

The UTF-8 encodes UCS values in the 0 through 0x7FFFFFFF range using multibyte characters with lengths of 1, 2, 3, 4, 5, and 6 bytes. Single-byte characters are reserved for the ASCII characters in the 0 through 0x7f range. These characters all have the high order bit set to 0. For all character encodings of more than one byte, the initial byte determines the number of bytes used, and the high-order bit in each byte is set. Every byte that does not start with the bit combination of 10xxxxxx, where x represents a bit that may be 0 or 1, is the start of a UCS character sequence. The following table provides UTF-8 multibyte codes:

Bytes	Bits	Hex Minimum	Hex Maximum	Byte Sequence in Binary
1	7	00000000	0000007F	0xxxxxxx
2	11	00000080	000007FF	110xxxxx 10xxxxxx
3	16	00000800	0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
4	21	00010000	001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
5	26	00200000	03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxx 10xxxxxx
6	31	04000000	7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

The UCS value is just the concatenation of the *x* bits in the multibyte encoding. When there are multiple ways to encode a value (for example, UCS 0), only the shortest encoding is permitted.

The following subset of UTF-8 is used to encode UCS-2:

Bytes	Bits	Hex Minimum	Hex Maximum	Byte Sequence in Binary
1	7	00000000	0000007F	0xxxxxxx
2	11	00000080	000007FF	110xxxxx 10xxxxxx
3	16	00000800	0000FFFF	1110xxxx 10xxxxxx 10xxxxxx

This subset of UTF-8 requires a maximum of three (3) bytes.

UTF-16

UTF-16 is the UCS Transformation Format for 16 planes of Group 00. UTF-16 is the ISO/IEC encoding that is equivalent to the Unicode Standard with the use of surrogates. In UTF-16, each UCS-2 code value represents itself. Non-BMP code values of ISO/EIC 10646 in planes 1..16 are represented using pairs of special codes. UTF-16 defines the transformation between the UCS-4 code positions in planes 1 to 16 of Group 00 and the pairs of special codes, and is identical to the transformation defined in the Unicode Standard.

Related Information

Low Function Terminal (LFT) Subsystem Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*. Keyboard Overview

Chapter 5. Converters Overview for Programming

National Language Support (NLS) provides a base for internationalization in which data often can be changed from one code set to another. Support of several standard converters for this purpose is provided. This section discusses the following aspects of conversion:

- “Standard Converters”
- “Understanding libiconv” on page 84
- “Using Converters” on page 87
- “List of Converters” on page 89

Data sent by one program to another program residing on a remote host may require conversion from the code set of the source machine to that of the receiver. For example, when communicating with a VM system, the workstation converts its ISO8859-1 data to an EBCDIC form.

Code sets define graphic characters and control character assignments to code points. These coded characters must also be converted when a program obtains data in one code set but displays it in another code set.

The system provides the following conversion interfaces:

iconv command
: Allows you to request a specific conversion by naming the *FromCode* and *ToCode* code sets.

libiconv functions
: Allows applications to request converters by name. For more information, see “Understanding libiconv” on page 84.

The system provides ready-to-use libraries of converters. The converter libraries are found in the **/usr/lib/nls/loc/iconv/*** and **/usr/lib/nls/loc/iconvTable/*** directories. Do not define your own converter unless absolutely necessary.

In addition to code set converters, the converter library also provides a set of network interchange converters. In a network environment, the code sets of the communications systems and the protocols of communication determine how the data should be converted.

Interchange converters are used to convert data sent from one system to another. Conversions from one internal code set to another code set require code set converters. When data must be converted from a sender’s code set to a receiver’s code set or from 8-bit data to 7-bit data, a uniform interface is required. The **iconv** subroutines provide this interface.

Standard Converters

The system supports standard converters for use with the **iconv** command and subroutines. The following are code set converter types:

Table converter
: Converts single-byte stateless code sets. Performs a table translation from one byte to another byte. For more information, see “PC, ISO, and EBCDIC Code Set Converters” on page 90.

Algorithmic converter
: Performs a conversion that cannot be implemented using a simple single-byte mapping table. All multibyte converters are implemented using this method. For more information, see “Multibyte Code Set Converters” on page 94.

The following are interchange converter types:

7–bit Converts between internal code sets and ISO2022 standard interchange formats (7-bit). For more information, see "Interchange Converters—7-bit" on page 98.

8–bit Converts between internal code sets and ISO2022 standard interchange formats (8-bit). For more information, see "Interchange Converters—8-bit" on page 100.

compound text
Converts between compound text and internal code sets. For more information, see "Interchange Converters—Compound Text" on page 103.

uucode
Provides the same mapping as that defined in the **uuencode** and **uudecode** command. For more information, see "Interchange Converters—uucode" on page 105.

UCS-2 Converts between UCS-2 and other code sets. For more information, see "UCS-2 Interchange Converters" on page 106.

UCS-4 Converts between UCS-4 and other code sets. For more information, see "UCS-4 and UTF-32" on page 81.

UTF-8 Converts between UTF-8 and other code sets. For more information, see"UTF-8 Interchange Converters" on page 108.

UTF-16
Converts between UTF-16 and other code sets. For more information, see"UTF-16" on page 82.

UTF-32
Converts between UTF-32 and other code sets. For more information, see"UCS-4 and UTF-32" on page 81.

Low-level converters can be used by some of the interchange converters. For a list of these converters, see "Miscellaneous Converters" on page 110.

Using the iconv Command

Any converter installed in the system can be used through the **iconv** command, which uses the **iconv** library. The **iconv** command acts as a filter for converting from one code set to another. For example, the following command filters data from PC Code (IBM-850) to ISO8859-1:

```
cat File | iconv -f IBM-850 -t ISO8859-1 | tftp -p - host /tmp/fo
```

The **iconv** command converts the encoding of characters read from either standard input or the specified file and then writes the results to standard output.

Understanding libiconv

The **iconv** application programming interface (API) consists of the following subroutines that accomplish conversion:

iconv_open
Performs the initialization required to convert characters from the code set specified by the *FromCode* parameter to the code set specified by the *ToCode* parameter. The strings specified are dependent on the converters installed in the system. If initialization is successful, the converter descriptor, **iconv_t**, is returned in its initial state.

iconv Invokes the converter function using the descriptor obtained from the **iconv_open** subroutine. The *inbuf* parameter points to the first character in the input buffer, and the *inbytesleft* parameter indicates the number of bytes to the end of the buffer being converted. The *outbuf* parameter points to the first available byte in the output buffer, and the *outbytesleft* parameter indicates the number of available bytes to the end of the buffer.

For state-dependent encoding, the subroutine is placed in its initial state by a call for which the *inbuf* value is a null pointer. Subsequent calls with the *inbuf* parameter as something other than a null pointer cause the internal state of the function to be altered as necessary.

iconv_close
Closes the conversion descriptor specified by the *cd* variable and makes it usable again.

In a network environment, the following factors determine how data should be converted:

- Code sets of the sender and the receiver
- Communication protocol (8-bit or 7-bit data)

The following table outlines the conversion methods and recommends how to convert data in different situations. See the "Interchange Converters—7-bit" on page 98 and the "Interchange Converters—8-bit" on page 100 for more information.

Outline of Methods and Recommended Choices				
	Communication with system using the same code set		**Communication with system using different code set (or receiver's code set is unknown)**	
	Protocol		**Protocol**	
Method to choose	7-bit only	8-bit	7-bit only	8-bit
as is	Not valid	Best choice	Not valid	Not valid if remote code set is unknown
fold7	OK	OK	Best choice	OK
fold8	Not valid	OK	Not valid	Best choice
uucode	Best choice	OK	Not valid	Not valid

If the sender uses the same code set as the receiver, the following possibilities exist:

- When protocol allows 8-bit data, the data can be sent without conversions.
- When protocol allows only 7-bit data, the 8-bit code points must be mapped to 7-bit values. Use the **iconv** interface and one of the following methods:

uucode
Provides the same mapping as the **uuencode** and **uudecode** commands. This is the recommended method. For more information, see "Interchange Converters—uucode" on page 105.

7–bit
Converts internal code sets using 7-bit data. This method passes ASCII without any change. For more information, see "Interchange Converters—7-bit" on page 98.

If the sender uses a code set different from the receiver, there are two possibilities:

- When protocol allows only 7-bit data, use the fold7 method.
- When protocol allows 8-bit data and you know the receiver's code set, use the iconv interface to convert the data. If you do not know the receiver's code set, use the following method:

8–bit
Converts internal code sets to standard interchange formats. The 8-bit data is transmitted and the information is preserved so that the receiver can reconstruct the data in its code set. For more information, see "Interchange Converters—8-bit" on page 100.

Using the iconv_open Subroutine

The following examples illustrate how to use the **iconv_open** subroutine in different situations:

- When the sender and receiver use the same code sets, and if the protocol allows 8-bit data, you can send data without converting it. If the protocol allows only 7-bit data, do the following:

```
Sender:
 cd = iconv_open("uucode", nl_langinfo(CODESET));


Receiver:
 cd = iconv_open(nl_langinfo(CODESET), "uucode");
```

- Whne the sender and receiver use different code sets, and if the protocol allows 8-bit data and the receiver's code set is unknown, do the following:

```
Sender:
 cd = iconv_open("fold8", nl_langinfo(CODESET));


Receiver:
 cd = iconv_open(nl_langinfo(CODESET),"fold8" );
```

  If the protocol allows only 7-bit data, do the following:

```
Sender:
 cd = iconv_open("fold7", nl_langinfo(CODESET));


Receiver:
 cd = iconv_open(nl_langinfo(CODESET), "fold7" );
```

The **iconv_open** subroutine uses the **LOCPATH** environment variable to search for a converter whose name is in the following form:

iconv/*FromCodeSet_ToCodeSet*

The *FromCodeSet* string represents the sender's code set, and the *ToCodeSet* string represents the receiver's code set. The underscore character separates the two strings.

Note: All **setuid** and **setgid** programs ignore the **LOCPATH** environment variable.

Because the **iconv** converter is a loadable object module, a different object is required when running in the 64-bit environment. In the 64-bit environment, the **iconv_open** routine uses the **LOCPATH** environment variable to search for a converter whose name is in the following form:

```
iconv/FromCodeSet_ToCodeSet__64.
```

The iconv library automatically chooses whether to load the standard converter object or the 64-bit converter object. If the **iconv_open** subroutine does not find the converter, it uses the *from,to* pair to search for a file that defines a table-driven conversion. The file contains a conversion table created by the **genxlt** command.

The iconvTable converter uses the **LOCPATH** environment variable to search for a file whose name is in the following form:

iconvTable/*FromCodeSet_ToCodeSet*

If the converter is found, it performs a load operation and is initialized. The converter descriptor, **iconv_t**, is returned in its initial state.

Converter Programs versus Tables

Converter programs are executable functions that convert data according to a set of rules. Converter tables are single-byte conversion tables that perform stateless conversions. Programs and tables are in separate directories, as follows:

/usr/lib/nls/loc/iconv	Converter programs

/usr/lib/nls/loc/iconvTable	Converter tables

After a converter program is compiled and linked with the **libiconv.a** library, the program is placed in the **/usr/lib/nls/loc/iconv** directory.

To build a table converter, build a source converter table file. Use the **genxlt** command to compile translation tables into a format understood by the table converter. The output file is then placed in the **/usr/lib/nls/loc/iconvTable** directory.

Unicode and Universal Converters

Unicode (or UCS-2) conversion tables are found in:

```
$LOCPATH/uconvTable/*CodeSet*
```

The **$LOCPATH/uconv/UCSTBL** converter program is used to perform the conversion to and from UCS-2 using the **iconv** utilities.

A Universal converter program is provided that can be used to convert between any two code sets whose conversions to and from UCS-2 is defined. Given the following uconv tables:

```
X     -> UCS-2
UCS-2 -> Y
```

a universal conversion can be defined that maps the following:

```
X -> UCS-2 -> Y
```

by use of the **$LOCPATH/iconv/Universal_UCS_Conv**.

Universal UCS Converter

UCS-2 is a universal 16-bit encoding that can be used as an interchange medium to provide conversion capability between virtually any code sets. The conversion can be accomplished using the Universal UCS Converter, which converts between any two code sets XXX and YYY as follows:

```
XXX <-> UTF-32 <-> YYY
```

The XXX and YYY conversions must be included in the supported List of UCS-2 Interchange Converters, and must be installed on the system.

The universal converter is installed as the file **/usr/lib/nls/loc/iconv/Universal_UCS_Conv**.

The conversion between multibyte and wide character code depends on the current locale setting. Do not exchange wide character codes between two processes, unless you have knowledge that each locale that might be used handles wide character codes in a consistent fashion. Most locales for this operating system use the Unicode character value as a wide character code, except locales based on IBM-eucTW codesets.

Using Converters

The iconv interface is a set of the following subroutines used to open, perform, and close conversions:

- **iconv_open**
- **iconv**
- **iconv_close**

Code Set Conversion Filter Example

The following example shows how you can use these subroutines to create a code set conversion filter that accepts the *ToCode* and *FromCode* parameters as input arguments:

```
#include <stdio.h>
#include <nl_types.h>
#include <iconv.h>
#include <string.h>
#include <errno.h>
#include <locale.h>

#define ICONV_DONE() (r>=0)
#define ICONV_INVAL() (r<0) && (errno==EILSEQ))
#define ICONV_OVER() (r<0) && (errno==E2BIG))
#define ICONV_TRUNC() (r<0) && (errno==EINVAL))

#define USAGE 1
#define ERROR 2
#define INCOMP 3

char ibuf[BUFSIZ], obuf[BUFSIZ];

extern int errno;

main (argc,argv)
int argc;
char **argv;
{
 size_t  ileft,oleft;
 nl_catd catd;
 iconv_t cd;
 int r;
 char *ip,*op;

 setlocale(LC_ALL,"");
 catd = catopen (argv[0],0);

 if(argc!=3){
  fprintf(stderr,
   catgets (catd,NL_SETD,USAGE,"usage;conv fromcode tocode\n"));
  exit(1);
 }

 cd=iconv_open(argv[2],argv[1]);

ileft=0;

while(!feof(stdin)) {
 /*
 * After the next operation,ibuf will
 * contain new data plus any truncated
 * data left from the previous read.
 */
 ileft+=fread(ibuf+ileft,1,BUFSIZ-ileft,stdin);
 do {
  ip=ibuf;
  op=obuf;
  oleft=BUFSIZ;

  r=iconv(cd,&ip,&ileft,&op,&oleft);

  if(ICONV_INVAL()){
   fprintf(stderr,
      catgets(catd,NL_SETD,ERROR,"invalid input\n"));
   exit(2);
 }
```

```
 fwrite(obuf,1,BUFSIZ-oleft,stdout);

 if(ICONV_TRUNC() || ICONV_OVER())
  /*
  *Data remaining in buffer-copy
  *it to the beginning
  */

  memcpy(ibuf,ip,ileft);

  /*
  *loop until all characters in the input
  *buffer have been converted.
  */
 } while(ICONV_OVER());
}

 if(ileft!=0){
  /*
  *This can only happen if the last call
  *to iconv() returned ICONV_TRUNC, meaning
  *the last data in the input stream was
  *incomplete.
  */
 fprintf(stderr,catgets(catd,NL_SETD,INCOMP,"input incomplete\n"));
 exit(3);
 }

 iconv_close(cd);
 exit(0);
}
```

Naming Converters

Code set names are in the form *CodesetRegistry*-*CodesetEncoding* where:

CodesetRegistry — Identifies the registration authority for the encoding. The *CodesetRegistry* must be made of characters from the portable code set (usually A-Z and 0-9).

CodesetEncoding — Identifies the coded character set defined by the registered authority.

The *from,to* variable used by the **iconv** command and **iconv_open** subroutine identifies a file whose name should be in the form ***/usr/lib/nls/loc/iconv/**%f_%t* or ***/usr/lib/nls/loc/iconvTable/**%f_%t*, where:

%f Represents the *FromCode* set name

%t Represents the *ToCode* set name

List of Converters

Converters change data from one code set to another. The sets of converters supported with the iconv library are listed in the following sections. All converters shipped with the BOS Runtime Environment are located in the **/usr/lib/nls/loc/iconv/*** or **/usr/lib/nls/loc/iconvTable/*** directory.

These directories also contain *private* converters; that is, they are used by other converters. However, users and programs should only depend on the converters in the following lists.

Any converter shipped with the BOS Runtime Environment and not listed here should be considered private and subject to change or deletion. Converters supplied by other products can be placed in the **/usr/lib/nls/loc/iconv/*** or **/usr/lib/nls/loc/iconvTable/*** directory.

Programmers are encouraged to use registered code set names or code set names associated with an application. The X Consortium maintains a registry of code set names for reference. See Chapter 4, “Code Sets for National Language Support,” on page 49 for more information about code sets.

PC, ISO, and EBCDIC Code Set Converters

These converters provide conversion between PC, ISO, and EBCDIC single-byte stateless code sets. The following types of conversions are supported: PC to/from ISO, PC to/from EBCDIC, and ISO to/from EBCDIC.

Conversion is provided between compatible code sets such as Latin-1 to Latin-1 and Greek to Greek. However, conversion between different EBCDIC national code sets is not supported. For information about converting between incompatible character sets, refer to the “Interchange Converters—7-bit” on page 98 and the “Interchange Converters—8-bit” on page 100.

Conversion tables in the **iconvTable** directory are created by the **genxlt** command.

Compatible Code Set Names

The following table lists code set names that are compatible. Each line defines to/from strings that may be used when requesting a converter.

Note: The PC and ISO code sets are ASCII-based.

Code Set Compatibility				
Character Set	**Languages**	**PC**	**ISO**	**EBCDIC**
Latin-1	U.S. English, Portuguese, Canadian French	N/A	ISO8859-1	IBM-037
Latin-1	Danish, Norwegian	N/A	ISO8859-1	IBM-277
Latin-1	Finnish, Swedish	N/A	ISO8859-1	IBM-278
Latin-1	Italian	N/A	ISO8859-1	IBM-280
Latin-1	Japanese	N/A	ISO8859-1	IBM-281
Latin-1	Spanish	N/A	ISO8859-1	IBM-284
Latin-1	U.K. English	N/A	ISO8859-1	IBM-285
Latin-1	German	N/A	ISO8859-1	IBM-273
Latin-1	French	N/A	ISO8859-1	IBM-297
Latin-1	Belgian, Swiss German	N/A	ISO8859-1	IBM-500
Latin-2	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene	IBM-852	ISO88859-2	IBM-870
Cyrillic	Bulgarian, Macedonian, Serbian Cyrillic, Russian	IBM-855	ISO8859-5	IBM-880 IBM-1025
Cyrillic	Russian	IBM-866	ISO8859-5	IBM-1025
Hebrew	Hebrew	IBM-856 IBM-862	ISO8859-8	IBM-424 IBM-803
Turkish	Turkish	IBM-857	ISO8859-9	IBM-1026
Arabic	Arabic	IBM-864 IBM-1046	ISO8859-6	IBM-420
Greek	Greek	IBM-869	ISO8859-7	IBM-875

Code Set Compatibility				
Character Set	**Languages**	**PC**	**ISO**	**EBCDIC**
Greek	Greek	IBM-869	ISO8859-7	IBM-875
Baltic	Lithuanian, Latvian, Estonian	IBM-921 IBM-922		IBM-1112 IBM-1122

Note: A character that exists in the source code set but does not exist in the target code set is converted to a converter-defined substitute character.

Files

The following table describes the **inconvTable** converters found in the **/usr/lib/nls/loc/iconvTable** directory:

iconvTable Converters		
Converter Table	**Description**	**Language**
IBM-037_IBM-850	IBM-037 to IBM-850	U.S. English, Portuguese, Canadian-French
IBM-273_IBM-850	IBM-273 to IBM-850	German
IBM-277_IBM-850	IBM-277 to IBM-850	Danish, Norwegian
IBM-278_IBM-850	IBM-278 to IBM-850	Finnish, Swedish
IBM-280_IBM-850	IBM-280 to IBM-850	Italian
IBM-281_IBM-850	IBM-281 to IBM-850	Japanese-Latin
IBM-284_IBM-850	IBM-284 to IBM-850	Spanish
IBM-285_IBM-850	IBM-285 to IBM-850	U.K. English
IBM-297_IBM-850	IBM-297 to IBM-850	French
IBM-420_IBM_1046	IBM-420 to IBM-1046	Arabic
IBM-424_IBM-856	IBM-424 to IBM-856	Hebrew
IBM-424_IBM-862	IBM-424 to IBM-862	Hebrew
IBM-500_IBM-850	IBM-500 to IBM-850	Belgian, Swiss German
IBM-803_IBM-856	IBM-803 to IBM-856	Hebrew
IBM-803_IBM-862	IBM-803 to IBM-862	Hebrew
IBM-850_IBM-037	IBM-850 to IBM-037	U.S. English, Portuguese, Canadian-French
IBM-850_IBM-273	IBM-850 to IBM-273	German
IBM-850_IBM-277	IBM-850 to IBM-277	Danish, Norwegian
IBM-850_IBM-278	IBM-850 to IBM-278	Finnish, Swedish
IBM-850_IBM-280	IBM-850 to IBM-280	Italian
IBM-850_IBM-281	IBM-850 to IBM-281	Japanese-Latin
IBM-850_IBM-284	IBM-850 to IBM-284	Spanish
IBM-850_IBM-285	IBM-850 to IBM-285	U.K. English
IBM-850_IBM-297	IBM-850 to IBM-297	French
IBM-850_IBM-500	IBM-850 to IBM-500	Belgian, Swiss German
IBM-856_IBM-424	IBM-856 to IBM-424	Hebrew
IBM-856_IBM-803	IBM-856 to IBM-803	Hebrew

iconvTable Converters		
Converter Table	**Description**	**Language**
IBM-856_IBM-862	IBM-856 to IBM-862	Hebrew
IBM-862_IBM-424	IBM-862 to IBM-424	Hebrew
IBM-862_IBM-803	IBM-862 to IBM-803	Hebrew
IBM-862_IBM-856	IBM-862 to IBM-856	Hebrew
IBM-864_IBM-1046	IBM-864 to IBM-1046	Arabic
IBM-921_IBM-1112	IBM-921 to IBM-1112	Lithuanian, Latvian
IBM-922_IBM-1122	IBM-922 to IBM-1122	Estonian
IBM-1112_IBM-921	IBM-1121 to IBM-921	Lithuanian, Latvian
IBM-1122_IBM-922	IBM-1122 to IBM-922	Estonian
IBM-1046_IBM-420	IBM-1046 to IBM-420	Arabic
IBM-1046_IBM-864	IBM-1046 to IBM-864	Arabic
IBM-037_ISO8859-1	IBM-037 to ISO8859-1	U.S. English, Portuguese, Canadian French
IBM-273_ISO8859-1	IBM-273 to ISO8859-1	German
IBM-277_ISO8859-1	IBM-277 to ISO8859-1	Danish, Norwegian
IBM-278_ISO8859-1	IBM-278 to ISO8859-1	Finnish, Swedish
IBM-280_ISO8859-1	IBM-280 to ISO8859-1	Italian
IBM-281_ISO8859-1	IBM-281 to ISO8859-1	Japanese-Latin
IBM-284_ISO8859-1	IBM-284 to ISO8859-1	Spanish
IBM-285_ISO8859-1	IBM-285 to ISO8859-1	U.K. English
IBM-297_ISO8859-1	IBM-297 to ISO8859-1	French
IBM-420_ISO8859-6	IBM-420 to ISO8859-6	Arabic
IBM-424_ISO8859-8	IBM-424 to ISO8859-8	Hebrew
IBM-500_ISO8859-1	IBM-500 to ISO8859-1	Belgian, Swiss German
IBM-803_ISO8859-8	IBM-803 to ISO8859-8	Hebrew
IBM-852_ISO8859-2	IBM-852 to ISO8859-2	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene
IBM-855_ISO8859-5	IBM-855 to ISO8859-5	Bulgarian, Macedonian, Serbian Cyrillic, Russian
IBM-866_ISO8859-5	IBM-866 to ISO8859-5	Russian
IBM-869_ISO8859-7	IBM-869 to ISO8859-7	Greek
IBM-875_ISO8859-7	IBM-875 to ISO8859-7	Greek
IBM-870_ISO8859-2	IBM-870 to ISO8859-2	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian, Slovak, Slovene
IBM-880_ISO8859-5	IBM-880 to ISO8859-5	Bulgarian, Macedonian, Serbian Cyrillic, Russian
IBM-1025_ISO8859-5	IBM-1025 to ISO8859-5	Bulgarian, Macedonian, Serbian Cyrillic, Russian
IBM-857_ISO8859-9	IBM-857 to ISO8859-9	Turkish
IBM-1026_ISO8859-9	IBM-1026 to ISO8859-9	Turkish

iconvTable Converters		
Converter Table	**Description**	**Language**
IBM-850_ISO8859-1	IBM-850 to ISO8859-1	Latin
IBM-856_ISO8859-8	IBM-856 to ISO8859-8	Hebrew
IBM-862_ISO8859-8	IBM-862 to ISO8859-8	Hebrew
IBM-864_ISO8859-6	IBM-864 to ISO8859-6	Arabic
IBM-1046_ISO8859-6	IBM-1046 to ISO8859-6	Arabic
ISO8859-1_IBM-850	ISO8859-1 to IBM-850	Latin
ISO8859-6_IBM-864	ISO8859-6 to IBM-864	Arabic
ISO8859-6_IBM-1046	ISO8859-6 to IBM-1046	Arabic
ISO8859-8_IBM-856	ISO8859-8 to IBM-856	Hebrew
ISO8859-8_IBM-862	ISO8859-8 to IBM-862	Hebrew
ISO8859-1_IBM-037	ISO8859-1 to IBM-037	U.S. English, Portuguese, Canadian French
ISO8859-1_IBM-273	ISO8859-1 to IBM-273	German
ISO8859-1_IBM-277	ISO8859-1 to IBM-277	Danish, Norwegian
ISO8859-1_IBM-278	ISO8859-1 to IBM-278	Finnish, Swedish
ISO8859-1_IBM-280	ISO8859-1 to IBM-280	Italian
ISO8859-1_IBM-281	ISO8859-1 to IBM-281	Japanese-Latin
ISO8859-1_IBM-284	ISO8859-1 to IBM-284	Spanish
ISO8859-1_IBM-285	ISO8859-1 to IBM-285	U.K. English
ISO8859-1_IBM-297	ISO8859-1 to IBM-297	French
ISO8859-1_IBM-500	ISO8859-1 to IBM-500	Belgian, Swiss German
ISO8859-2_IBM-852	ISO8859-2 to IBM-852	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene
ISO8859-2_IBM-870	ISO8859-2 to IBM-870	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene
ISO8859-5_IBM-855	ISO8859-5 to IBM-855	Bulgarian, Macedonian, Serbian Cyrillic, Russian
ISO8859-5_IBM-880	ISO8859-5 to IBM-880	Bulgarian, Macedonian, Serbian Cyrillic, Russian
ISO8859-5_IBM-1025	ISO8859-5 to IBM-1025	Bulgarian, Macedonian, Serbian Cyrillic, Russian
ISO8859-6_IBM-420	ISO8859-6 to IBM-420	Arabic
ISO8859-5_IBM-866	ISO8859-5 to IBM-866	Russian
ISO8859-7_IBM-869	ISO8859-7 to IBM-869	Greek
ISO8859-7_IBM-875	ISO8859-7 to IBM-875	Greek
ISO8859-8_IBM-424	ISO8859-8 to IBM-424	Hebrew
ISO8859-8_IBM-803	ISO8859-8 to IBM-803	Hebrew
ISO8859-9_IBM-857	ISO8859-9 to IBM-857	Turkish
ISO8859-9_IBM-1026	ISO8859-9 to IBM-1026	Turkish

Multibyte Code Set Converters

Multibyte code-set converters convert characters among the following code sets:

- PC multibyte code sets
- EUC multibyte code sets (ISO-based)
- EBCDIC multibyte code sets

The following table lists code set names that are compatible. Each line defines to/from strings that may be used when requesting a converter.

Code Set Compatibility			
Language	**PC**	**ISO**	**EBCDIC**
Japanese	IBM-932	IBM-eucJP	IBM-930, IBM-939
Japanese (MS compatible)	IBM-943	IBM-eucJP	IBM-930, IBM-939
Korean	IBM-934	IBM-eucKR	IBM-933
Traditional Chinese	IBM-938, big-5	IBM-eucTW	IBM-937
Simplified Chinese	IBM-1381	IBM-eucCN	IBM-935

1. Conversions between Simplified and Traditional Chinese are provided (IBM-eucTW <—> IBM-eucCN and big5 <—> IBM-eucCN).
2. UTF-8 is an additional code set. See "UTF-8 Interchange Converters" on page 108 for more information.

Files

The following list describes the Multibyte Code Set converters that are found in the **/usr/lib/nls/loc/iconv** directory.

Converter	**Description**
IBM-eucJP_IBM-932	IBM-eucJP to IBM-932
IBM-eucJP_IBM-943	IBM-eucJP to IBM-943
IBM-eucJP_IBM-930	IBM-eucJP to IBM-930
IBM-eucCN_IBM-936(PC5550)	IBM-eucCN to IBM-936(PC5550)
IBM-eucCN_IBM-935	IBM-eucCN to IBM-935
IBM-eucJP_IBM-939	IBM-eucJP to IBM-939
IBM-eucCN_IBM-1381	IBM-eucCN to IBM-1381
IBM-943_IBM-932	IBM-943 to IBM-932
IBM-932_IBM-943	IBM-932 to IBM-943
IBM-930_IBM-932	IBM-930 to IBM-932
IBM-930_IBM-943	IBM-930 to IBM-943
IBM-930_IBM-eucJP	IBM-930 to IBM-eucJP
IBM-932_IBM-eucJP	IBM-932 to IBM-eucJP
IBM-932_IBM-930	IBM-932 to IBM-930
IBM-943_IBM-eucJP	IBM-943 to IBM-eucJP
IBM-943_IBM-930	IBM-943 to IBM-930
IBM-936(PC5550)_IBM-935	IBM-936(PC5550) to IBM-935
IBM-936_IBM-935	IBM-936 to IBM-935

Converter	Description
IBM-932_IBM-939	IBM-932 to IBM-939
IBM-939_IBM-932	IBM-939 to IBM-932
IBM-943_IBM-939	IBM-943 to IBM-939
IBM-939_IBM-943	IBM-939 to IBM-943
IBM-935_IBM-936(PC5550)	IBM-935 to IBM-936(PC5550)
IBM-935_IBM-936	IBM-935 to IBM-936
IBM-1381_IBM-935	IBM-1381 to IBM-935
IBM-935_IBM-1381	IBM-935 to IBM-1381
IBM-935_IBM-eucCN	IBM-935 to IBM-eucCN
IBM-936(PC5550)_IBM-eucCN	IBM-936(PC5550) to IBM-eucCN
IBM-eucTW_IBM-eucCN	IBM-eucTW to IBM-eucCN
big5_IBM-eucCN	big5 to IBM-eucCN
IBM-1381_IBM-eucCN	IBM-1381 to IBM-eucCN
IBM-939_IBM-eucJP	IBM-939 to IBM-eucJP
IBM-eucKR_IBM-934	IBM-eucKR to IBM-934
IBM-934_IBM-eucKR	IBM-934 to IBM-eucKR
IBM-eucKR_IBM-933	IBM-eucKR to IBM-933
IBM-933_IBM-eucKR	IBM-933 to IBM-eucKR
IBM-eucTW_IBM-937	IBM-eucTW to IBM-937
IBM-938_IBM-937	IBM-938 to IBM-937
big-5_IBM-937	big-5 to IBM-937
IBM-eucCN_IBM-eucTW	IBM-eucCN to IBM-eucTW
IBM-937_IBM-eucTW	IBM-937 to IBM-eucTW
IBM-937_IBM-938	IBM-937 to IBM-938
IBM-eucTW_IBM-938	IBM_eucTW to IBM_938
IBM-eucCN_big5	IBM-eucCN to big5
IBM-eucTW_big-5	IBM_eucTW to big-5
IBM-937_big-5	IBM-937 to big-5
CNS11643.1992-3_IBM-eucTW	CNS11643.1992-3 to IBM_eucTW
CNS11643.1992-3-GL_IBM-eucTW	CNS11643.1992-3-GL to IBM_eucTW
CNS11643.1992-3-GR_IBM-eucTW	CNS11643.1992-3-GR to IBM_eucTW
CNS11643.1992-4_IBM-eucTW	CNS11643.1992-4 to IBM_eucTW
CNS11643.1992-4-GL_IBM-eucTW	CNS11643.1992-4-GL to IBM_eucTW
CNS11643.1992-4-GR_IBM-eucTW	CNS11643.1992-4-GR to IBM_eucTW
IBM-eucTW_CNS11643.1992-3	IBM_eucTW to CNS11643.1992-3
IBM-eucTW_CNS11643.1992-3-GL	IBM_eucTW to CNS11643.1992-3-GL
IBM-eucTW_CNS11643.1992-3-GR	IBM_eucTW to CNS11643.1992-3-GR
IBM-eucTW_CNS11643.1992-4	IBM_eucTW to CNS11643.1992-4
IBM-eucTW_CNS11643.1992-4-GL	IBM_eucTW to CNS11643.1992-4-GL
IBM-eucTW_CNS11643.1992-4-GR	IBM_eucTW to CNS11643.1992-4-GR
IBM-eucCN_GB2312.1980-1	IBM-eucCN to GB2312.1980-1

Converter	Description
IBM-eucCN_GB2312.1980-1-GL	IBM-eucCN to GB2312.1980-1-GL
IBM-eucCN_GB2312.1980-1-GR	IBM-eucCN to GB2312.1980-1-GR
IBM-937_csic	IBM-937 to csic
csic_IBM-937	csic to IBM-937
IBM-938_csic	IBM-938 to csic
csic_IBM-938	csic to IBM-938
IBM-eucTW_ccdc	IBM-eucTW to ccdc
ccdc_IBM-eucTW	ccdc to IBM-eucTW
IBM-eucTW_cns	IBM-eucTW to cns
cns_IBM-eucTW	cnd to IBM-eucTW
IBM-eucTW_csic	IBM-eucTW to csic
csic_IBM-eucTW	csic to IBM-eucTW
IBM-eucTW_sops	IBM-ecuTW to sops
sops_IBM-eucTW	sops to IBM-eucTW
IBM-eucTW_tca	IBM-eucTW to tca
tca_IBM-eucTW	tca to IBM-eucTW
big5_cns	big5 to cns
cns_big5	cns to big5
big5_csic	big5 to csic
csic_big5	csic to big5
big5_ttc	big5 to ttc
ttc_big5	ttc to big5
big5_ttcmin	big5 to ttcmin
ttcmin_big5	ttcmin to big5
big5_unicode	big5 to unicode
unicode_big5	unicode to big5
big5_wang	big5 to wang
wang_big5	wang to big5
ccdc_csic	ccdc to csic
csic_ccdc	csic to_ccdc
csic_sops	csic to sops
sops_csic	sops to csic
CNS11643.1986-1_big5	CNS11643.1986-1 to big5
big5_CNS11643.1986-1	big5 to CNS11643.1986-1
CNS11643.1986-1-GR_big5	CNS11643.1986-1-GR to big5
big5_CNS11643.1986-1-GR	big5 to CNS11643.1986-1-GR
CNS11643.1986-2_big5	CNS11643.1986-2 to big5
big5_CNS11643.1986-2	big5 to CNS11643.1986-2
CNS11643.1986-2-GR_big5	CNS11643.1986-2-GR to big5
big5_CNS11643.1986-2-GR	big5 to CNS11643.1986-2-GR
CNS11643.CT-GR_big5	CNS11643.CT-GR to big5

Converter	Description
big5_CNS11643.CT-GR	big5 to CNS11643.CT-GR
IBM-sbdTW-GR_big5	IBM-sbdTW-GR to big5
big5_IBM-sbdTW-GR	big5 to IBM-sbdTW-GR
IBM-sbdTW.CT-GR_big5	IBM-sbdTW.CT-GR to big5
big5_IBM-sbdTW.CT-GR	big5 to IBM-sbdTW.CT-GR
IBM-sbdTW_big5	IBM-sbdTW to big5
big5_IBM-sbdTW	big5 to IBM-sbdTW
IBM-udcTW-GR_big5	IBM-udcTW-GR to big5
big5_IBM-udcTW-GR	big5 to IBM-udcTW-GR
IBM-udcTW.CT-GR_big5	IBM-udcTW.CT-GR to big5
big5_IBM-udcTW.CT-GR	big5 to IBM-udcTW.CT-GR
ISO8859-1_big5	ISO8859 to big5
big5_ISO8859-1	big5 to ISO8859
IBM-sbdTW_big5	IBM-sbdTW to big5
big5_IBM-sbdTW	big5 to IBM-sbdTW
big5_ASCII-GR	big5 to ASCII-GR
ASCII-GR_big5	ASCII-GR to big5
GBK_big5	GBK to big5
big5_GBK	big5 to GBK
GBK_IBM-eucTW	GBK to IBM-eucTW
IBM-eucTW_GBK	IBM-eucTW to GBK
CNS11643.1986-1_GBK	CNS11643.1986-1 to GBK
GBK_CNS11643.1986-1	GBK to CNS11643.1986-1
CNS11643.1986-2_GBK	CNS11643.1986-2 to GBK
GBK_CNS11643.1986-2	GBK to CNS11643.1986-2
CNS11643.1986-1-GR_GBK	CNS11643.1986-1-GR to GBK
GBK_CNS11643.1986-1-GR	GBK to CNS11643.1986-1-GR
CNS11643.1986-2-GR_GBK	CNS11643.1986-2-GR to GBK
GBK_CNS11643.1986-2-GR	GBK to CNS11643.1986-2-GR
CNS11643.1986-1-GL_GBK	CNS11643.1986-1-GL to GBK
GBK_CNS11643.1986-1-GL	GBK to CNS11643.1986-1-GL
CNS11643.1986-2-GL_GBK	CNS11643.1986-2-GL to GBK
GBK_CNS11643.1986-2-GL	GBK to CNS11643.1986-2-GL
CNS11643.CT-GR_GBK	CNS11643.CT-GR to GBK
GBK_CNS11643.CT-GR	GBK to CNS11643.CT-GR
GB2312.1980.CT-GR_GBK	GB2312.1980.CT-GR to GBK
GBK_GB2312.1980.CT-GR	GBK to GB2312.1980.CT-GR
GB2312.1980-0_GBK	GBK2312.1980-0 to GBK
GBK_GB2312.1980-0	GBK to GBK2312.1980-0
GB2312.1980-0-GR_GBK	GB2312.1980-0-GR to GBK
GBK_GB2312.1980-0-GR	GBK to GB2312.1980-0-GR

Converter	Description
GB2312.1980-0-GL_GBK	GB2312.1980-0-GL to GBK
GBK_GB2312.1980-0-GL	GBK to GB2312.1980-0-GL
ASCII-GR_GBK	ASCII-GR to GBK
GBK_ASCII-GR	GBK to ASCII-GR
ISO8859-1_GBK	ISO8859-1 to GBK
GBK_ISO8859-1	GBK to ISO8859-1
IBM-eucCN_GBK	IBM-eucCN to GBK
GBK_IBM-eucCN	GBK to IBM-eucCN

Interchange Converters—7-bit

This converter provides conversion between internal code and 7-bit standard interchange formats (fold7). The fold7 name identifies encodings that can be used to pass text data through 7-bit mail protocols. The encodings are based on ISO2022. For more information about fold7, see "Understanding libiconv" on page 84.

The fold7 converters convert characters from a code set to a canonical 7-bit encoding that identifies each character. This type of conversion is useful in networks where clients communicate with different code sets but use the same character sets. For example:

IBM-850 <—> ISO8859-1	Common Latin characters
IBM-932 <—>IBM-eucJP	Common Japanese characters

The following escape sequences designate standard code sets:

Escape Sequence	Standard Code Set
01/11 02/04 04/00	GL JIS X0208.1978-0.
01/11 02/04 02/08 04/01	GL left half of GB2312.1980-0.
01/11 02/08 04/02	GL 7-bit ASCII or left half of ISO8859-1.
01/11 02/14 04/01	GL right half of ISO8859-1.
01/11 02/14 04/02	GL right half of ISO8859-2.
01/11 02/14 04/03	GL right half of ISO8859-3.
01/11 02/14 04/04	GL right half of ISO8859-4.
01/11 02/14 04/06	GL right half of ISO8859-7.
01/11 02/14 04/07	GL right half of ISO8859-6.
01/11 02/14 04/08	GL right half of ISO8859-8.
01/11 02/14 04/12	GL right half of ISO8859-5.
01/11 02/14 04/13	GL right half of ISO8859-9.
01/11 02/08 04/09	GL right half of JIS X0201.1976-0.
01/11 02/08 04/10	GL left half of JIS X0201.1976.
01/11 02/04 04/02	GL JIS X0208.1983-0.
01/11 02/04 02/08 04/02	GL JIS X0208.1983-0.
01/11 02/04 02/08 04/00	GL JISX0208.1978-0.
01/11 02/05 02/15 03/01 M L 06/09 06/02 06/13 02/13 03/08 03/05 03/00 00/02	GL right half of IBM-850 unique characters. Characters common to ISO8859-1 do not use this escape sequence.

Escape Sequence	Standard Code Set
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02	GL Japanese) IBM-udcJP) user-definable characters.
01/11 02/04 02/08 04/03	GL KSC5601-1987.
01/11 02/04 02/09 03/00	GL CNS11643-1986-1.
01/11 02/04 02/10 03/01	GL CNS11643-1986-2.
01/11 02/05 02/15 03/00 M L 05/05 05/04 04/06 02/13 03/07 00/02	UCS-2 encoded as base64; used only for those characters not encoded by any of the other 7-bit escape sequences listed above.

When converting from a code set to fold7, the escape sequence used to designate the code set is chosen according to the order listed. For example, the JISX0208.1983-0 characters use **01/11 01/04 04/02** as the designation.

Files

The following list describes the fold7 converters that are found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
fold7_IBM-850	Interchange format to IBM-850
fold7_IBM-921	Interchange format to IBM-921
fold7_IBM-922	Interchange format to IBM-922
fold7_IBM-932	Interchange format to IBM-932
fold7_IBM-943	Interchange format to IBM-943
fold7_IBM_1124	Interchange format to IBM-1124
fold7_IBM_1129	Interchange format to IBM-1129
fold7_IBM_eucCN	Interchange format to IBM-eucCN
fold7_IBM-eucJP	Interchange format to IBM-eucJP
fold7_IBM-eucKR	Interchange format to IBM-eucKR
fold7_IBM-eucTW	Interchange format to IBM-eucTW
fold7_ISO8859-1	Interchange format to ISO8859-1
fold7_ISO8859-2	Interchange format to ISO8859-2
fold7_ISO8859-3	Interchange format to ISO8859-3
fold7_ISO8859-4	Interchange format to ISO8859-4
fold7_ISO8859-5	Interchange format to ISO8859-5
fold7_ISO8859-6	Interchange format to ISO8859-6
fold7_ISO8859-7	Interchange format to ISO8859-7
fold7_ISO8859-8	Interchange format to ISO8859-8
fold7_ISO8859-9	Interchange format to ISO8859-9
fold7_TIS-620	Interchange format to TIS-620
fold7_UTF-8	Interchange format to UTF-8
fold7_big5	Interchange format to big5
fold7_GBK	Interchange format to GBK
IBM-921_fold7	IBM-921 to interchange format
IBM-922_fold7	IBM-922 to interchange format

Converter	Description
IBM-850_fold7	IBM-850 to interchange format
IBM-932_fold7	IBM-932 to interchange format
IBM-943_fold7	IBM-943 to interchange format
IBM-1124_fold7	IBM-1124 to interchange format
IBM-1129_fold7	IBM-1129 to interchange format
IBM-eucCN_fold7	IBM-eucCN to interchange format
IBM-eucJP_fold7	IBM-eucJP to interchange format
IBM-eucKR_fold7	IBM-eucKR to interchange format
IBM-eucTW_fold7	IBM-eucTW to interchange format
ISO8859-1_fold7	ISO8859-1 to interchange format
ISO8859-2_fold7	ISO8859-2 to interchange format
ISO8859-3_fold7	ISO8859-3 to interchange format
ISO8859-4_fold7	ISO8859-4 to interchange format
ISO8859-5_fold7	ISO8859-5 to interchange format
ISO8859-6_fold7	ISO8859-6 to interchange format
ISO8859-7_fold7	ISO8859-7 to interchange format
ISO8859-8_fold7	ISO8859-8 to interchange format
ISO8859-9_fold7	ISO8859-9 to interchange format
TIS-620_fold7	TIS-620 to interchange format
UTF-8_fold7	UTF-8 to interchange format
big5_fold7	big5 to interchange format
GBK_fold7	GBK to interchange format

Interchange Converters—8-bit

This converter provides conversions between internal code and 8-bit standard interchange formats (fold8). The fold8 name identifies encodings that can be used to pass text data through 8-bit mail protocols. The encodings are based on ISO2022. For more information about fold8, see “Understanding libiconv” on page 84.

The fold8 converters convert characters from a specific code set encoding to a canonical 8-bit encoding that identifies each character. This type of conversion is useful in networks where clients communicate with different code sets but use the same character sets. For example:

IBM-850 <—> **ISO8859-1**	Common Latin characters
IBM-932 <—>**IBM-eucJP**	Common Japanese characters

The following escape sequences designate standard code sets.

Escape Sequence	Standard Code Set
01/11 02/04 02/09 04/01	GR right half of GB2312.1980-0.
01/11 02/13 04/01	GR right half of ISO8859-1.
01/11 02/13 04/02	GR right half of ISO8859-2.
01/11 02/13 04/03	GR right half of ISO8859-3.
01/11 02/13 04/04	GR right half of ISO8859-4.

Escape Sequence	Standard Code Set
01/11 02/13 04/06	GR right half of ISO8859-7.
01/11 02/13 04/07	GR right half of ISO8859-6.
01/11 02/13 04/08	GR right half of ISO8859-8.
01/11 02/13 04/13	GR right half of ISO8859-5.
01/11 02/13 04/13	GR right half of ISO8859-9.
01/11 02/09 04/09	GR right half of JIS X0201.1976-1.
01/11 02/04 02/09 04/02	GR JIS X0208.1983-1.
01/11 02/04 02/09 04/00	GR JISX0208.1978-1.
01/11 02/09 04/02	GR 7-bit ASCII or left half of ISO8859-1.
01/11 02/05 02/15 03/01 M L 04/09 04/02 04/13 02/13 03/08 03/05 03/00 00/02	GR right half of IBM-850 unique characters. Characters common to ISO8859-1 should not use this escape sequence.
01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02	GR right half of Japanese user-definable characters.
01/11 02/08 04/02	GL 7-bit ASCII or left half of ISO8859-1.
01/11 02/14 04/01	GL right half of ISO8859-1.
01/11 02/14 04/02	GL right half of ISO8859-2.
01/11 02/14 04/03	GL right half of ISO8859-3.
01/11 02/14 04/04	GL right half of ISO8859-4.
01/11 02/14 04/06	GL right half of ISO8859-7.
01/11 02/14 04/07	GL right half of ISO8859-6.
01/11 02/14 04/08	GL right half of ISO8859-8.
01/11 02/14 04/12	GL right half of ISO8859-5.
01/11 02/14 04/13	GL right half of ISO8859-9.
01/11 02/08 04/09	GL right half of JIS X0201.1976-0.
01/11 02/08 04/10	GL left half of JIS X0201.1976.
01/11 02/04 02/08 04/02	GL JIS X0208.1983-0.
01/11 02/04 04/02	GL JIS X0208.1983-0.
01/11 02/04 04/00	GL JIS X0208.1978-0.
01/11 02/05 02/15 03/01 M L 06/09 06/02 06/13 02/13 03/08 03/05 03/00 00/02	GL right half of IBM-850 unique characters. Characters common to ISO8859-1 do not use this escape sequence.
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02	GL Japanese (IBM-udcJP) user-definable characters.
01/11 02/04 02/09 04/03	GR KSC5601-1987.
01/11 02/04 02/09 03/00	GR CNS11643-1986-1.
01/11 02/04 02/10 03/01	GR CNS11643-1986-2.
01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/05 06/04 06/03 05/05 05/08 00/02	GR right half of Traditional Chinese user-definable characters.
01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/03 06/02 06/04 05/05 05/08 00/02	GR right half of IBM-850 unique symbols.
01/11 02/04 02/08 04/03	GL KSC5601-1987.
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 05/05 05/08 00/02	GL Traditional Chinese (IBM-udcTW) user-definable characters.

Escape Sequence	Standard Code Set
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/03 06/02 06/04 05/05 05/08 00/02	GL Traditional Chinese IBM-850 unique symbols (IBM-shdTW) user-definable characters.
01/11 02/05 02/15 03/00 M L 05/05 05/04 04/06 02/13 03/08 00/02	UCS-2 encoded as UTF-8; used only for those characters not encoded by any of the above escape sequences listed above.

When converting from a code set to fold8, the escape sequence used to designate the code set is chosen according to the order listed. For example, the JISX0208.1983-0 characters use **01/11 02/04 02/08 04/02** as the designation.

Files

The following list describes the fold8 converters found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
fold8_IBM-850	Interchange format to IBM-850
fold8_IBM-921	Interchange format to IBM-921
fold8_IBM-922	Interchange format to IBM-922
fold8_IBM-932	Interchange format to IBM-932
fold8_IBM-943	Interchange format to IBM-943
fold8_IBM-1124	Interchange format to IBM-1124
fold8_IBM-1129	Interchange format to IBM-1129
fold8_IBM-eucCN	Interchange format to IBM-eucCN
fold8_IBM-eucJP	Interchange format to IBM-eucJP
fold8_IBM-eucKR	Interchange format to IBM-eucKR
fold8_IBM-eucTW	Interchange format to IBM-eucTW
fold8_IBM-eucCN	Interchange fromat to IBM-eucCN
fold8_ISO8859-1	Interchange format to ISO8859-1
fold8_ISO8859-2	Interchange format to ISO8859-2
fold8_ISO8859-3	Interchange format to ISO8859-3
fold8_ISO8859-4	Interchange format to ISO8859-4
fold8_ISO8859-5	Interchange format to ISO8859-5
fold8_ISO8859-6	Interchange format to ISO8859-6
fold8_ISO8859-7	Interchange format to ISO8859-7
fold8_ISO8859-8	Interchange format to ISO8859-8
fold8_ISO8859-9	Interchange format to ISO8859-9
fold8_TIS-620	Interchange format to TIS-620
fold8_UTF-8	Interchange format to UTF-8
fold8_big5	Interchange format to big5
fold8_GBK	Interchange format to GBK
IBM-921_fold8	IBM-921 to interchange format
IBM-922_fold8	IBM-922 to interchange format
IBM-850_fold8	IBM-850 to interchange format
IBM-932_fold8	IBM-932 to interchange format

Converter	Description
IBM-943_fold8	IBM-943 to interchange format
IBM-1124_fold8	IBM-1124 to interchange format
IBM-1129_fold8	IBM-1129 to interchange format
IBM-eucCN_fold8	IBM-eucCN to interchange format
IBM-eucJP_fold8	IBM-eucJP to interchange format
IBM-eucKR_fold8	IBM-eucKR to interchange format
IBM-eucTW_fold8	IBM-eucTW to interchange format
IBM-eucCN_fold8	IBM-eucCN to interchange format
ISO8859-1_fold8	ISO8859-1 to interchange format
ISO8859-2_fold8	ISO8859-2 to interchange format
ISO8859-3_fold8	ISO8859-3 to interchange format
ISO8859-4_fold8	ISO8859-4 to interchange format
ISO8859-5_fold8	ISO8859-5 to interchange format
ISO8859-6_fold8	ISO8859-6 to interchange format
ISO8859-7_fold8	ISO8859-7 to interchange format
ISO8859-8_fold8	ISO8859-8 to interchange format
ISO8859-9_fold8	ISO8859-9 to interchange format
TIS-620_fold8	TIS-620 to interchange format
UTF-8_fold8	UTF-8 to interchange format
big5_fold8	big5 to interchange format
GBK_fold8	GBK to interchange format

Interchange Converters—Compound Text

Compound text interchange converters convert between compound text and internal code sets.

Compound text is an interchange encoding defined by the X Consortium. It is used to communicate text between X clients. Compound text is based on ISO2022 and can encode most character sets using standard escape sequences. It also provides extensions for encoding private character sets. The supported code sets provide a converter to and from compound text. The name used to identify the compound text encoding is ct.

The following escape sequences are used to designate standard code sets in the order listed below.

01/11 02/05 02/15 03/01 M L 04/09 04/02 04/13 02/13 03/08 03/05 03/00 00/02
GR right half of IBM-850 unique characters. Characters common to ISO8859-1 should not use this escape sequence.

01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02
GR right half of Japanese user-definable characters.

01/11 02/05 02/15 03/01 M L 06/09 06/02 06/13 02/13 03/08 03/05 03/00 00/02
GL right half of IBM-850 unique characters. Characters common to ISO8859-1 do not use this escape sequence.

01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02
GL Japanese (IBM-udcJP) user-definable characters.

Files

The following list describes the compound text converters that are found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
ct_IBM-850	Interchange format to IBM-850
ct_IBM-921	Interchange format to IBM-921
ct_IBM-922	Interchange format to IBM-922
ct_IBM-932	Interchange format to IBM-932
ct_IBM-943	Interchange format to IBM-943
ct_IBM-1124	Interchange format to IBM-1124
ct_IBM-1129	Interchange format to IBM-1129
ct_IBM-eucCN	Interchange format to IBM-eucCN
ct_IBM-eucJP	Interchange format to IBM-eucJP
ct_IBM-eucKR	Interchange format to IBM-eucKR
ct_IBM-eucTW	Interchange format to IBM-eucTW
ct_ISO8859-1	Interchange format to ISO8859-1
ct_ISO8859-2	Interchange format to ISO8859-2
ct_ISO8859-3	Interchange format to ISO8859-3
ct_ISO8859-4	Interchange format to ISO8859-4
ct_ISO8859-5	Interchange format to ISO8859-5
ct_ISO8859-6	Interchange format to ISO8859-6
ct_ISO8859-7	Interchange format to ISO8859-7
ct_ISO8859-8	Interchange format to ISO8859-8
ct_ISO8859-9	Interchange format to ISO8859-9
ct_TIS-620	Interchange format to TIS-620
ct_big5	Interchange format to big5
ct_GBK	Interchange format to GBK
IBM-850_ct	IBM-850 to interchange format
IBM-921_ct	IBM-921 to interchange format
IBM-922_ct	IBM-922 to interchange format
IBM-932_ct	IBM-932 to interchange format
IBM-943_ct	IBM-943 to interchange format
IBM-1124_ct	IBM-1124 to interchange format
IBM-1129_ct	IBM-1129 to interchange format
IBM-eucCN_ct	IBM-eucCN to interchange format
IBM-eucJP_ct	IBM-eucJP to interchange format
IBM-eucKR_ct	IBM-eucKR to interchange format
IBM-eucTW_ct	IBM-eucTW to interchange format
ISO8859-1_ct	ISO8859-1 to interchange format
ISO8859-2_ct	ISO8859-2 to interchange format
ISO8859-3_ct	ISO8859-3 to interchange format
ISO8859-4_ct	ISO8859-4 to interchange format

Converter	Description
ISO8859-5_ct	ISO8859-5 to interchange format
ISO8859-6_ct	ISO8859-6 to interchange format
ISO8859-7_ct	ISO8859-7 to interchange format
ISO8859-8_ct	ISO8859-8 to interchange format
ISO8859-9_ct	ISO8859-9 to interchange format
TIS-620_ct	TIS-620 to interchange format
big5_ct	big5 to interchange format
GBK_ct	GBK to interchange format

Interchange Converters—uucode

This converter provides the same mapping as the **uuencode** and **uudecode** commands.

During conversion from uucode, 62 bytes at a time (including a new-line character trailing the record) are converted, and generating 45 bytes in *outbuf*.

Files

The following list describes the uucode converters found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
IBM-850_uucode	IBM-850 to uucode
IBM-921_uucode	IBM-921 to uucode
IBM-922_uucode	IBM-922 to uucode
IBM-932_uucode	IBM-932 to uucode
IBM-943_uucode	IBM-943 to uucode
IBM-1124_uucode	IBM-1124 to uucode
IBM-1129_uucode	IBM-1129 to uucode
IBM-eucJP_uucode	IBM-eucJP to uucode
IBM-eucKR_uucode	IBM-eucKR to uucode
IBM-eucTW_uucode	IBM-eucTW to uucode
IBM-eucCN_uucode	IBM-eucCN to uucode
ISO8859-1_uucode	ISO8859-1 to uucode
ISO8859-2_uucode	ISO8859-2 to uucode
ISO8859-3_uucode	ISO8859-3 to uucode
ISO8859-4_uucode	ISO8859-4 to uucode
ISO8859-5_uucode	ISO8859-5 to uucode
ISO8859-6_uucode	ISO8859-6 to uucode
ISO8859-7_uucode	ISO8859-7 to uucode
ISO8859-8_uucode	ISO8859-8 to uucode
ISO8859-9_uucode	ISO8859-9 to uucode
TIS-620_uucode	TIS-620 to uucode
big5_uucode	big5 to uucode
GBK_uucode	GBK to uucode
uucode_IBM-850	uucode to IBM-850

Converter	Description
uucode_IBM-921	uucode to IBM-921
uucode_IBM-922	uucode to IBM-922
uucode_IBM-932	uucode to IBM-932
uucode_IBM-943	uucode to IBM-943
uucode_IBM-1124	uucode to IBM-1124
uucode_IBM-1129	uucode to IBM-1129
uucode_IBM-eucCN	uucode to IBM-eucCN
uucode_IBM-eucJP	uucode to IBM-eucJP
uucode_IBM-eucKR	uucode to IBM-eucKR
uucode_IBM-eucTW	uucode to IBM-eucTW
uucode_ISO8859-1	uucode to ISO8859-1
uucode_ISO8859-2	uucode to ISO8859-2
uucode_ISO8859-3	uucode to ISO8859-3
uucode_ISO8859-4	uucode to ISO8859-4
uucode_ISO8859-5	uucode to ISO8859-5
uucode_ISO8859-6	uucode to ISO8859-6
uucode_ISO8859-7	uucode to ISO8859-7
uucode_ISO8859-8	uucode to ISO8859-8
uucode_ISO8859-9	uucode to ISO8859-9
uucode_TIS-1124	uucode to TIS-1129
uucode_big5	uucode to big5
uucode_GBK	uucode to GBK

UCS-2 Interchange Converters

UCS-2 uses a universal 16-bit encoding. Conversions for each code set are provided in both directions, between the code set and UCS-2. For more information, see Chapter 4, “Code Sets for National Language Support,” on page 49.

UCS-2 converters are found in **/usr/lib/nls/loc/uconvTable** and **/usr/lib/nls/loc/uconv** directories. The **uconvdef** command is used to generate new converters or to customize existing UCS-2 converters.

Converter	Description
ISO8859-1	UCS-2 <—> ISO Latin-1
ISO8859-2	UCS-2 <—> ISO Latin-2
ISO8859-3	UCS-2 <—> ISO Latin-3
ISO8859-4	UCS-2 <—> ISO Latin-4
ISO8859-5	UCS-2 <—> ISO Cyrillic
ISO8859-6	UCS-2 <—> ISO Arabic
ISO8859-7	UCS-2 <—> ISO Greek
ISO8859-8	UCS-2 <—> ISO Hebrew
ISO8859-9	UCS-2 <—> ISO Turkish
JISX0201.1976-0	UCS-2 <—> Japanese JISX0201-0

Converter	Description
JISX0208.1983-0	UCS-2 <—> Japanese JISX0208-0
CNS11643.1986-1	UCS-2 <—> Chinese CNS11643-1
CNS11643.1986-2	UCS-2 <—> Chinese CNS11643-2
KSC5601.1987-0	UCS-2 <—> Korean KSC5601-0
IBM-eucCN	UCS-2 <—> Simplified Chinese EUC
IBM-udcCN	UCS-2 <—> Simplified Chinese user-defined characters
IBM-sbdCN	UCS-2 <—> Simplified Chinese IBM-specific characters
GB2312.1980-0	UCS-2 <—> Simplified Chinese GB
IBM-1381	UCS-2 <—> Simplified Chinese PC data code
IBM-935	UCS-2 <—> Simplified Chinese EBCDIC
IBM-936	UCS-2 <—> Simplified Chinese PC5550
IBM-eucJP	UCS-2 <—> Japanese EUC
IBM-eucKR	UCS-2 <—> Korean EUC
IBM-eucTW	UCS-2 <—> Traditional Chinese EUC
IBM-udcJP	UCS-2 <—> Japanese user-defined characters
IBM-udcTW	UCS-2 <—> Traditional Chinese user-defined characters
IBM-sbdTW	UCS-2 <—> Traditional Chinese IBM-specific characters
UTF-8	UCS-2 <—> UTF-8
IBM-437	UCS-2 <—> USA PC data code
IBM-850	UCS-2 <—> Latin-1 PC data code
IBM-852	UCS-2 <—> Latin-2 PC data code
IBM-857	UCS-2 <—> Turkish PC data code
IBM-860	UCS-2 <—> Portuguese PC data code
IBM-861	UCS-2 <—> Icelandic PC data code
IBM-863	UCS-2 <—> French Canadian PC data code
IBM-865	UCS-2 <—> Nordic PC data code
IBM-869	UCS-2 <—> Greek PC data code
IBM-921	UCS-2 <—> Baltic Multilingual data code
IBM-922	UCS-2 <—> Estonian data code
IBM-932	UCS-2 <—> Japanese PC data code
IBM-943	UCS-2 <—> Japanese PC data code
IBM-934	UCS-2 <—> Korea PC data code
IBM-936	UCS-2 <—> People’s Republic of China PC data code
IBM-938	UCS-2 <—> Taiwanese PC data code
IBM-942	UCS-2 <—> Extended Japanese PC data code
IBM-944	UCS-2 <—> Korean PC data code
IBM-946	UCS-2 <—> People’s Republic of China SAA data code
IBM-948	UCS-2 <—> Traditional Chinese PC data code
IBM-1124	UCS-2 <—> Ukranian PC data code
IBM-1129	UCS-2 <—> Vietnamese PC data code
TIS-620	UCS-2 <—> Thailand PC data code

Converter	Description
IBM-037	UCS-2 <—> USA, Canada EBCDIC
IBM-273	UCS-2 <—> Germany, Austria EBCDIC
IBM-277	UCS-2 <—> Denmark, Norway EBCDIC
IBM-278	UCS-2 <—> Finland, Sweden EBCDIC
IBM-280	UCS-2 <—> Italy EBCDIC
IBM-284	UCS-2 <—> Spain, Latin America EBCDIC
IBM-285	UCS-2 <—> United Kingdom EBCDIC
IBM-297	UCS-2 <—> France EBCDIC
IBM-500	UCS-2 <—> International EBCDIC
IBM-875	UCS-2 <—> Greek EBCDIC
IBM-930	UCS-2 <—> Japanese Katakana-Kanji EBCDIC
IBM-933	UCS-2 <—> Korean EBCDIC
IBM-937	UCS-2 <—> Traditional Chinese EBCDIC
IBM-939	UCS-2 <—> Japanese Latin-Kanji EBCDIC
IBM-1026	UCS-2 <—> Turkish EBCDIC
IBM-1112	UCS-2 <—> Baltic Multilingual EBCDIC
IBM-1122	UCS-2 <—> Estonian EBCDIC
IBM-1124	UCS-2 <—> Ukranian EBCDIC
IBM-1129	UCS-2 <—> Vietnamese EBCDIC
TIS-620	UCS-2 <—>Thailand EBCDIC

UTF-8 Interchange Converters

UTF-8 is a universal, multibyte encoding described in the "UCS-2 and UTF-8" on page 80. Conversions for each code set are provided in both directions, between the code set and UTF-8.

UTF-8 conversions are usually done by using the Universal_UCS_Conv and **/usr/lib/nls/loc/uconv/UTF-8** converter. For more information, see "UCS-2 Interchange Converters" on page 106.

Converter	Description
ISO8859-1	UTF-8 <—> ISO Latin-1
ISO8859-2	UTF-8 <—> ISO Latin-2
ISO8859-3	UTF-8 <—> ISO Latin-3
ISO8859-4	UTF-8 <—> ISO Latin-4
ISO8859-5	UTF-8 <—> ISO Cyrillic
ISO8859-6	UTF-8 <—> ISO Arabic
ISO8859-7	UTF-8 <—> ISO Greek
ISO8859-8	UTF-8 <—> ISO Hebrew
ISO8859-9	UTF-8 <—> ISO Turkish
JISX0201.1976-0	UTF-8 <—> Japanese JISX0201-0
JISX0208.1983-0	UTF-8 <—> Japanese JISX0208-0
CNS11643.1986-1	UTF-8 <—> Chinese CNS11643-1
CNS11643.1986-2	UTF-8 <—> Chinese CNS11643-2

Converter	Description
KSC5601.1987-0	UTF-8 <—> Korean KSC5601-0
IBM-eucCN	UTF-8 <—> Simplified Chinese EUC
IBM-eucJP	UTF-8 <—> Japanese EUC
IBM-eucKR	UTF-8 <—> Korean EUC
IBM-eucTW	UTF-8 <—> Traditional Chinese EUC
IBM-udcJP	UTF-8 <—> Japanese user-defined characters
IBM-udcTW	UTF-8 <—> Traditional Chinese user-defined characters
IBM-sbdTW	UTF-8 <—> Traditional Chinese IBM-specific characters
UCS-2	UTF-8 <—> UCS-2
IBM-437	UTF-8 <—> USA PC data code
IBM-850	UTF-8 <—> Latin-1 PC data code
IBM-852	UTF-8 <—> Latin-2 PC data code
IBM-857	UTF-8 <—> Turkish PC data code
IBM-860	UTF-8 <—> Portuguese PC data code
IBM-861	UTF-8 <—> Icelandic PC data code
IBM-863	UTF-8 <—> French Canadian PC data code
IBM-865	UTF-8 <—> Nordic PC data code
IBM-869	UTF-8 <—> Greek PC data code
IBM-921	UTF-8 <—> Baltic Multilingual data code
IBM-922	UTF-8 <—> Estonian data code
IBM-932	UTF-8 <—> Japanese PC data code
IBM-943	UTF-8 <—> Japanese PC data code
IBM-934	UTF-8 <—> Korea PC data code
IBM-935	UTF-8 <—> Simplified Chinese EBCDIC
IBM-936	UTF-8 <—> People's Republic of China PC data code
IBM-938	UTF-8 <—> Taiwanese PC data code
IBM-942	UTF-8 <—> Extended Japanese PC data code
IBM-944	UTF-8 <—> Korean PC data code
IBM-946	UTF-8 <—> People's Republic of China SAA data code
IBM-948	UTF-8 <—> Traditional Chinese PC data code
IBM-1124	UTF-8 <—> Ukrainian PC data code
IBM-1129	UTF-8 <—> Vietnamese PC data code
TIS-620	UTF-8 <—> Thailand PC data code
IBM-037	UTF-8 <—> USA, Canada EBCDIC
IBM-273	UTF-8 <—> Germany, Austria EBCDIC
IBM-277	UTF-8 <—> Denmark, Norway EBCDIC
IBM-278	UTF-8 <—> Finland, Sweden EBCDIC
IBM-280	UTF-8 <—> Italy EBCDIC
IBM-284	UTF-8 <—> Spain, Latin America EBCDIC
IBM-285	UTF-8 <—> United Kingdom EBCDIC
IBM-297	UTF-8 <—> France EBCDIC

Converter	Description
IBM-500	UTF-8 <—> International EBCDIC
IBM-875	UTF-8 <—> Greek EBCDIC
IBM-930	UTF-8 <—> Japanese Katakana-Kanji EBCDIC
IBM-933	UTF-8 <—> Korean EBCDIC
IBM-937	UTF-8 <—> Traditional Chinese EBCDIC
IBM-939	UTF-8 <—> Japanese Latin-Kanji EBCDIC
IBM-1026	UTF-8 <—> Turkish EBCDIC
IBM-1112	UTF-8 <—> Baltic Multilingual EBCDIC
IBM-1122	UTF-8 <—> Estonian EBCDIC
IBM-1124	UTF-8 <—> Ukranian EBCDIC
IBM-1129	UTF-8 <—> Vietnamese EBCDIC
IBM-1381	UTF-8 <—> Simplified Chinese PC data code
GB18030	UTF-8<—> Simplified Chinese
TIS-620	UTF-8 <—> Thailand EBCDIC

Miscellaneous Converters

A set of low-level converters used by the code set and interchange converters is provided. These converters are called *miscellaneous converters*. These low-level converters may be used by some of the interchange converters. However, the use of these converters is discouraged because they are intended for support of other converters.

Files

The following list describes the miscellaneous converters found in the **/usr/lib/nls/loc/iconv** and **/usr/lib/nls/loc/iconvTable** directories:

Converter	Description
IBM-932_JISX0201.1976-0	IBM-932 to JISX0201.1976-0
IBM-932_JISX0208.1983-0	IBM-932 to JISX0208.1983-0
IBM-932_IBM-udcJP	IBM-932 to IBM-udcJP (Japanese user-defined characters)
IBM-943_JISX0201.1976-0	IBM-943 to JISX0201.1976-0
IBM-943_JISX0208.1983-0	IBM-943 to JISX0208.1983-0
IBM-943_IBM-udcJP	IBM-943 to IBM-udcJP (Japanese user-defined characters
IBM-eucJP_JISX0201.1976-0	IBM-eucJP to JISX0201.1976-0
IBM-eucJP_JISX0208.1983-0	IBM-eucJP to JISX0208.1983-0
IBM-eucJP_IBM-udcJP	IBM-eucJP to IBM-udcJP (Japanese user-defined characters)
IBM-eucKR_KSC5601.1987-0	IBM_eucKR to KSC5601.1987-0
IBM-eucTW_CNS11643.1986-1	IBM-eucTW to CNS11643.1986.1
IBM-eucTW_CNS11643.1986-2	IBM-eucTW to CNS11643.1986-2
IBM-eucCN_GB2312.1980-0	IBM-eucCN to GB2312.1980-0

Writing Converters Using the iconv Interface

This section provides information about the **iconv** subroutines and structures in preparation for writing code set converters. Included in this discussion are an overview of the control flow and the order in which the framework operates, details about writing code set converters, and an example including the code, header file, and a makefile. This section applies to the **iconv** framework within AIX.

Under the framework of the **iconv_open**, **iconv** and **iconv_close** subroutines, you can create and use several different types of converters. Applications can call these subroutines to convert characters in one code set into characters in a different code set. The access and use of the **iconv_open**, **iconv** and **iconv_close** subroutines is standardized by X/Open Portability Guide Issue 4.

Code Sets and Converters

Code sets can be classified into two categories: stateful encodings and stateless encodings.

Stateful Code Sets and Converters

The stateful encodings use shift-in and shift-out codes to change state. Shift-out can be used to indicate the start of host double-byte data in a data stream of characters, and shift-in can be used to indicate the end of this double-byte character data. When the double-byte data is off, it signals the start of single-byte character data. An example of such a stateful code set is IBM-930 used mainly on mainframes (hosts).

Converters written to do the conversion of stateful encodings to other code sets tend to be complex because of the extra processing needed.

Stateless Code Sets and Converters

The stateless code sets are those that can be classified as one of the following types:

- Single-byte code sets, such as ISO8859 family (ISO8859-1, ISO8859-2, and so on)
- Multibyte code sets, such as IBM-eucJP (Japanese), IBM-932 (Shift-JIS).

Note that conversions are meaningful only if the code sets represent the same characters.

The simplest types of code-set conversion can be found in single-byte code set converters, such as the converter from ISO8859-1 to IBM-850. These single-byte code set converters are based on simple table-based conversions. The conversion of multibyte character encodings, such as IBM-eucJP to IBM-932, are in general based on an algorithm and not on tables, because the tables can get lengthy.

Overview of iconv Framework Structures

The **iconv** framework consists of the **iconv_open**, **iconv** and **iconv_close** subroutines, and is based on a common core structure that is part of all converters. The core structure is initialized at the load time of the converter object module. After the loading of the converter is complete, the main entry point, which is always the **instantiate** subroutine, is invoked. This initializes the core structure and returns the core converter descriptor. This is further used during the call to the **init** subroutine provided by the converter to allocate the converter-specific structures. This **init** subroutine returns another converter descriptor that has a pointer to the core converter descriptor. The **init** subroutine allocates memory as needed and may invoke other converters if needed. The **init** subroutine is the place for any converter-specific initialization, whereas the **instantiate** subroutine is a generic entry point.

After the converter descriptor for this converter is allocated and initialized, the next step is to provide the actual code needed for the **exec** part of the functionality. If the converter is a table-based converter, the only need is to provide a source file format that conforms to the input needs of the **genxlt** utility, which takes this source table as the input and generates an output file format usable by the **iconv** framework.

iconv.h File and Structures

The **iconv.h** file in **/usr/include** defines the following structures:

```
typedef struct __iconv_rec   iconv_rec, *iconv_t;
struct __iconv_rec   {
    _LC_object_t  hdr;
    iconv_t (*open)(const char *tocode, const char *fromcode);
    size_t (*exec)(iconv_t cd, char **inbuf, size_t *inbytesleft,
            char **outbuf, size_t *outbytesleft);
    void  (*close)(iconv_t cd);
};
```

The common core structure is as follows (**/usr/include/iconv.h**):

```
typedef struct _LC_core_iconv_type      _LC_core_iconv_t;
struct _LC_core_iconv_type {
        _LC_object_t  hdr;
        /* implementation initialization */
        _LC_core_iconv_t        *(*init)();
        size_t  (*exec)();
        void    (*close)();
};
```

Every converter has a static memory area, which contains the **_LC_core_iconv_t** structure. It is initialized in the **instantiate** subroutine provided as part of the converter program.

iconv Control Flow

An application invokes a code set converter by the following call:

```
iconv_open(char *to_codeset, char *from_codeset)
```

The *to* and *from* code sets are used in selecting the converter by way of the search path defined by the **LOCPATH** environment variable. The **iconv_open** subroutine uses the **_lc_load** subroutine to load the object module specified by concatenating the *from* and *to* code set names to the **iconv_open** subroutine.

```
CONVERTER NAME= "from_codeset" + "_" +"to_codeset"
```

If the *from_codeset* is IBM-850 and the *to_codeset* is ISO8859-1, the converter name is IBM-850_ISO8859-1.

After loading the converter, its entry point is invoked by the **_lc_load** loader subroutine. This is the first call to the converter. The **instantiate** subroutine then initializes the **_LC_core_iconv_t** core structure. The **iconv_open** subroutine then calls the **init** subroutine associated with the core structure thus returned. The **init** subroutine allocates the converter-specific descriptor structure and initializes it as needed by the converter. The **iconv_open** subroutine returns this converter-specific structure. However, the return value is typecast to **iconv_t** in the user's application. Thus, the application does not see the whole of the converter-specific structure; it sees only the public **iconv_t** structure. The converter code itself uses the private converter structure. Applications that use **iconv** converters should not change the converter descriptor; the converter descriptor should be used as an opaque structure.

An *entry point* is declared in every converter so that when the converter is opened by a call to the **iconv_open** subroutine, that entry point is automatically invoked. The entry point is the **instantiate** subroutine that should be provided in all converters. The entry point is specified in the makefile as follows:

```
LDENTRY=-einstantiate
```

When the converter is loaded on a call to the **iconv_open** subroutine, the **instantiate** subroutine is invoked. This subroutine initializes a static core conversion descriptor structure **_LC_core_iconv_t cd**.

The core conversion descriptor **cd** contains pointers to the **init**, **_iconv_exec**, and **_iconv_close** subroutines supplied by the specific converter. The **instantiate** subroutine returns the core conversion descriptor to be used later. The **_LC_core_iconv_t** structure is defined in **/usr/include/iconv.h**.

When the **iconv_open** subroutine is called, the following actions occur:

1. The converter is found using the **LOCPATH** environment variable, the converter is loaded, and the **instantiate** subroutine is invoked. On success, it returns the core conversion descriptor. (**_LC_core_iconv_t *cd**). The **instantiate** subroutine provided by the converter is responsible for initializing the header in the core structure.
2. The **iconv_open** subroutine then invokes the **init** subroutine specified in the core conversion descriptor. The **init** subroutine provided by the converter is responsible for allocation of memory needed to hold the converter descriptor needed for this specific converter. For example, the following might be the structure needed by a stateless converter:

```
typedef struct _LC_sample_iconv_rec {
                  LC_core_iconv_t        core;

            } _LC_sample_iconv_t;

To initialize this, the converter has to do the following in the
init subroutine:

static _LC_sample_iconv_t*
init (_LC_core_iconv_t *core_cd, char* toname,char* fromname)
{
   _LC_sample_iconv_t     *cd;    /* converter descriptor */

    /*
    **       Allocate a converter descriptor
    **/
    if(!(cd = ( _LC_sample_iconv_t *) malloc (
                    sizeof(_LC_sample_iconv_t ))))
            return (NULL);

    /*
    ** Copy the core part of converter descriptor which is
    ** passed in
    */
    cd->core = *core_cd;
    /*
    **      Return the converter descriptor
    */
    return cd;
}
```

An application invokes the **iconv** subroutine to do the actual code set conversions. The **iconv** subroutine invokes the **exec** subroutine in the core structure.

An application invokes the **iconv_close** subroutine to free any memory allocated for conversions. The **iconv_close** subroutine invokes the **close** subroutine in the core structure.

Writing a Code Set Converter

This section provides information on how to write a converter using the concepts explained so far. Every converter should define the following subroutines:

- **instantiate**
- **init**
- **iconv_exec**
- **iconv_close**

The converter-specific structure should have the core **iconv** structure as its first element. For example:

```
typedef struct _LC_example_rec {
    /* Core should be the first element */
        _LC_core_iconv_t        core;
    /* The rest are converter specific data (optional) */
        iconv_t         curcd;
```

```
                iconv_t         sb_cd;
                iconv_t         db_cd;
                unsigned char   *cntl;
        } _LC_example_iconv_t;
```

Another converter structure:

```
        typedef struct _LC_sample_iconv_rec {
                _LC_core_iconv_t        core;
        } _LC_sample_iconv_t;
```

Algorithm-Based Stateless Converters

Every converter should have the subroutines previously specified. Only the subroutine headers are provided without details, except for the **instantiate** subroutine that is common to all converters and should be coded in the same way.

The following example of an algorithm-based stateless converter is a sample converter of the IBM-850 code set to the ISO8859-1 code set.

```
#include <stdlib.h>
#include <iconv.h>
#include "850_88591.h"
/*
 *      Name :  _iconv_exec()
 *
 *      This contains actual conversion method.
 */
static size_t   _iconv_exec(_LC_sample_iconv_t *cd,
                         unsigned char** inbuf,
                         size_t *inbytesleft,
                         unsigned char** outbuf,
                         size_t *outbytesleft)
/*
 *      cd               : converter descriptor
 *      inbuf            : input buffer
 *      outbuf           : output buffer
 *      inbytesleft      : number of data(in bytes) in input buffer
 *      outbytesleft     : number of data(in bytes) in output buffer
 */
{
}

/*
 *      Name :   _iconv_close()
 *
 *      Free the allocated converter descriptor
 */
static void      _iconv_close(iconv_t cd)
{
}

/*
 *      Name :  init()
 *
 *      This allocates and initializes the converter descriptor.
 */
static _LC_sample_iconv_t        *init (_LC_core_iconv_t *core_cd,
                                  char* toname, char* fromname)
{
}

/*
 *       Name :  instantiate()
 *
 *       Core part of a converter descriptor is initialized here.
 */
_LC_core_iconv_t         *instantiate(void)
```

```
{
        static _LC_core_iconv_t   cd;

        /*
        * * Initialize _LC_MAGIC and _LC_VERSION are
        ** defined in <lc_core.h>. _LC_ICONV and _LC_core_iconv_t
        ** are defined in <iconv.h>.
         */
        cd.hdr.magic = _LC_MAGIC;
        cd.hdr.version = _LC_VERSION;
        cd.hdr.type_id = _LC_ICONV;
        cd.hdr.size = sizeof (_LC_core_iconv_t);

        /*
         *      Set pointers to each method.
         */
        cd.init = init;
        cd.exec = _iconv_exec;
        cd.close = _iconv_close;

        /*
         *      Returns the core part
         */
        return &cd;
}
```

Stateful Converters

Because stateful converters need more information, they provide additional converter-dependent information. The following example of a stateful converter is a sample converter of IBM-930 to IBM-932 code set.

The `host.h` file contains the following structure:

```
typedef struct _LC_host_iconv_rec {
                _LC_core_iconv_t        core;
                iconv_t         curcd;
                iconv_t         sb_cd;
                iconv_t         db_cd;
                unsigned char   *cntl;
        } _LC_host_iconv_t;

#include <stdlib.h>
#include <sys/types.h>
#include <iconv.h>
#include "host.h"

/*
** The _iconv_exec subroutine to be invoked via cd->exec()
*/
static int      _iconv_exec(_LC_host_iconv_t *cd,
        unsigned char **inbuf, size_t *inbytesleft,
        unsigned char **outbuf, size_t *outbytesleft)
{
        unsigned char    *in, *out;
        int              ret_value;

    if (!cd){
        errno = EBADF; return NULL;
    }

    if (!inbuf) {
        cd->curcd = cd->sb_cd;
        return ICONV_DONE;
    }

    do {
```

```
    if ((ret_value = iconv(cd->curcd, inbuf, inbytesleft, outbuf,
            outbytesleft)) != ICONV_INVAL)
            return ret_value;
        in = *inbuf;
        out = *outbuf;
        if (in[0] == SO) {
            if (cd->curcd == cd->db_cd){
                errno = EILSEQ;
                return ICONV_INVAL;
            }
            cd->curcd = cd->db_cd;
        }
        else if (in[0] == SI) {
            if (cd->curcd == cd->sb_cd){
                errno = EILSEQ;
                return ICONV_INVAL;
            }
                cd->curcd = cd->sb_cd;
        }else if (in[0] <= 0x3f &&
            cd->curcd == cd->sb_cd) {
            if (*outbytesleft < 1){
                errno = E2BIG;
                return ICONV_OVER;
            }
                out[0] = cd->cntl[in[0]];
                *outbuf = ++out;
                (*outbytesleft)--;
        }
        else {
            errno = EILSEQ; return ICONV_INVAL;
        }
        *inbuf = ++in;
        (*inbytesleft)--;
    } while (1);
}

/*
** The iconv_close subroutine is a macro accessing this
** subroutine as set in the core iconv structure.
*/
static void   _iconv_close(_LC_host_iconv_t *cd)
{
    if (cd) {
        if (cd->sb_cd)
            iconv_close(cd->sb_cd);
        if (cd->db_cd)
            iconv_close(cd->db_cd);
        free(cd);
    }else{
        errno = EBADF;
    }
}

/*
** The init subroutine to be invoked when iconv_open() is called.
*/
static _LC_host_iconv_t   *init(_LC_core_iconv_t *core_cd,
               char* toname, char* fromname)
{
        _LC_host_iconv_t*  cd;
        int   i;

    for (i = 0; 1; i++) {
        if (!_iconv_host[i].local)
            return NULL;
        if (strcmp(toname, _iconv_host[i].local) == 0 &&
            strcmp(fromname, _iconv_host[i].host) == 0)
```

```
            break;
    }

        if (!(cd = (_LC_host_iconv_t *)
                        malloc(sizeof(_LC_host_iconv_t))))
                return (NULL);

    if (!(cd->sb_cd = iconv_open(toname, _iconv_host[i].sbcs))) {
        free(cd);
        return NULL;
    }
    if (!(cd->db_cd = iconv_open(toname, _iconv_host[i].dbcs))) {
        iconv_close(cd->sb_cd);
        free(cd);
        return NULL;
    }
            cd->core = *core_cd;
    cd->cntl = _iconv_host[i].fcntl;
    cd->curcd = cd->sb_cd;
    return cd;
}

/*
** The instantiate() method is called when iconv_open() loads the
** converter by a call to __lc_load().
*/
_LC_core_iconv_t         *instantiate(void)
{
        static _LC_core_iconv_t
cd;

        cd.hdr.magic = _LC_MAGIC;
        cd.hdr.version = _LC_VERSION;
        cd.hdr.type_id = _LC_ICONV;
        cd.hdr.size = sizeof (_LC_core_iconv_t);
        cd.init = init;
        cd.exec = _iconv_exec;
        cd.close = _iconv_close;
        return &cd;
}
```

Examples

- This example provides sample code for a stateless converter that performs an algorithm-based convertion of the IBM-850 code set to the ISO8859-1 code set. The file name for this example is 850_88591.c.

```
#include <stdlib.h>
#include <iconv.h>
#include "850_88591.h"

#define DONE     0

/*
 *  Name :  _iconv_exec()
 *
 *  This contains actual conversion method.
 */
static size_t  _iconv_exec(_LC_sample_iconv_t *cd,
      unsigned char** inbuf, size_t *inbytesleft,
      unsigned char** outbuf, size_t *outbytesleft)
/*
 *  cd      : converter descriptor
 *  inbuf       : input buffer
 *  outbuf      : output buffer
 *  inbytesleft   : number of data(in bytes) in input buffer
 *  outbytesleft    : number of data(in bytes) in output buffer
```

```
*/
{
        unsigned char   *in;   /* point the input buffer */
        unsigned char   *out;  /* point the output buffer */
        unsigned char   *e_in; /* point the end of input buffer*/
        unsigned char   *e_out; /* point the end of output buffer*/

    /*
     *   If given converter discripter is invalid,
     *   it sets the errno and returns the number
     *   of bytes left to be converted.
     */
        if (!cd) {
        errno = EBADF;
        return *inbytesleft;
        }

        /*
         *      If the input buffer does not exist or there
         *      is no character to be converted, it returns
         *      0 (no characters to be converted).
         */
        if (!inbuf || !(*inbytesleft))
                return DONE;

        /*
         *      Set up pointers and initialize other variables
         */
        e_in = (in = *inbuf) + *inbytesleft;
        e_out = (out = *outbuf) + *outbytesleft;

     /*
      *     Perform code point conversion until all input
      *     is consumed.
      *     When error occurs (i.e. buffer overflow), error
      *     number is set and exit this loop.
      */
         while (in < e_in) {

        /*
         *   If there is not enough space left in output buffer
         *   to hold the converted data, it stops converting and
         *   sets the errno to E2BIG.
         */
        if (e_out <= out) {
            errno = E2BIG;
            break;
        }

        /*
         *   Convert the input data and store it into the output
         *   buffer, then advance the pointers which point to the
         *   buffers.
         */
        *out++ = table[*in++];

    }   /* while */

    /*
     *  Update the pointers to the buffers and
     *  input /output byte counts
     */
     *inbuf = in;
        *outbuf = out;
    *inbytesleft = e_in - in;
    *outbytesleft = e_out - out;
```

```
    /*
     *   Reurn the number of bytes left to be converted
     *   (0 for successful conversion completion)
     */
    return *inbytesleft;
}

/*
 *  Name :  _iconv_close()
 *
 *  Free the allocated converter descriptor
 */
static void _iconv_close(iconv_t cd)
{
    if (!cd)
        free(cd);
    else
        /*
         *  If given converter is not valid,
         *  it sets the errno to EBADF
         */
        errno = EBADF;
}

/*
 *  Name :  init()
 *
 *  This allocates and initializes the converter descriptor.
 */
static _LC_sample_iconv_t*
init (_LC_core_iconv_t *core_cd,  char* toname, char* fromname)
{
        _LC_sample_iconv_t *cd; /* converter descriptor */

    /*
     *   Allocate a converter descriptor
     */
        if (!(cd = (_LC_sample_iconv_t *)
                        malloc(sizeof(_LC_sample_iconv_t))))
                return (NULL);

    /*
     *Copy the core part of converter descriptor which is passed       *in
     */
        cd->core = *core_cd;

    /*
     *   Return the converter descriptor
     */
        return cd;
}

/*
 *  Name :  instantiate()
 *
 *  Core part of a converter descriptor is initialized here.
 */
_LC_core_iconv_t*   instantiate(void)
{
        static _LC_core_iconv_t  cd;

    /*
     *  Initialize
     *  _LC_MAGIC and _LC_VERSION are defined in <lc_core.h>.
     *  _LC_ICONV and _LC_core_iconv_t are defined in <iconv.h>.
     */
    cd.hdr.magic = _LC_MAGIC;
```

```
    cd.hdr.version = _LC_VERSION;
    cd.hdr.type_id = _LC_ICONV;
    cd.hdr.size = sizeof (_LC_core_iconv_t);

    /*
     *  Set pointers to each method.
     */
        cd.init = init;
        cd.exec = _iconv_exec;
        cd.close = _iconv_close;

    /*
     *  Returns the core part
     */
        return &cd;
}
```

- This example contains a sample header file named **850_88591.h**.

```
#ifndef _ICONV_SAMPLE_H
#define _ICONV_SAMPLE_H

/*
 *      Define _LC_sample_iconv_t
 */
typedef struct _LC_sample_iconv_rec {
        _LC_core_iconv_t   core;
} _LC_sample_iconv_t;

static unsigned char     table[] = { /*

        ________________________________________
        |                                      |
        |      IBM-850              ISO8859-1  |
        |______________________________________|
        /*      0x00     */              0x00,
        /*      0x01     */              0x01,
        /*      0x02     */              0x02,
        /*      0x03     */              0x03,
        /*      0x04     */              0x04,
        /*      0x05     */              0x05,
        /*      0x06     */              0x06,
        /*      0x07     */              0x07,
        /*      0x08     */              0x08,
        /*      0x09     */              0x09,
        /*      0x0A     */              0x0A,
        /*      0x0B     */              0x0B,
        /*      0x0C     */              0x0C,
        /*      0x0D     */              0x0D,
        .
        .
        .
        /*      0xF3     */              0xBE,
        /*      0xF4     */              0xB6,
        /*      0xF5     */              0xA7,
        /*      0xF6     */              0xF7,
        /*      0xF7     */              0xB8,
        /*      0xF8     */              0xB0,
        /*      0xF9     */              0xA8,
        /*      0xFA     */              0xB7,
        /*      0xFB     */              0xB9,
        /*      0xFC     */              0xB3,
        /*      0xFD     */              0xB2,
        /*      0xFE     */              0x1A,
        /*      0xFF     */              0xA0,
};
#endif
```

- This example is a sample makefile.

```
SHELL   = /bin/ksh
CFLAGS  = $(COMPOPT) $(INCLUDE) $(DEFINES)
INCLUDE = -I.
COMPOPT =
DEFINES = -D_POSIX_SOURCE -D_XOPEN_SOURCE
CC      = /bin/xlc
LD      = /bin/ld
RM      = /bin/rm

SRC     = 850_88591.c
TARGET  = 850_88591

ENTRY_POINT       = instantiate

$(TARGET) :
       cc -e $(ENTRY_POINT) -o $(TARGET) $(SRC) -l iconv

clean :
        $(RM) -f $(TARGET)
        $(RM) -f *.o
```

Related Information

"List of National Language Support Subroutines" on page 174.

Chapter 4, "Code Sets for National Language Support," on page 49 in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

The **iconv** command, **uuencode** and **uudecode** commands.

The **iconv_open** subroutine, **iconv** subroutine, **iconv_close** subroutine.

"List of National Language Support Subroutines" on page 174.

Chapter 4, "Code Sets for National Language Support," on page 49 in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

The **iconv** command, **uuencode** and **uudecode** commands.

The **iconv_open** subroutine, **iconv** subroutine, **iconv_close** subroutine.

Chapter 6. Input Methods

For an application to run in the international environment for which National Language Support (NLS) provides a base, input methods are needed. The Input Method is an application programming interface (API) that allows you to develop applications independent of a particular language, keyboard, or code set. Each type of input method has the following features:

Keymaps
Set of input method keymaps (imkeymaps) that works with the input method and determines the supported locales.

Keysyms
Set of key symbols (keysyms) that the input method can handle.

Modifiers
Set of modifiers or states, each having a mask value, that the input method supports.

Input Method Introduction

An input method is a set of functions that translates key strokes into character strings in the code set specified by your locale. Input method functions include locale-specific input processing and keyboard controls (for example, Ctrl, Alt, Shift, Lock, and Alt-Graphic). The input method allows various types of input, but only keyboard events are dealt with in this chapter.

Your locale determines which input method should be loaded, how the input method runs, and which devices are used. The input method then defines states and their outcome.

When the input method translates a keystroke into a character string, the translation process takes into account the keyboard and the code set you are using. You can write your own input method if you do not have a standard keyboard or if you customize your code set.

Many languages use a small set of symbols or letters to form words. To enter text with a keyboard, you press keys that correspond to symbols of the alphabet. When a character in your alphabet does not exist on the keyboard, you must press a combination of keys. Input methods provide algorithms that allow you to compose such characters.

Some languages use an ideographic writing system. They use a unique symbol, rather than a group of letters, to represent a word. For instance, the character sets used in China, Japan, Korea, and Taiwan have more than 5,000 characters. Consequently, more than one byte must be used to represent a character. Moreover, a single keyboard cannot include all the required ideographic symbols. You need input methods that can compose multibyte characters.

The **/usr/lib/nls/loc** directory contains the input methods installed on your system. You can list the contents of this directory to determine which input methods are available to you. Input method file names have the format *Language_Territory*.**im**. For example, the **fr_BE.im** file is the input method file for the French language as used in Belgium.

Through a well-structured protocol, input methods allow applications to support different input without using locale-specific input processing.

In AIX, the input method is provided in the aixterm. When characters typed from the AIXwindows interface reach the server, the characters are in the form of key codes. A table provided in the client converts key codes into *keysyms*, a predefined set of codes. Any key code generated by a keyboard should have a keysym. These keysyms are maintained and allocated by the MIT X Consortium. The keysyms are passed to the client aixterm terminal emulator. In the aixterm, the input keysyms are converted into file codes by the input method and are then sent to the application. The X server is designed to work with the display adapter provided in the system hardware. The X server communicates with the X client through sockets. Thus, the server and the client can reside on different systems in a network, provided they can communicate with each other. The data from the keyboard enters the X server, and from the server, it is passed to the terminal emulator. The terminal emulator passes the data to the application. When data comes from applications to the display device, it passes through the terminal emulator by sockets to the server and from the server to the display device.

Input Method Names

The set of input methods available depends on which locales have been installed and what input methods those locales provide. The name of the input method usually corresponds to the locale. For example, the Greek Input Method is named el_GR, which is the same as the locale for the Greek language spoken in Greece.

When there is more than one input method for a locale, any secondary input method is identified by a modifier that is part of the locale name. For example, the French locale, as spoken in Canada, has three input methods, the default and two alternative methods. The input method names are:

fr_CA	Default input method
fr_CA@im=alt	Alternative input method
fr_CA.im__64	64-bit input method

The **fr** portion of the locale represents the language name (French), and the **CA** represents the territory name (Canada). The **@im=alt** string is the modifier portion of the locale that is used to identify the alternative input method. All modifier strings are identified by the format **@im=***Modifier*.

Because the input method is a loadable object module, a different object is required when running in the 64-bit environment. In the 64-bit environment, the input method library automatically appends **__64** to the name when searching for the input method. In the preceding example, the name of the input method would be **fr_CA.im__64**.

It is possible to name input methods without using the locale name. Because the **libIM** library does not restrict names to locale names, the calling application must ensure that the name passed to **libIM** can be found. However, applications should request only modifier strings of the form **@im=***Modifier* and that the user's request be concatenated with the return string from the **setlocale** (**LC_CTYPE,NULL**) subroutine.

Input Method Areas

Complex input methods require direct dialog with users. For example, the Japanese Input Method may need to show a menu of candidate strings based on the phonetic matches of the keys that you enter. The feedback of the key strokes appears in one or more areas on the display. The input method areas are as follows:

Status
: Text data and bitmaps can appear in the Status area. The Status area is an extension of the light-emitting diodes (LEDs) on the keyboard.

Pre-edit
: Intermediate text appears in the Pre-edit area for languages that compose before the client handles the data.

: A common feature of input methods is that you press a combination of keys to represent a single character or set of characters. This process of composing characters from keystrokes is called *pre-editing*.

Auxiliary
: Menus and dialogs that allow you to customize the input method appear in the Auxiliary area. You can have multiple Auxiliary areas managed by the input method and independent of the client.

Management for input method areas is based on the division of responsibility between the application (or toolkit) and the input method. The divisions of responsibility are as follows:

- Applications are responsible for the size and position of the input method area.
- Input methods are responsible for the contents of the input area. The input method area cannot suggest a placement.

Input Method Command

An Input Method is a set of subroutines that translate key strokes into character strings in the code set specified by a locale. Input Method subroutines include logic for locale-specific input processing and keyboard controls (Ctrl, Alt, Shift, Lock, Alt Graphic). The following command allows for the customizing of input method mapping for the use of input method subroutines:

keycomp
Compiles a keyboard mapping file into an input method keymap file.

Programming Input Methods

The input method is a programming interface that allows applications to run in an international environment provided through NLS. The input method has the following characteristics:

- Localized input support (defined by locale)
- Multiple keyboard support
- Multibyte character-input processing

Note: Do not assume any particular physical keyboard is in use. Use an input method based on the locale setting to handle keyboard input.

Initialization

You can use the **IMQueryLanguage** subroutine to determine if an input method is available without initializing it. An application (toolkit) initializes a locale-specific input method by calling the **IMInitialize** subroutine, which initializes a locale-specific input method editor (IMED). The subroutine uses the **LOCPATH** environment variable to search for the input method named by the **LANG** environment variable. The **LOCPATH** environment variable specifies a set of directory names used to search for input methods.

If the input method is found, the **IMInitialize** subroutine uses the **load** subroutine to load the input method and attach the **imkeymap** file. When the input method is accessed, an object of the type **IMFep** (input method front-end processor) is returned. The **IMFep** should be treated as an opaque structure.

The **IMInitialize** subroutine links the converter function using the **load** subroutine. The **load** subroutine is similar to the **exec** subroutine and links the converter program at run-time. Since the **IMInitialize** subroutine is called as a library function, it must preserve security for certain programs. When the **IMInitialize** subroutine is called from a set root ID program, it ignores the **LOCPATH** environment variable and searches for converters only in the **/usr/lib/nls/loc/iconv** and **/etc/nls/loc/iconv** directories.

Each **IMFep** inherits the locale's code set when the **IMInitialize** subroutine is called. Consequently, strings returned by the **IMFilter** and **IMLookupString** subroutines are in the locale's code set. Changing the locale after the **IMInitialize** subroutine is called does not affect the code set of the **IMFep**.

For each **IMFep**, the application can use the **IMCreate** subroutine to create one or more **IMObject** instances. The **IMObject** manages its own state and can manage several Input Method Areas (see "Input Method Areas" on page 124). How each **IMObject** defines input processing depends on the code set and keyboard associated with the locale. In the simplest case, a single **IMObject** is needed if the application is managing a single dialog with the user. The input method also supports other user interfaces where the application allows multiple dialogs with the user, and each dialog requires one **IMObject**.

The difference between an **IMFep** and **IMObject** is that the **IMFep** is a handle that binds the application to the code of the input method, while the **IMObject** is a handle that represents an instance of a state of an input device, such as a keyboard. The **IMFep** does not represent a state of the input method. Each **IMObject** is initialized to a specific input state and is changed according to the sequence of events it receives.

After the **IMObject** is created, the application can process key events. The application should pass key events to the **IMObject** using the **IMFilter** and **IMLookupString** subroutines. These subroutines are provided to isolate the internal processing of the IMED from the customized key event mapping process.

Input Method Management

The input method provides the following subroutines for maintenance purposes:

IMInitialize
Initializes the standard input method for a specified language. Returns a handle to an IMED associated with the locale. The handle is an opaque structure of type **IMFep**.

IMQueryLanguage
Checks whether the specified language is supported.

IMCreate
Creates one instance of a particular input method. This subroutine must be called before any key event processing is performed.

IMClose
Closes the input method.

IMDestroy
Destroys an instance of an input method.

Input Method Keymap Management

The input method provides several subroutines to map key events to a string. The mapping is maintained in an **imkeymap** file located in the **LOCPATH** directory. The subroutines used for mapping are as follows:

IMInitializeKeymap
Initializes the imkeymap associated with a specified language.

IMFreeKeymap
Frees resources allocated by the **IMInitializeKeymap** subroutine.

IMAIXMapping
Translates a pair of key-symbol and state parameters to a string and returns a pointer to that string.

IMSimpleMapping
Translates a pair of key-symbol and state parameters to a string and returns a pointer to that string.

Key Event Processing

Input processing begins when you press keys on the keyboard. The application must have created an **IMObject** before calling these functions:

IMFilter
Asks the IMED to indicate if a key event is used internally. If the IMED is composing a localized string, it maps the key event to that string.

IMLookupString
Maps the key event to a localized string.

IMProcessAuxiliary
Notifies the input method of input for an auxiliary area.

IMIoctl
Performs a variety of control or query operations on the input method.

Callbacks

The IMED communicates directly with the user by using the Input Method-Callback (IM-CB) API to access the graphic-dependent functions (*callbacks*) provided by the application. The application attaches the callbacks, which perform output functions and query information, to the **IMObject** during initialization. The application still handles all the input.

The set of callback functions that the IMED uses to communicate with a user must be provided by the caller. See “Using Callbacks” on page 128 for a discussion of the subroutines defined by the IM-CB API.

Input Method Structures

The major structures used by the input method are as follows:

IMFepRec	Contains the front end information
IMObjectRec	Contains the common part of input method objects
IMCallback	Registers callback subroutines to the **IMFep**
IMTextInfo	Contains information about the text area, primarily the pre-editing string
IMAuxInfo	Defines the contents of the auxiliary area and the type of processing requested
IMIndicatorInfo	Indicates the current value of the indicators
IMSTR	Designates strings that are not null-terminated
IMSTRATT	Designates strings that are not null-terminated and their attributes

Working with Keyboard Mapping

The following model shows how input methods are used by applications. Use this information to help you customize keyboard mapping.

Input processing is divided into three steps:

1. **keycode/keystate(raw) - > keysym/modifier(new)**

   This step is application and environment-dependent. The application is responsible for mapping the raw key event into a keysym/modifier for input to the input method.

   In the AIXwindows environment, the client uses the server's keysym table, **xmodmap**, which is installed at the server, to perform this step. The **xmodmap** defines the mapping of the Shift, Lock, and Alt-Graphic keys. The client uses the **xmodmap** as well as the Shift and Lock modifiers from the X event to determine the keysym/modifier represented by this event.

   For example, if you press the `XK_a` keysym with a Shift modifier, the **xmodmap** maps it to the `XK_A` keysym. Because you used the Shift key to map the key code to a keysym, the application should mask the Shift modifier from the original X event. Consequently, the input to the input method would be the `XK_A` keysym and no modifier.

   In another environment, if the device provides no additional information, the input method receives the `XK_a` keysym with the Shift modifier. The input method should perform the same mapping in both cases and return the letter A.

2. **keysym/modifier(new) - > localized string**

   This step depends on the localized IMED and varies with each locale. It notifies the IMED that a key event occurred and to ask for an indication that their IMED uses the key event internally. This occurs when the application calls the **IMFilter** subroutine.

   If the IMED indicates that the key event is used for internal processing, the application ignores the event. Because the IMED is the first to see the event, this step should be done before the application interprets the event. The IMED only uses key events that are essential.

   If the IMED indicates the event is not used for internal processing, the application performs the next step.

3. **keysym/modifier(new) - > customized string**

   This step occurs when the application calls the **IMLookupString** subroutine. The input method keymap (created by the **keycomp** command) defines the mapping for this phase. It is the last attempt to map the key event to a string and allows a user to customize the mapping.

   If the keysym/modifier (new) combination is defined in the input method keymap (imkeymap), a string is returned. Otherwise, the key event is unknown to the input method.

Input Method Keymaps

The input method provides support for user-defined imkeymaps, allowing you to customize input method mapping. The input methods support imkeymaps for each locale. The file name for imkeymaps is similar to that of input methods, except that the suffix for imkeymap files is **.imkeymap** instead of **.im**.

This example uses the Italian input method to illustrate how you can customize your **imkeymap** file:

1. Copy the default **imkeymap** source file to your **$HOME** directory by typing:
   ```
   cd $HOME
   cp /usr/lib/nls/loc/it_IT.ISO8859-1.imkeymap.src .
   ```
2. Edit the **imkeymap** source file following the default file format by typing:
   ```
   vi it_IT.ISO8859-1.imkeymap.src
   ```
3. Compile the **imkeymap** source file by typing:
   ```
   keycomp < it_IT.ISO8859-1.imkeymap.src > it_IT.ISO8859-1.imkeymap
   ```
4. Make sure the **LOCPATH** environment variable specifies **$HOME** before **/usr/lib/nls/loc** by typing:
   ```
   LOCPATH=$HOME:$LOCPATH
   ```

   Note: All **setuid** and **setgid** programs ignore the **LOCPATH** environment variable.

Inbound and Outbound Mapping

The imkeymaps map a key symbol to a file code set string. The localized imkeymaps found in the **/usr/lib/nls/loc** library are defined to include mapping for all of the inbound keys. The imkeymaps provide the following types of mapping:

Inbound mapping
Mapping of a keysym/modifier that generates a target string encoded in the code set of the locale.

Outbound mapping
Mapping of a keysym/modifier that does not generate a target string included in the code set of the locale.

A special imkeymap, /**usr**/**lib**/**nls**/**loc**/**C@outbound.imkeymap**, defines outbound mapping for all keyboards made by this manufacturer and is primarily intended for use by aixterm. This imkeymap includes mapping of PF keys, cursor keys, and other special keys commonly used by applications. Internationalized applications that use standard input and standard output should limit their dependency on outbound mapping, which does not vary on different keyboards. For example, the Alt-a is defined in the same way on all keyboards made by this manufacturer. Yet, the Alt-tilde is different depending on the keyboard used.

The aixterm bases its outbound mapping on the **C@outbound** imkeymap. Applications that require more mapping should modify the localized imkeymap source to include the necessary definitions.

Using Callbacks

Applications that use input methods should provide callback functions so that the Input Method Editor (IMED) can communicate with the user. The type of input method you use determines whether or not callbacks are necessary. For example, the single-byte input method does not need callbacks, but the Japanese input method uses them extensively with the pre-edit facility. Pre-editing allows processing of characters before they are committed to the application.

When you use an input method, only the application can insert or delete pre-edit data and scroll the text. Consequently, the echo of the keystrokes is achieved by the application at the request of the input method logic through callbacks.

When you enter a keystroke, the application calls the **IMFilter** subroutine. Before returning, the input method can call the echoing callback function for inserting new keystrokes. After a character has been composed, the **IMFilter** subroutine returns it, and the keystrokes are deleted.

In several cases, the input method logic has to call back the client. Each of these is defined by a callback action. The client specifies which callback should be called for each action.

Types of callbacks are described as follows:

- Text drawing

  The IMED uses text callbacks to draw any pre-editing text currently being composed. When the callbacks are needed, the application and the IMED share a single-line buffer, where the editing is performed. The IMED also provides cursor information that the callbacks then present to the user.

  The text callbacks are as follows:

IMTextDraw Asks the application program to draw the text string
IMTextHide Tells the application program to hide the text area
IMTextStart Notifies the application program of the length of the pre-editing space
IMTextCursor Asks the application program to move the text cursor

- Indicator (status)

  The IMED uses indicator callbacks to request internal status. The **IMIoctl** subroutine works with the **IMQueryIndicatorString** command to retrieve the text string that provides the internal status. Indicator callbacks are similar to text callbacks, except that instead of sharing a single-line buffer, a status value is used.

  The indicator callbacks are as follows:

IMIndicatorDraw Tells the application program to draw the status indicator
IMIndicatorHide Tells the application program to hide the status indicator
IMBeep Tells the application program to emit a beep sound

- Auxiliary

  The IMED uses auxiliary callbacks to request complex dialogs with the user. Consequently, these callbacks are more sophisticated than text or indicator callbacks.

  The auxiliary callbacks are as follows:

IMAuxCreate Tells the application program to create an auxiliary area
IMAuxDraw Tells the application program to draw an auxiliary area
IMAuxHide Tells the application program to hide an auxiliary area
IMAuxDestroy Tells the application program to destroy an auxiliary area

  The **IMAuxInfo** structure defines the dialog needed by the IMED.

  The contents of the auxiliary area are defined by the **IMAuxInfo** structure, found in the **/usr/include/im.h** library.

  The **IMAuxInfo** structure contains the following fields:

IMTitle Defines the title of the auxiliary area. This is a multibyte string. If `title.len` is 0, no title displays.

IMMessage
Defines a list of messages to be presented. From the applications perspective, the **IMMessage** structure should be treated as informative, output-only text. However, some input methods use the **IMMessage** structure to conduct a dialog with the user in which the key events received by way of the **IMFilter** or **IMLookupString** subroutine are treated as input to the input method. In such cases, the input method may treat the **IMMessage** structure as either a selectable list or a prompt area. In either case, the application displays only the message contents.

The **IMProcessAuxiliary** subroutine need not be called if the **IMSelection** structure contains no **IMPanel** structures and the `IMButton` field is null.

The `message.nline` indicates the number of messages contained in the **IMMessage** structure. Each message is assumed to be a single line. Control characters, such as \t, are not recognized. The text of each message is defined by the **IMSTRATT** structure, which consists of both a multibyte string and an attribute string. Each attribute is mapped one-to-one for each byte in the text string.

If `message.cursor` is True, then the **IMMessage** structure defines a text cursor at location `message.cur_row`, `message.cur_col`. The `message.cur_col` field is defined in terms of bytes. The `message.maxwidth` field contains the maximum width of all text messages defined in terms of columns.

IMButton
Indicates the possible buttons that can be presented to a user. The **IMButton** field tells the application which user interface controls should be provided for the end user. The button member is of type int and may contain the following masks:

IM_OK Present the OK button.

IM_CANCEL
Present the CANCEL button.

IM_ENTER
Present the ENTER button.

IM_RETRY
Present the RETRY button.

IM_ABORT
Present the ABORT button.

IM_YES
Present the YES button.

IM_NO Present the NO button.

IM_HELP
Present the HELP button.

IM_PREV
Present the PREV button.

IM_NEXT
Present the NEXT button.

The application should use the **IMProcessAuxiliary** subroutine to communicate the button selection.

IMSelection Defines a list of items, such as ideographs, that an end user can select. This structure is used when the input method wants to display a large number of items but does not want to control how the list is presented to the user.

The **IMSelection** structure is defined as a list of **IMPanel** structures. Not all applications support **IMSelection** structures inside the **IMAuxInfo** structure. Applications that do support **IMSelection** structures should perform the **IM_SupportSelection** operation using the **IMIoctl** subroutine immediately after creation of the **IMObject**. In addition, not all applications support multiple **IMPanel** structures. Therefore, the `panel_row` and `panel_col` fields are restricted to a setting of 1 by all input methods.

Each **IMPanel** structure consists of a list of `IMItem` fields that should be treated as a two-dimensional, row/column list whose dimensions are defined as `item_row` times `item_col`. If `item_col` is 1, there is only one column. The size of the **IMPanel** structure is defined in terms of bytes. Each item within the **IMPanel** structure is less than or equal to `panel->maxwidth`.

The application should use the **IMProcessAuxiliary** subroutine to communicate one or more user selections. The **IM_SELECTED** value indicates which item is selected. The **IM_CANCEL** value indicates that the user wants to terminate the auxiliary dialog.

hint Used by the input method to provide information about the context of the **IMAuxInfo** structure. A value of `IM_AtTheEvent` indicates that the **IMAuxInfo** structure is associated with the last event passed to the input method by either the **IMFilter** or **IMLookupString** subroutine. Other hints are used to distinguish when multiple **IMAuxInfo** structures are being displayed.

status Used by the input method for internal processing. This field should not be used by applications.

Each **IMAuxInfo** structure is independent of the others. The method used for displaying the members is determined by the caller of the input method. The **IMAuxInfo** structure is used by the **IMAuxDraw** callback.

Initializing Callbacks

All callbacks must be identified when you call the **IMCreate** subroutine. The **IMCallback** structure contains the address for each callback function. The caller of the **IMCreate** subroutine must initialize the **IMCallback** structure with the addresses.

The callback functions can be called before the **IMCreate** subroutine returns control to the caller. Usually, the **IMTextStart** callback is called to identify the size of the pre-edit buffer.

Bidirectional Input Method

The Bidirectional Input Method (BIM) is similar to the Single-Byte Input Method except that it is customized to process the Arabic and Hebrew keyboards. BIM also links the Hebrew and Arabic states to the Latin states. The Alt+Right Shift keys allow the user to toggle between the Arabic/Hebrew and Latin language layers. The use of these keys is derived from BIM. The features of BIM are as follows:

- Supports Arabic, Hebrew, and Latin states
- Supports the ISO8859-6, ISO8859-8, IBM-1046, and IBM-856 code sets
- Performs diacritical composing

Keymaps

The following keymaps are supported on BIM:

- ar_AA.ISO8859-6.imkeymap
- ar_AA@alt.ISO8859-6.imkeymap
- Ar_AA.IBM-1046.imkeymap
- Ar_AA@alt.IBM-1046.imkeymap
- iw_IL.ISO8859-8.imkeymap

- iw_IL@alt.ISO8859-8.imkeymap
- Iw_IL.IBM-856.imkeymap
- Iw_IL@alt.IBM-856.imkeymap

Key Settings

The following key settings are supported on BIM:

scr-rev()	Reverses the screen orientation and sets the keyboard layer to the default language of the new orientation.
ltr-lang()	Enables the Latin keyboard layer.
rtl-lang()	Enables the Arabic/Hebrew keyboard layer.
col-mod()	Enables the column heading adjustment, which handles each word as a separate column.
auto-push()	Toggles the Autopush mode, which handles mixed left-to-right and right-to-left text. When you enable the Autopush mode, reversed segments are automatically initiated and terminated according to the entered character or the selected language layer. Thus, you are relieved of manually invoking the Push function.
chg-push()	Toggles the Push mode. This mode causes the cursor to remain in its position and pushes the typed characters in the direction opposed to the field direction.
shp-in()	Shapes Arabic characters in their initial forms.
shp-is()	Shapes Arabic characters in their isolated forms.
shp-p()	Shapes Arabic characters in their passthru forms.
shp-asd()	Shapes Arabic characters in their automatic forms.
shp-m()	Shapes Arabic characters in their middle forms.
shp-f()	Shapes Arabic characters in their final forms.

Modifiers

The following modifiers are supported on BIM:

ShiftMask	0x01
LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10

Cyrillic Input Method (CIM)

The Cyrillic Input Method (CIM) is similar to the Single-Byte Input Method, except that it is customized for processing the Cyrillic keyboard. The features of CIM are as follows:

- Supports Cyrillic and Latin states.You can toggle between the two states by pressing the Alt key and the Left or Right Shift key simultaneously.

  Note: The Alt-Graphic (Right Alt) key can be used to generate additional characters within each keyboard layer.
- For the Russian and Bulgarian locales, both 101-key and 102-key keyboard drivers are supported.
- Supports the ISO8859-5 code set.

Keymap

The following keymaps are supported on the CIM:

- bg_BG.ISO8859-5.imkeymap
- mk_MK.ISO8859-5.imkeymap
- sr_SP.ISO8859-5.imkeymap

- ru_RU.ISO8859-5.imkeymap
- be-BY.ISO8859-5.imkeymap
- uk-UA.ISO8859-5.imkeymap

Keysyms

The CIM uses the keysyms in the **XK_CYRILLIC**, **XK_LATIN1**, and **XK_MISCELLANY** groups.

The following reserved keysyms are unique to the input method of this system:

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff
XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdiaeresis	0x180007ae

Modifiers

The following modifiers are supported on CIM:

ShiftMask	0x01
LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10

The following internal modifier is supported on CIM:

Cyrillic Layer	0x20

Greek Input Method (GIM)

The Greek Input Method (GIM) is similar to the Single-Byte Input Method (SIM), but handles both Latin and Greek character sets, by providing two layers or states of keyboard mappings, which correspond to the two character sets.

The keyboard is initially in the Latin input state. However, if the left-shift key is pressed while the left-alt key is held down, the keyboard is put in the Greek input state. The keyboard can be returned to the Latin state by pressing the right-shift key, while the left-alt key is held down. These are locking shift keys, because the state is locked when they are pressed.

While in the Greek state, the input method recognizes the following diacritical characters and valid subsequent characters for diacritical composing as shown in the following table:

Greek Composing Characters

Keysym	Valid Composing Characters
dead_acute	uppercase and lowercase: alpha, epsilon, eta, iota, omicron, upsilon, omega
dead_diaeresis	uppercase and lowercase: iota, upsilon
dead_accentdiaeresis	lowercase only: iota, upsilon

In the Latin state, there are no composing diacriticals, and the keys shown in the table above are treated as simple graphic characters.

The Greek and Single-Byte Input Methods also differ in their handling of illegal diacritical composing sequences. In such cases, the GIM beeps and returns no characters. The SIM does not beep and returns both the diacritical character and a graphic character associated with the invalid key.

Note: The Alt-Graphic (right-alt) key can be used to generate additional characters within each keyboard state.

Keymap

The following keymap is supported on GIM:

- el_GR.ISO8859-7.imkeymap

Keysyms

The GIM uses the keysyms in the **XK_LATIN1**, **XK_GREEK**, and **XK_MISCELLANY** groups.

The following reserved keysyms are unique to the input method of this system.

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff
XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdiaeresis	0x180007ae

Modifiers

The following modifiers are supported on GIM:

ShiftMask	0x01
LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10

The following internal modifier is supported on GIM:

Greek Layer	0x20

Japanese Input Method (JIM)

The Japanese Input Method (JIM) provides Japanese input. The features include the following:

- Supports Romaji to Kana character conversion (RKC).
- Supports Kana to Kanji character conversion (KKC).
- Includes Hankaku (half-width) and Zenkaku (full-width) character input.
- Provides system and user dictionary lookup.
- Provides run-time registration of a word to the user dictionary.
- Requires Callback functions to support:
  - Status and Pre-edit drawing
  - All candidate menus
  - JIS Kutan number input and IBM Kanji number input
- Supports IBM-943, IBM-932 and IBM-eucJP code sets. For internal processing, the JIM uses the IBM-942 code set. However, the JIM supports any code set, such as IBM-eucJP, that can be converted from IBM-932.
- Located in the **/usr/lib/nls/loc/JP.im** file. All other localized input methods are aliases to this file.

The Japanese code sets consist of the following character groups:

- Katakana
- Hiragana
- Kanji

Katakana and Hiragana consist of approximately 50 characters each and form the set of phonetic characters referred to as Kana. All of the sounds in the Japanese language can be represented in Kana.

Kanji is a set of ideographs. A simple concept can be represented by a single Kanji character, while more complicated meanings can be formed with strings of Kanji characters. Several thousand Kanji characters exist.

The Japanese also use the Roman alphabet. Called Romaji, the Roman alphabet consists of 26 characters. It is used mostly in technical and professional environments to represent technical vocabulary that does not exist in Japanese. A typical sentence is usually a mixture of Katakana, Hiragana, Kanji, Romaji, numbers, and other characters.

Japanese Character Processing

The Japanese Industrial Standard (JIS) specifies about 7000 Kanji characters processed by computer systems. Japanese products made by this manufacturer support all of the standard characters, as well as others. Input of the characters is accomplished through the following:

- Kana-to-Kanji conversion (KKC)
- Romaji-to-Kana conversion (RKC)

The following special keys appear on the 106-key Japanese keyboard to allow for these conversions:

Special Japanese Keys		
Key Function	**Key Name**	**Description of Function**
KKC Non-conversion key	muhenkan	Leaves Kana characters as is.
KKC Conversion key	henkan	Converts Kana to Kanji.

KKC All Candidates key	zenkouho	Shows all possible Kanji representatives.
RKC Romaji Mode key	romaji	Toggles RKC on and off.
Hiragana Shift key	hiragana	Becomes Hiragana shift state.
Katakana Shift key	katakana	Becomes Katakana shift state.
Romaji Shift key	eisu	Becomes Romaji shift state.

Note: Shift states are maintained until you press another shift key. The initial state is Romaji.

Kana-To-Kanji Conversion (KKC) Technology

The Japanese Input Method's (JIM) KKC technology is based on the fact that every Kanji character or set of Kanji characters has a phonetic sound or sounds that can be expressed by Katakana or Hiragana characters.

It is much easier to input Hiragana or Katakana characters than Kanji characters. The JIM analyzes the phonetic values of the Hiragana and Katakana characters to determine the best Kanji-character equivalent. Such phonetic analysis depends on the dictionary and tables provided to the JIM.

Input Modes

The JIM has the following modes that can be used to control the input processing:

- Keyboard Mapping

  Allows invocation of alphanumeric, Katakana, or Hiragana modes.
- Character Size

  Inputs in Zenkaku (full-width) or Hankaku (half-width) mode.
- RKC off/on

  Inputs Kana directly or invokes the pre-edit composing mode to input Kana with a combination of alphabetic characters. The pre-editing facility allows processing of characters before they are committed to the application.

When the keyboard-mapping mode is alphanumeric and the character size mode is Hankaku, the JIM maps keys to Romaji characters. This mode combination is known as the ″English″ mode. Pre-editing is not needed in English mode and cannot be invoked regardless of the RKC mode setting. The other mode combinations may initiate pre-editing and characters generated in these modes are not ASCII.

The following keys are used to perform Kana-to-Kanji conversion by the JIM.

Keysym	Keyboard Mapping
Katakana	Katakana shift
Eisu_toggle	Alphanumeric shift
Hiragana	Hiragana shift

Keysym	Character Size
Zenkaku_Hankaku	Full-width or Half-width toggle
Hankaku	Half-width
Zenkaku	Full-width

Keysym	RKC on/off
Alt-Hiragana	Enables/Disables Romaji-to-Kana conversion

Romaji	*The same effect

* Keysyms unique to the manufacturer

The following keys are also used when the JIM is pre-editing a Kanji string.

Keysym	Kanji pre-edit
Muhenkan	Non-conversion - commit Kana
Henkan	Conversion - get next candidate
Kanji	Same as Henkan
BunsetsuYomi	*Moves back a phrase
MaeKouko	*Moves to previous candidate
LeftDouble	*Moves cursor two characters left
RightDouble	*Moves cursor two characters right
ErInput	*Discards the current pre-edit string

Keysym	Auxiliary pre-edit
Alt-Henkan	All candidates
Touroku	Run-time registration
ZenKouho	*All candidates (the same effect)
KanjiBangou	*Kanji Number Input
HenkanMenu	*Changes conversion mode

* Keysyms unique to the manufacturer

Keyboard Mapping

The following keyboard mapping states are possible: Alphanumeric (Romaji), Katakana, and Hiragana. Each state is invoked by a keysym that acts as a locking shift key. The keysyms are Katakana, Eisu_toggle, and Hiragana shift.

When one of these keysyms is pressed, keyboard mapping enters the state associated with the key. This state is maintained until one of the other keysyms is pressed. The initial shift state is Eisu_toggle, which can be changed by customization.

When you invoke the Hiragana or Katakana state, each key is mapped to a phonetic character within the respective character set. For example, if you press *q*, a Hiragana character pronounced ″ta″ is produced during Hiragana shift state, a Katakana character pronounced ″ta″ is produced during Katakana shift state, or a Romaji *q* is produced during Eisu_toggle shift state. On Japanese IBM keyboards, the tops of keys show all three symbols.

Also, when keyboard mapping is in Hiragana state, the input method is automatically put into a composing pre-editing mode where each Hiragana character can be converted into a Kanji character. See “Kanji Pre-edit” on page 138 for more information.

Some keys have two Hiragana or Katakana characters assigned. For example, the 7 key has large and small Hiragana characters both having the pronunciation ″ya″. These characters are not uppercase and lowercase equivalents of each other because Kanji, Hiragana, and Katakana do not have uppercase and lowercase. The small characters are used to express special phonetic sounds. These characters can be distinguished by using the shift key.

Character Size

A subset of the Japanese character set is represented in both full-width and half-width. Kanji ideographic characters are usually full-width. The phonetic and ASCII characters have both full-width and half-width representations. The user controls character size by pressing the Zenkaku_Henkaku keysym, which toggles between full-width and half-width.

Romaji-To-Kana Conversion (RKC)

For users familiar with alphanumeric keyboards, it is easier to type the phonetic sounds rather than the Hiragana or Katakana characters. The JIM provides Romaji-to-Kana conversion (RKC), allowing the user to type in the phonetic sounds of Hiragana or Katakana characters on an alphanumeric keyboard.

Kanji Pre-edit

When operating in Romaji-To-Kana conversion mode, you must follow two steps to produce Kanji characters. First, the user inputs Hiragana characters by typing their Romaji phonetic characters. In this step, you produce a Hiragana character by typing 1 to 3 Romaji alphabetic keys that compose the phonetic sound of the Hiragana character. Second, convert the Hiragana characters to Kanji characters by pressing the Henkan key. Many Kanji characters may be associated with a single phonetic phrase. The Henkan key displays the most likely Kanji candidates. Repeated pressing of the Henkan key displays all the additional candidates.

For example, when you enter the Kanji characters for the phonetic sound ″k-a-n-j-i″, you must do two things:

1. Set the keyboard mapping to the Hiragana state.
2. Enable Romaji-to-Kana mapping by pressing the Alt-Hiragana key. This action invokes the alphanumeric keyboard.

You can now press the keys that spell ″kanji″. As each phonetic sound is completed, a Hiragana character displays.

The Hiragana character is displayed with visual feedback to indicate that the JIM is composing in a pre-edit state. The character is underlined and shown in reverse video. This feedback facility is known as a *callback*. See “Using Callbacks” on page 128 for more information.

To convert the Hiragana character within the pre-edit string to a Kanji character, press the Henkan key. The most likely candidate associated with the phonetic Hiragana sound displays. Pressing this key repeatedly shows other candidates.

During the composition process, the pre-edit string is partitioned into segments that can be considered Kanji words. After a string of kana characters is converted into a candidate, it is treated as one of these convertible segments. While the pre-edit string is displayed, the JIM uses the cursor key and other keys to manipulate the string.

To commit the pre-edit string to the program, the user presses the Enter key. In this case, the Enter key code itself is not sent to the program, only the string.

The Muhenkan keysym can also be used to turn off pre-edit and commit the Hiragana or Katakana character directly to the program.

The following table depicts the shift state transition and the interaction of the RKC mode key with the shift states.

Character Encoding	Code Points	Description	Count
000xxxxx	00–1F	Controls	32

00100000	20	Space	1
0xxxxxxx	21–7E	7-bit ASCII	94
01111111	7F	Delete	1
10000000	80	Undefined	1
100xxxxx 01xxxxxx	[81–9F] [40–7E]	Double byte	1953
100xxxxx 1xxxxxxx	[81–9F] [80–FC]	Double byte	3844
10100000	A0	Undefined	1
1xxxxxxx	A1–DF	8-bit single byte	63
111xxxxx 01xxxxxx	[E0–FC] [40–7E]	Double byte	1827
111xxxxx 1xxxxxxx	[E0–FC] [80–FC]	Double byte	3596
11111101	FD	Undefined	1
11111110	FE	Undefined	1
11111111	FF	All ones	1

The JIM has the following types of auxiliary areas:

- All Candidates menu
- Kanji Number Input dialog
- Conversion Mode menu
- Runtime Registration dialog

A Kana-to-Kanji conversion operation on a string of Hiragana or Katakana characters can yield from one to a hundred Kanji candidates. At worst, you would have to press the conversion key more than a hundred times to get the correct Kanji character.

In such cases, it is more convenient to find the correct character by requesting the All Candidates menu with the ZenKouho or the Alt-Henkan keysym. This menu displays if the current target (a Kanji word that the cursor is pointing to in the pre-edit area) has several alternative candidates associated with it. The menu contains multiple candidates for selection. The All Candidates menu disappears when the Reset keysym is pressed, the Enter key is pressed, or a candidate is selected.

A Kanji Number Input dialog prompts the user to select the Kanji character by entering 3 to 5 digits. The digits represent the code of the character. Online dictionaries allow a user to search for the code. The ordering formats for these dictionaries vary. For example, one dictionary lists codes by phonetic sound. Another dictionary orders codes by the number of strokes used to compose the character. The KanjiBangou keysym invokes this menu. The menu is terminated with either the Reset or Return keysym.

The HenkanMenu keysym invokes the Conversion Mode menu. Four items are displayed for selection. The most important items are the word-conversion mode and phrase-conversion mode. Make a selection by choosing a number and pressing the Return keysym. This menu is terminated when either a selection is made or the Reset keysym is pressed.

A run-time registration dialog prompts the user to input a Kana string and a Kanji string for registering the mapping of the strings in the user dictionary. After the pair is registered, the JIM can use it as a conversion candidate. The menu is terminated with the Escape or Reset keysym.

The presentation of menus depends on the interface environment in which the JIM is operating. For example, some interfaces support scrolling menus that use the Page Down and Page Up keys.

Keymaps

The following keymaps are supported by the JIM:

- ja_JP.IBM-eucJP.imkeymap
- Ja_JP.IBM-932.imkeymap
- Ja_JP.IBM-943.imkeymap

Keysyms

The JIM uses the keysyms in the **XK_KATAKANA**, **XK_LATIN1**, and **XK_MISCELLANY** groups.

The following reserved keysyms are unique to the input method of this system:

XK_BunsetsuYomi	0x1800ff05	Back a phrase to Yomi
XK_MaeKouho	0x1800ff04	Previous candidate
XK_ZenKouho	0x1800ff01	All candidates.
XK_KanjiBangou	0x1800ff02	Kanji number input.
XK_HenkanMenu	0x1800ff03	Changes conversion mode.
XK_LeftDouble	0x1800ff06	Moves cursor two characters left.
XK_RightDouble	0x1800ff07	Moves cursor two characters right.
XK_LeftPhrase	0x1800ff08	Reserved for future use.
XK_RightPhrase	0x1800ff09	Reserved for future use.
XK_ErInput	0x1800ff0a	Discards the current pre-edit string
XK_Resetreset	0x1800ff0b	Reset

XK_Kanji Convert Hiragana to Kanji.
XK_Muhenkan Cancels conversion.
XK_Romaji Puts JIM in Romaji input mode.
XK_Hiragana Puts JIM in Hiragana input mode.
XK_Katakana Puts JIM in Katakana input mode.
XK_Zenkaku_Hankaku Toggles between full-width and half-width character input mode.
XK_Touroku Registers a word to the user dictionary.
XK_Eisu_toggle Puts JIM in alphanumeric input mode.

Modifiers

The following modifiers are supported by the JIM:

ShiftMask 0x01
LockMask 0x02
ControlMask 0x04
Mod1Mask (Left-Alt) 0x08
Mod2Mask (Right-Alt) 0x10

The following internal modifiers are supported by the JIM:

Kana 0x20
Romaji 0x40

Korean Input Method (KIM)

The Korean EUC code set consists of the following main character groups:

- ASCII (English)
- Hangul (Korean characters)

The Hangul code set includes Hangul and Hanja (Chinese) characters. One Hangul character can comprise several consonants and vowels. However, most Hangul words can be expressed in Hanja. Each Hanja character has its own meaning and is thus more specific than Hangul.

The current Korean standard code set, KSC5601, contains 8224 Hangul, Hanja, and special characters. To comply with the Korean standard Extended UNIX Code (EUC), this code set is assigned to CS1 of the EUC.

Input of characters can be accomplished through the following:

- ASCII

  ASCII mode is used for entering English characters.
- Hangul

  The XK_Hangul key invokes Hangul mode, which must be used to enter Hangul characters. After Hangul mode is invoked, the KIM composes incoming consonants and vowels according to Hangul composition rules. A Hangul character is composed of a consonant followed by a vowel. A final consonant is optional. If incoming characters violate the construct rule, a warning beep is sounded.

  There are about 1500 special characters in the standard code set. These characters must be entered with the Code Input function of the KIM. The Code Input key invokes the Code Input function. When the Code Input function is invoked, the code point for a desired character can be entered in the Code Input auxiliary window.
- Hanja

  The XK_Hangul_Hanja key invokes the Hanja mode. Hanja characters can only be converted from the appropriate Hangul character. There are two modes for Hangul-to-Hanja Conversion (HHC): single-candidate and multi-candidate. In this context, a candidate is a selection of possible character choices.

  In single-candidate mode, the candidates display one by one on the command line. In multi-candidate mode, up to ten candidates at a time display in an auxiliary window.

  When the Hanja conversion mode is employed, any Hangul character can be converted into Hanja when the Conversion key is pressed. Similarly, any Hanja word can be converted to the appropriate Hangul word.

  Hanja can also be entered with the Code Input function in the same manner used for entering Hangul.

To allow for these conversions, the following special keys appear on the 106-key Korean keyboard.

Special Korean Keys		
Key Function	**Keysym**	**Description of Function**
Hangul/English toggle key	XK_Hangul	Toggles between Hangul and English modes
Hanja toggle key	XK_Hangul_Hanja	Toggles Hanja mode on and off
Code Input key	XK_Hangul_ Codeinput	Invokes the Code Input function, which allows characters to be entered by their code points
HHC All-Candidate key	XK_Hangul_ MultipleCandidate	Invokes the multi-candidate mode

HHC Conversion key	XK_Hangul_ Conversion	Invokes the single-candidate mode and also scrolls forward through the candidates in both single-candidate and multi-candidate modes
HHC Non-Conversion key	XK_Hangul_ NonConversion	Scrolls backwards through the candidates

Latvian Input Method (LVIM)

The Latvian Input Method (LVIM) is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Latvian keyboard. The features of LVIM are as follows:

- Supports QWERTY and Ergonomic groups, as two main groups. There are two more supplementary groups which are accessible through dead keys from both main groups:
  - Pressing the left-alt key and left-shift key simultaneously, puts keyboard in the Ergonomic group.
  - Pressing the left-alt key and right-shift key simultaneously, puts keyboard in the QWERTY group.
- Supports the IBM-921 code set.

Keymap

The following keymap is supported by the LVIM:

- Lv_LV.IBM-921.imkeymap

Lithuanian Input Method (LTIM)

The Lithuanian Input Method (LTIM) is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Lithuanian keyboard. The features of LTIM are as follows:

- Supports Programmed and Lithuanian groups, as two main groups. There are two more supplementary groups which are accessible through dead keys from both main groups.
  - Pressing the left-alt key and left-shift key simultaneously, puts keyboard in the Lithuanian group.
  - Pressing the left-alt key and right-shift key simultaneously, puts keyboard in the Programmed group.
- Supports the IBM-921 code set.

Keymap:

The following keymap is supported by the LTIM:

- Lt_LT.IBM-921.imkeymap

Thai Input Method (THIM)

The Thai Input Method is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Thai language.

Specifically, it is designed to prevent entry of combinations of Thai characters (consonants, upper/lower vowels, tone marks) that are not valid in the Thai language. The features of the THIM are as follows:

- Supports Latin and Thai groups, as the two main groups on the keyboard.
  - Pressing the left-alt key and left-shift key puts the keyboard in the Thai group.
  - Pressing the left-alt key and right-shift key puts the keyboard in the Latin group.
- Supports the TIS-620 codeset.

Keymap

The following keymap is supported by the THIM:

- th_TH.TIS-620.imkeymap

Vietnamese Input Method

The Vienamese Input Method (VNIM) is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Vietnamese language.

Specifically, it is designed to prevent entry of combinations of Vietnamese characters (tone marks), that are not valid in the Vietnamese language. The Vietnamese tone-mark characters can only be entered immediately after one of the Vietnamese vowels (a, e, i, o, u, y, a-circumflex, e-circumflex, o-circumflex, a-breve, o-horn, or u-horn).

The VNIM supports a single keyboard layer, including some pre-composed characters and Vietnamese tone marks.

The VNIM supports the IBM-1129 codeset.

Keymap

The following keymap is supported by the VNIM:

- Vi_VN.IBM-1129.imkeymap

Simplified Chinese Input Method (ZIM-UCS)

The UCS-2 code set consists of almost all character groups. The following character groups exist for the ZH_CN locale:

- ASCII (English)
- Glyphs
- Chinese, Japanese, and Korean (CJK) Characters (unification characters)

The CJK character set contains 20,992 character positions, but only 20,902 positions are assigned to Chinese characters.

The pronunciation of simplified Chinese is represented by phonetic symbols called Bopomofo. There are 25 phonetic symbols. Simplified Chinese characters are represented by one to three phonetic symbols.

ZIM-UCS features the following characteristics:

- The following commonly used input methods exist:

  Intelligent ABC
  An input method based on the phonetic representation of Chinese characters.

  Pin Yin Input Method
  An input method based on the phonetic representation of Chinese characters. A Chinese character is divided into one or several phonemes according to its pronunciation.

  Wu Bi (Five Strike) Input Method
  An input method based on the grapheme representation of Chinese characters. According to the WuBi grapheme input method, Chinese characters are classified into three levels: stroke, radical and single-character.

  Zheng Ma
  An input method based on the grapheme representation of Chinese word.

Biao Xing Ma Input Method
: An input method in which a Chinese character is divided into several components,or *radicals*. When coding a character, these radicals are presented with the corresponding English letters.

Internal Code Input Method
: An input method in accordance with the code table defined in GB18030 (Chinese Internal Code Specification) and UCS-2 (Unicode System Version 2).

- Half-width and full-width character input. Supports ASCII characters in both single-byte and multibyte modes.
- Auxiliary window to support all the candidate lists. For example, Intelligent ABC generate a list of possible characters that contain the same sound symbols (*radicals*). Users select the desired characters by pressing the conversion key.
- Over-the-spot pre-editing drawing area. Allows entry of radicals in reverse video area that temporarily covers the text line. The complete character is sent to the editor by pressing the conversion key.

The UCS-ZIM files are in the **/usr/lib/nls/loc** directory.

The UCS-ZIM keymap is in the **/usr/lib/nls/loc/ZH_CN.UTF-8.imkeymap** directory.

Chinese (CJK) Character Processing

UCS-ZIM is invoked by pressing one of the input method keys. Each radical or phonetic symbol is assigned to a key. The user inputs radicals or phonetic symbols to an over-the-spot pre-editing area. For internal code input method, a character is generated when the last key is pressed. Other input methods generate a list of candidates that display in a window. The user chooses the desired character by selecting the candidate number. Invalid input generates a beep and an error message. The glyphs can be input using the ABC input method.

Single-Byte Input Method

The Single-Byte Input Method (SIM) is the standard that supports most of the locales. SIM is a mapping function that supports simple composing defined on workstation keyboards associated with single-byte locales. SIM supports any keyboard, code set, and language that the **keycomp** command can describe. You can customize SIM using imkeymaps. The coded strings returned by the input method depend on the imkeymap.

Most single-byte locales share one SIM. The SIM features are as follows:

- Supports 101-key and 102-key keyboard mapping.
- Supports Alt-Numpad composing.

  When you press the Alt key, the input method composes a character by using the next three numeric keys pressed. The three numeric keys represent the decimal encoding of the character. For example, entering the sequence `XK_0`, `XK_9`, `XK_7` maps to the character *a* (097).
- Supports the Num-Lock state for the numeric keypad.
- Supports diacritical composing.

  The e-umlaut key is an example of diacritical composing. To compose e-umlaut, the user presses the appropriate diacritical key (umlaut) followed by an alphabetic key (e). The specific set of diacritical keys in use depend on the locale and keyboard definition. When a space follows a diacritical key, the diacritical character represented by the key is returned if it is in the locale's code set.
- Does not require callback functions.
- Located in the **/usr/lib/nls/loc/sbcs.im** file. Most of the other localized input methods are aliases to this file.

Keymaps

The following keymaps are used by the SIM:

cs_CZ.ISO8859-2.imkeymap
da_DK.ISO8859-1.imkeymap
de_CH.ISO8859-1.imkeymap
de_DE.ISO8859-1.imkeymap
en_GB.ISO8859-1.imkeymap
en_GB.ISO8859-1@alt.imkeymap
en_US.ISO8859-1.imkeymap
es_ES.ISO8859-1.imkeymap
Et_EE.IBM-922 - imkeymap
pl_PL.ISO8859-2@alt.imkeymap
sq_AL.ISO8859-1.imkeymap
fi_FI.ISO8859-1.imkeymap
fi_FI.ISO8859-1@alt.imkeymap
fr_BE.ISO8859-1.imkeymap
fr_CA.ISO8859-1.imkeymap
fr_CH.ISO8859-1.imkeymap
fr_FR.ISO8859-1.imkeymap
fr_FR.ISO8859-1@alt.imkeymap
hr_HR.ISO8859-2.imkeymap
hu_HU.ISO8859-2.imkeymap
is_IS.ISO8859-1.imkeymap
it_IT.ISO8859-1.imkeymap
it_IT.ISO8859-1@alt.imkeymap
nl_BE.ISO8859-1.imkeymap
nl_NL.ISO8859-1.imkeymap
no_NO.ISO8859-1.imkeymap
pl_PL.ISO8859-2.imkeymap
pt_BR.ISO8859-1.imkeymap
pt_PT.ISO8859-1.imkeymap
ro_RO.ISO8859-2.imkeymap
sh_SP.ISO8859-2.imkeymap
sl_SI.ISO8859-2.imkeymap
sk_SK.ISO8859-2.imkeymap
sv_SE.ISO8859-1.imkeymap
sv_SE.ISO8859-1@alt.imkeymap
tr_TR.ISO8859-1.imkeymap

Reserved Keysyms

The following keysyms are unique to this input method and are described in the **/usr/include/X11/aix_keysym.h** file.

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff

XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdieresis	0x180007ae

Modifiers

The following modifiers are used by the SIM:

ShiftMask	0x01
LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10
Mod5Mask (Num Lock)	0x80

Traditional Chinese Input Method (TIM)

The Traditional Chinese code sets consist of the following character groups:

- ASCII (English)
- Traditional Chinese characters

The Traditional Chinese character set contains more than 100,000 characters, but only about 5000 are frequently used. Each character comprises one to five components known as *radicals*.

The pronunciation of Traditional Chinese is represented by phonetic symbols called Dsu-Yin or Bo-Po-Mo-Fo. There are 37 phonetic symbols, as well as four intonation indicators. Chinese characters are represented by one to three phonetic symbols. The character can include one intonation symbol. The omission of an intonation symbol implies a fifth intonation accent.

TIM Features

TIM features the following characteristics:

- The following input methods are used:

  Tsang-Jye
  Supports radicals to generate a character. Most frequently used by data entry personnel.

  Simplified Tsang-Jye
  Supports wildcard input and radicals. Also allows entry of partial characters.

  Phonetic symbols
  Inputs a character based on its pronunciation.

  Internal Code
  Generates characters by EUC hexadecimal, code point input.

  Decimal value
  Generates characters by decimal value. Can be invoked from any of the input modes.
- Half-width and full-width character input. Supports ASCII characters in both single-byte and multibyte modes.
- System-defined and user-definable character input.
- Auxiliary window to support all the candidate lists. Simplified Tsang-Jye and phonetic input methods generate a list of character candidates that contains the same input radicals or sound symbols. Users select characters by pressing the corresponding number.

- Over-the-spot pre-editing drawing area. Allows entry of radicals in reverse video area that temporarily covers the text line. The complete character is sent to the editor by pressing the conversion key.

The TIM file is found in the **/usr/lib/nls/loc/TW.im** directory.

The TIM keymap is found in the **/usr/lib/nls/loc/zh_TW.IBM-eucTW.imkeymap** directory.

Traditional Chinese Character Processing

TIM is invoked by pressing one of the input-method keys. Each radical or phonetic symbol is assigned to a key. The user inputs radicals or phonetic symbols to an over-the-spot pre-editing area. For Tsang-Jye and Internal Code input, a character is generated when the conversion key is pressed. Simplified Tsang-Jye and Phonetic input generate a list of candidates that display in a swindow. The user chooses the desired character by selecting the candidate number. Invalid input generates a beep and an error message.

The following special keys for the Traditional Chinese Input Method are defined on the Traditional Chinese 106-key keyboard.

Special Traditional Chinese Keys		
Key Function	**Keysym**	**Description of Function**
Tsang-Jye Shift key	XK_Chinese _Tsangjei	Invokes both the Tsang-Jye and Simplified Tsang-Jye input methods.
Phonetic Shift key	XK_Chinese _Phonetic	Invokes the Phonetic input method.
Half/Full-Width toggle key	XK_Chinese _Full_Half	Toggles between half-width and full-width.
Conversion key	XK_Convert	Converts radical and phonetic symbols or EUC code symbols into characters. Displays the candidate list in an auxiliary window, if needed.
Non-Conversion key	XK_Non _Convert	Interprets a phonetic symbol as a character.
English/Numeric key	XK_Alph_Num	Invokes ASCII mode.
ALT-Tsang-Jye Shift key	XK_Internal _Code	Invokes Internal Code input method.
ALT plus number keypad		Invoke the decimal value input method.

Universal Input Method

The Universal Input Method is used in the Unicode/UTF-8 locales to provide complete multlingual input method support. Features of the Universal Input Method are as follows:

- Supports Input Method Switching
  – Pressing the `Ctrl` key and the left `Alt` and the letter `i` simultaneously, presents a menu listing the other available input methods. Selecting an input method from the list remaps the keyboard and loads the given input method, allowing character entry using the loaded input method.
- Supports Point and Click Character Input
  – Pressing the `Ctrl` key and the left `Alt` and the letter `l` simultaneously, presents a menu listing the various categories of characters contained in the Unicode standard. Selecting a character list presents a matrix of the available characters from the list. Clicking on a given character will then send that character through the input method to the application.
  – Pressing the `Ctrl` key and the left `Alt` and the letter `c` returns to the application, or if already in the application, returns to the most recently used character list for point and click character entry.
- Supports the UTF-8 code set.

Keymap

XX_XX.UTF-8.imkeymap

Reserved Keysyms

The keysyms listed are reserved for use by the input methods:

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff
XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdieresis	0x180007ae
XK_BunsetsuYomi	0x1800ff05
XK_MaeKouho	0x1800ff04
XK_ZenKouho	0x1800ff01
XK_KanjiBangou	0x1800ff02
XK_HenkanMenu	0x1800ff03
XK_LeftDouble	0x1800ff06
XK_RightDouble	0x1800ff07
XK_LeftPhrase	0x1800ff08
XK_RightPhrase	0x1800ff09
XK_ErInput	0x1800ff0a
XK_Reset	0x1800ff0b

Reserved Keysyms for Traditional Chinese

XK_Full_Size	0xff42
XK_Phonetic	0xff48
XK_Alph_Num	0xaff50
XK_Non_Convert	0xaff52
XK_Convert	0xaff51
XK_Tsang_Jye	0xff47
XK_Internal_Code	0xff4a

Reserved Keysyms for Simplified Chinese (ZIM and ZIM-UCS)

XK_Alph_Num	0xaff47
XK_Non_Convert	0xaff59
XK_Row_Column	0xaff48
XK_PinYin	0xaff49
XK_English_Chinese	0xaff50
XK_ABC	0xaff51
XK_Fivestroke	0xaff62
XK_User-defined	0xaff56

XK_Legend	0xaff55
XK_ABC_Set_Option	0xaff60
XK_Half_full	0xaff53

Related Information

The **IMClose** subroutine, **IMCreate** subroutine, **IMDestroy** subroutine, **IMInitialize** subroutine, **IMInitializeKeymap** subroutine, **IMIoctl** subroutine, **IMFilter** subroutine, **IMLookupString** subroutine, **IMProcessAuxiliary** subroutine, **IMQueryLanguage** subroutine.

Chapter 7. Message Facility

To facilitate translations of messages into various languages and make them available to a program based on a user's locale, it is necessary to keep messages separate from the program by providing them in the form of message catalogs that the program can access at run time. To aid in this task, commands and subroutines are provided by the Message Facility.

Message source files containing application messages are created by the programmer and converted to message catalogs. The application uses these catalogs to retrieve and display messages, as needed. Translating message source files into other languages and then converting the files to message catalogs does not require changing and recompiling a program.

The following information is provided for understanding the Message Facility:

- "Creating a Message Source File"
- "Creating a Message Catalog" on page 155
- "Displaying Messages outside of an Application Program" on page 157

Creating a Message Source File

The Message Facility provides commands and subroutines to retrieve and display program messages located in externalized message catalogs. A programmer creates a message source file containing application messages and converts it to a message catalog with the **gencat** command.

To create a message-text source file, open a file using any text editor. Enter a message identification number or symbolic identifier. Then enter the message text as shown in the following example:

```
1 message-text   $       (This message is numbered)
2 message-text   $       (This message is numbered)
OUTMSG message-text  $   (This message has a symbolic identifier \
                          called OUTMSG)
4 message-text   $       (This message is numbered)
```

Usage Considerations

Consider the following:

- One blank character must exist between the message ID number (or identifier) and the message text.
- A symbolic identifier must begin with an alphabetic character and can contain only letters of the alphabet, decimal digits, and underscores.
- The first character of a symbolic identifier cannot be a digit.
- The maximum length of a symbolic identifier is 64 bytes.
- Message ID numbers must be assigned in ascending order within a single message set, but need not be contiguous. 0 (zero) is not a valid message ID number.
- Message ID numbers must be assigned as if intervening symbolic identifiers are also numbered. If, for example, you had numbered the lines as in the previous example, `1`, `2`, `OUTMSG`, and `3`, the program would contain an error, because the **mkcatdefs** command also assigns numbers to symbolic identifiers, and would have assigned number `3` to the `OUTMSG` symbolic identifier.

  Note: Symbolic identifiers are specific to the Message Facility. Portability of message source files can be affected by the use of symbolic identifiers.

Adding Comments to the Message Source File

You can include a comment anywhere in a message source file except within message text. Leave at least one space or tab (blank) after the $ (dollar sign). The following is an example of a comment:

```
$ This is a comment.
```

Comments do not appear in the message catalog generated from the message source file.

Comments can help developers in the process of maintaining message source files, translators in the process of translation, and writers in the process of editing and documenting messages. Use comments to identify what variables, such as **%s**, **%c**, and **%d,** represent. For example, create a note that states whether the variable refers to a user, file, directory, or flag. Comments also should be used to identify obsolete messages.

For clarity, you should place a comment line directly beneath the message to which it refers, rather than at the bottom of the message catalog. Global comments for an entire set can be placed directly below the **$set** directive.

Continuing Messages on the Next Line

All text following the blank after the message number is included as message text, up to the end of the line. Use the escape character \ (backslash) to continue message text on the following line. The \ (backslash) must be the last character on the line as in the following example:

```
5 This is the text associated with \
message number 5.
```

These two physical lines define the single-line message:

```
This is the text associated with message number 5.
```

Note: The use of more than one blank character after the message number or symbolic identifier is specific to the Message Facility. Portability of message source files can be affected by the use of more than one blank.

Including Special Characters in the Message Text

The \ (backslash) can be used to insert special characters into the message text. These special characters are as follows:

\n Inserts a new-line character.

\t Inserts a horizontal tab character.

\v Inserts a vertical tab character.

\b Inserts a backspace character.

\r Inserts a carriage-return character.

\f Inserts a form-feed character.

**** Inserts a \ (backslash) character.

*****ddd* Inserts a single-byte character associated with the octal value represented by the valid octal digits *ddd*.
Note: One, two, or three octal digits can be specified. However, you must include a leading zero if the characters following the octal digits are also valid octal digits. For example, the octal value for $ (dollar sign) is 44. To display $5.00, use \0445.00, and not \445.00, or the 5 will be parsed as part of the octal value.

\x*dd* Inserts a single-byte character associated with the hexadecimal value represented by the two valid hexadecimal digits *dd*. You must include a leading zero to avoid parsing errors (see the note about *****ddd*).

\x*dddd* Inserts a double-byte character associated with the hexadecimal value represented by the four valid hexadecimal digits *dddd*. You must include a leading zero to avoid parsing errors (see the note about *****ddd*).

Defining a Character to Delimit Message Text

You can use the **$quote** directive in a message source file to define a character for delimiting message text. This character should be an ASCII character. The format is:

```
$quote [character] [comment]
```

Use the specified character before and after the message text. In the following example, the **$quote** directive sets the quote character to _ (underscore), and then disables it before the last message, which contains quotation marks:

```
$quote _   Use an underscore to delimit message text
$set MSFAC        Message Facility - symbolic identifiers
SYM_FORM   _Symbolic identifiers can contain alphanumeric \
characters or the \_ (underscore character)\n_
SYM_LEN    _Symbolic identifiers can be up to 65 \
characters long \n_
5              _You can mix symbolic identifiers and numbers \n_
$quote
MSG_H     Remember to include the _msg_h_ file in your program\n
```

The last **$quote** directive in the previous example disables the underscore character.

In the following example, the **$quote** directive defines ″ (double quotation marks) as the quote character. The quote character must be the first non-blank character following the message number. Any text following the next occurrence of the quote character is ignored.

```
$quote "   Use a double quote to delimit message text
$set 10           Message Facility - Quote command messages
1   "Use the $quote directive to define a character \
\n for delimiting message text"
2   "You can include the \"quote\" character in a message \n \
by placing a \\ in front of it"
3   You can include the "quote" character in a message \n \
by having another character as the first nonblank \
\n character after the message ID number
$quote
4   You can disable the quote mechanism by \n \
using the $quote directive without a character \n\
after it
```

The preceding example illustrates two ways the quote character can be included in message text:

- Place a \ (backslash) in front of the quote character.
- Use some other character as the first non-blank character following the message number. This disables the quote character only for that message.

The preceding example also shows the following:

- A \ (backslash) is still required to split a quoted message across lines.
- To display a \ (backslash) in a message, place another \ (backslash) in front of it.
- You can format a message with a new-line character by using **\n**.
- Using the **$quote** directive with no character argument disables the quote mechanism.

Assigning Message Set Numbers and Message ID Numbers

All message sets require a set number or symbolic identifier. Use the **$set** directive in a source file to give a group of messages a number or identifier:

$set *n* [*comment*]

The message set number is specified by the value of *n*, a number between 1 and **NL_SETMAX**. Instead of a number, you can use a symbolic identifier. All messages following the **$set** directive are assigned to that set number until the next occurrence of a **$set** directive. The default set number is 1. Set numbers must be assigned in ascending order, but need not be in series. Empty sets are created for skipped numbers. However, large gaps in the number sequence can decrease efficiency and performance. Moreover, performance is not enhanced by using more than one set number in a catalog.

You can also include a comment in the **$set** directive, as follows:

```
$set 10  Communication Error Messages

$set OUTMSGS  Output Error Messages
```

Many AIX message sets have a symbolic identifier of the form **MS_***PROG*, where **MS** represents Message Set and *PROG* is the name of the program or utility related to the message set. For example:

```
$set MS_WC  Message Set for the wc Utility

$set MS_XLC1  Message Set 1 for the C For AIX compiler

$set MS_XLC2  Message Set 2 for the C For AIX compiler
```

Removing Messages from a Catalog

The **$delset** directive removes all of the messages belonging to a specified set from an existing catalog:

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

The message set is specified by *n*. The **$delset** directive must be placed in the proper set-number order with respect to any **$set** directives in the same source file. You can also include a comment in the **$delset** directive.

Length of Message Text

The **$len** directive establishes the maximum display length of message text:

`$len [`*n* `[` *comment* `] ]`

If *n* is not specified or if the **$len** directive is not included, the message text display is set to the **NL_TEXTMAX** value. The message-text display length is the maximum number of bytes allowed for a message. Any subsequent specification of a **$len** directive overrides a previous specification. The value of *n* cannot exceed the **NL_TEXTMAX** value.

Content of Message Text

Whenever possible, tell users exactly what has happened and what they can do to remedy the situation. The following example shows how cause and recovery information can improve a message:

```
Original  Message:   Bad arg

Revised Message:   Specify year as a value between 1 and 9999.
```

The message `Bad arg` does not help users much; whereas the message `Do not specify more than 2 files on the command line` tells users exactly what they must do to make the command work. Similarly, the message `Line too long` does not give recovery information to users. The message `Line cannot exceed 20 characters` provides the missing information.

Examples of Message Source Files

1. The following example message source file uses numbers for message ID numbers and for message set numbers:

```
$ This is a message source file sample.
$ Define the Quote Character.
$quote "
$set 1 This is the set 1 of messages.
1 "The specified file does not have read permission on\n"
2 "The %1$s file and the %2$s file are same\n"
3 "Hello world!\n"
$Define the quote character
$quote '
$set 2 This is the set 2 of messages
1       'fieldef: Cannot open %1$s \n'
2       'Hello world\n'
```

2. The following example message source file uses symbolic identifiers for message ID numbers and for message set numbers:

```
$ This is a message source file sample.
$ Define the Quote Character.
$quote "
$set MS_SET1 This is the set 1 of messages.
MSG_1   "The specified file does not have read permission on\n"
MSG_2   "The %1$s file and the %2$s file are same\n"
MSG_3   "Hello world\n"
$Define the quote character
$quote
$set 2 This is the set 2 of messages.
$EMSG_1 'fieldef: Cannot open %1$s\n'
$EMSG_2 'Hello world!\n'
```

3. The following examples show how symbolic identifiers can make the specification of a message more understandable:

```
catgets(cd, 1, 1, "default message")
```

```
catgets(cd, MS_SET1, MSG_1, "default message")
```

Creating a Message Catalog

The Message Facility provides commands and subroutines to retrieve and display program messages located in externalized message catalogs. A programmer creates a message source file containing application messages and converts it to a message catalog. Translating message source files into other languages and then converting the files to message catalogs does not require changing or recompiling a program.

To create a message catalog, process your completed message source file with the message facility's **gencat** command. This command can be used in the following ways:

- Use the **gencat** command to process a message source file containing set numbers, message ID numbers, and message text. Message source files containing symbolic identifiers cannot be processed directly by the **gencat** command. The following example uses the information in the `x.msg` message source file to generate a catalog file:

  ```
  gencat x.cat x.msg
  ```

- Use the **mkcatdefs** command to preprocess a message source file containing symbolic identifiers. The resulting file is then piped to the **gencat** command. The **mkcatdefs** command produces a *SymbolName*_**msg.h** file containing definition statements. These statements equate symbolic identifiers with set numbers and message ID numbers assigned by the **mkcatdefs** command. The *SymbolName*_**msg.h** file should be included in programs using these symbolic identifiers. The **mkcatdefs** command is specific to AIX. The following example uses the information in the **x.msg** message source file to generate the **x_msg.h** header file:

  ```
  mkcatdefs x x.msg
  ```

- Use the **runcat** command to automatically process a source file containing symbolic identifiers. The **runcat** command invokes the **mkcatdefs** command and pipes its output to the **gencat** command. The **runcat** command is specific to AIX. The following example uses the information in the `x.msg` message source file to generate the **x_msg.h** header file and the **X.cat** catalog file:

  ```
  runcat x x.msg
  ```

The preceding example is equivalent to the following example:

```
mkcatdefs x x.msg | gencat x.cat
```

If a message catalog with the name specified by the *CatalogFile* parameter exists, the **gencat** command modifies the catalog according to the statements in the message source files. If a message catalog does not exist, the **gencat** command creates a catalog file with the name specified by the *CatalogFile* parameter.

You can specify any number of message text source files. Multiple files are processed in the sequence you specify. Each successive source file modifies the catalog. If you do not specify a source file, the **gencat** command accepts message source data from standard input.

Catalog Sizing

A message catalog can be virtually any size. The maximum numbers of sets in a catalog, messages in a catalog, and bytes in a message are defined in the **/usr/include/limits.h** file by the following macros:

NL_SETMAX
Specifies the maximum number of set numbers that can be specified by the **$set** directive. If the **NL_SETMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.

NL_MSGMAX
Specifies the maximum number of message ID numbers allowed by the system. If the **NL_MSGMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.

NL_TEXTMAX
Specifies the maximum number of bytes a message can contain. If the **NL_TEXTMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.

Examples

1. This example shows how to create a message catalog from a source file containing message identification numbers. The following is the text of the **hello.msg** message source file:

   ```
   $ file: hello.msg
   $set 1    prompts
   1 Please, enter your name.
   2 Hello, %s \n
   $ end of file: hello.msg
   ```

   To create the **hello.cat** message catalog from the **hello.msg** source file, type:

   ```
   gencat hello.cat hello.msg
   ```

2. This example shows how to create a message catalog from a source file with symbolic references. The following is the text of the **hello.msg** message source file that contains symbolic references to the message set and the messages:

   ```
   $ file: hello.msg
   $quote "
   $set PROMPTS
   PLEASE  "Please, enter your name."
   HELLO   "Hello, %s \n"
   $ end of file: hello.msg
   ```

   The following is the text of the **msgerrs.msg** message source file that contains error messages that can be referenced by their symbolic IDs:

   ```
   $ file: msgerrs.msg
   $quote "
   $set CAT_ERRORS
   MAXOPEN  "Cannot open message catalog %s \n \
   Maximum number of catalogs already open "
   NOT_EX "File %s not executable \n "
   $set MSG_ERRORS
   NOT_FOUND "Message %1$d, Set %2$d not found \n "
   $ end of file: msgerrs.msg
   ```

   To process the **hello.msg** and **msgerrs** message source files, type:

   ```
   runcat hello hello.msg
   runcat msgerrs msgerrs.msg /usr/lib/nls/msg/$LANG/msgerrs.cat
   ```

   The **runcat** command invokes the **mkcatdefs** and **gencat** commands. The first call to the **runcat** command takes the **hello.msg** source file and uses the second parameter, `hello`, to produce the **hello.cat** message catalog and the **hello_msg.h** definition file.

The **hello_msg.h** definition file contains symbolic names for the message catalog and symbolic IDs for the messages and sets. The symbolic name for the **hello.cat** message catalog is `MF_HELLO`. This name is produced automatically by the **mkcatdefs** command.

The second call to the **runcat** command takes the **msgerrs.msg** source file and uses the first parameter, `msgerrs`, to produce the **msgerrs_msg.h** definition file.

Because the third parameter, `/usr/lib/nls/msg/$LANG/msgerrs.cat`, is present, the **runcat** command uses this parameter for the catalog file name. This parameter is an absolute path name that specifies exactly where the **runcat** command must put the file. The symbolic name for the `msgerrs.cat` catalog is `MF_MSGERRS`.

Displaying Messages outside of an Application Program

The following commands allow you to display messages outside of an application program. These commands are specific to AIX.

dspcat
Displays the messages contained in the specified message catalog. The following example displays the messages located in the **x.cat** message source file:

```
dspcat x.cat
```

dspmsg
Displays a single message from a message catalog. The following example displays the message located in the **x.cat** message source file that has the ID number of `1` and the set number of `2`:

```
dspmsg x.cat -s 2 1
```

You can use the **dspmsg** command in shell scripts when a message must be obtained from a message catalog.

Displaying Messages with an Application Program

When programming with the Message Facility, you must include the following items in your application program:

- The *CatalogFile*_**msg.h** definition file created by the **mkcatdefs** or **runcat** command if you used symbolic identifiers in the message source file, or the **limits.h** and **nl_types.h** files if you did not use symbolic identifiers
- A call to initialize the locale environment
- A call to open a catalog
- A call to read a message
- A call to display a message
- A call to close the catalog

The following subroutines provide the services necessary for displaying program messages with the message facility:

setlocale
Sets the locale. Specify the **LC_ALL** or **LC_MESSAGES** environment variable in the call to the **setlocale** subroutine for the preferred message catalog language.

catopen
Opens a specified message catalog and returns a catalog descriptor, which you use to retrieve messages from the catalog.

catgets
Retrieves a message from a catalog after a successful call to the **catopen** subroutine.

printf
Converts, formats, and writes to the stdout (standard output) stream.

catclose
Closes a specified message catalog.

The following C program, `hello`, illustrates opening the `hello.cat` catalog with the **catopen** subroutine, retrieving messages from the catalog with the **catgets** subroutine, displaying the messages with the **printf** subroutine, and closing the catalog with the **catclose** subroutine.

```
/* program: hello */
#include <nl_types.h>
#include <locale.h>
nl_catd catd;
main()
{
/*  initialize the locale  */
setlocale (LC_ALL, "");
/* open the catalog */
catd=catopen("hello.cat",NL_CAT_LOCALE);
printf(catgets(catd,1,1,"Hello World!"));
catclose(catd);                  /* close the catalog */
exit(0);
}
```

In the previous example, the **catopen** subroutine refers to the **hello.cat** message catalog only by file name. Therefore, you must make sure that the **NLSPATH** environment variable is set correctly. If the message catalog is successfully opened by the **catopen** subroutine, the **catgets** subroutine returns a pointer to the specified message in the `hello.cat` catalog. If the message catalog is not found or the message does not exist in the catalog, the **catgets** subroutine returns the `Hello World!` default string.

Understanding the NLSPATH Environment Variable

The **NLSPATH** environment variable specifies the directories to search for message catalogs. The **catopen** subroutine searches these directories in the order specified when called to locate and open a message catalog. If the message catalog is not found, the message-retrieving routine returns the program-supplied default message. See the **/etc/environment** file for the **NLSPATH** default path.

Retrieving Program-Supplied Default Messages

All message-retrieving routines return the program-supplied default message text if the desired message cannot be retrieved for any reason. Program-supplied default messages are generally brief one-line messages that contain no message numbers in the text. Users who prefer these default messages can set the **LC_MESSAGES** category to the C locale or unset the **NLSPATH** environment variable. When none of the **LC_ALL**, **LC_MESSAGES**, or **LANG** environment variables are set, the **LC_MESSAGES** category defaults to the C locale.

Setting the Language Hierarchy

Multilingual users may specify a language hierarchy for message text. To set the language hierarchy for the system default or for an individual user, see “Changing the Language Environment” on page 5, or use SMIT. To use SMIT, to set the language hierarchy, type the SMIT fastpath `smit mlang` at the command line.

Select **Change / Show Language Hierarchy**.

OR

At the command line, type:

```
smit
```

Select **System Environments**.

Select **Manage Language Environment**.

Select **Change / Show Language Hierarchy**.

Example of Retrieving a Message from a Catalog

This example has three parts: the message source file, the command used to generate the message catalog file, and an example program using the message catalog.

1. The following example shows the **example.msg** message source file:

```
$quote "
$ every message catalog should have a beginning set number.
$set MS_SET1
MSG1 "Hello world\n"
MSG2 "Good Morning\n"
ERRMSG1 "example: 1000.220 Read permission is denied for the file
%s.\n"
$set MS_SET2
MSG3 "Howdy\n"
```

2. The following command uses the **example.msg** message source file to generate the **example.h** header file and the **example.cat** catalog file in the current directory:

```
runcat example example.msg
```

3. The following example program uses the **example.h** header file and accesses the **example.cat** catalog file:

```
#include <locale.h>
#include <nl_types.h>
#include "example_msg.h" /*contains definitions for symbolic
                           identifiers*/
main()
{
        nl_catd catd;
        int error;

        (void)setlocale(LC_ALL, "");

        catd = catopen(MF_EXAMPLE, NL_CAT_LOCALE);
        /*
        ** Get the message number 1 from the first set.
        */
        printf( catgets(catd,MS_SET1,MSG1,"Hello world\n") );

        /*
        ** Get the message number 1 from the second set.
        */
        printf( catgets(catd, MS_SET2, MSG3,"Howdy\n") );
        /*
        ** Display an error message.
        */
        printf( catgets(catd, MS_SET1, ERRMSG1,"example: 100.220
                Permission is denied to read the file %s.\n") ,
                filename);
        catclose(catd);
}
```

Writing Messages

The following tips help you make messages meaningful and concise:

- Plan for the internationalization of all messages, including messages that are displayed on panels.
- Allow sufficient space for translated messages to be displayed. Translated messages often occupy more display columns than the original message text. In general, allow about 20% to 30% more space for translated messages, but in some cases, you might need to allow 100% more space for translated messages.
- Use message catalogs to externalize any user and error messages. X applications can use resource files to externalize messages for each locale.
- Provide default messages.

- Make each message in a message source file be a complete entity. Building a message by concatenating parts makes translation difficult.
- Use the **$len** directive in the message source file to control the maximum display length of the message text. (The **$len** directive is specific to the Message Facility.)
- Use symbolic identifiers to specify the set number and message number. Programs should refer to set numbers and message numbers by their symbolic identifiers, not by their actual numbers. (The use of symbolic identifiers is specific to the Message Facility.)
- Facilitate the reordering of sentence clauses by numbering the %**s** variables. This allows the translator to reorder the clauses if needed. For example, if a program needs to display the English message: `The file %s is referenced in %s`, a program may supply the two strings as follows:

  ```
  printf(message_pointer, name1, name2)
  ```

  The English message numbers the %**s** variables as follows:

  ```
  The file %1$s is referenced in %2$s\n
  ```

  The translated equivalent of this message may be:

  ```
  %2$s contains a reference to file %1$s\n
  ```

- Do not use **sys_errlist[errno]** to obtain an error message. This defeats the purpose of externalizing messages. The **sys_errlist[]** is an array of error messages provided only in the English language. Use **strerror**(*errno*) , as it obtains messages from catalogs.
- Do not use **sys_siglist[signo]** to obtain an error message. This defeats the purpose of externalizing messages. The **sys_siglist[]** is an array of error messages provided only in the English language. Use **psignal() ,** as it obtains messages from catalogs.
- Use the message comments facility to aid in the maintenance and translation of messages.
- In general, create separate message source files and catalogs for messages that apply to each command or utility.

Describing Command Syntax in Messages

- Show the command syntax in the usage statement. For example, a possible usage statement for the **rm** command is:

  ```
  Usage: rm [-firRe] [--] File ...
  ```

- Capitalize the first letter of such words as File, Directory, String, and Number in usage statement messages.
- Do not abbreviate parameters on the command line. For example, `Num` spelled out `as Number` can be more easily translated.
- Use only the following delimiters in usage statement messages:

[] Encloses an optional parameter.

{} Encloses multiple parameters, one of which is required.

| Separates parameters that cannot both be chosen. For example, `[a|b]` indicates that you can choose `a`, `b` , or neither `a` nor `b` ; and `{a|b}` indicates that you must choose `a` or `b` .

... Follows a parameter that can be repeated on the command line. Note that there is a space before the ellipsis.

\- Indicates standard input.

- Do not use any delimiters for a required parameter that is the only choice. For example:

  ```
  banner String
  ```

- Put a space character between flags that must be separated on the command line. For example:

  ```
  unget [-n] [-rSID] [-s] {File|-}
  ```

- Do not separate flags that can be used together on the command line. For example:

  ```
  wc [-cwl] {File ...|-}
  ```

- Put flags in alphabetic order when the order of the flags on the command line does not make a difference. Put lowercase flags before uppercase flags. For example:

  ```
  get -aAijlmM
  ```

- Use your best judgment to determine where you should end lines in the usage statement message. The following example shows a lengthy usage statement message:

  ```
  Usage: get [-e|-k] [-c Cutoff] [-i List] [-r SID] [-w String] [-x List] [-b] [-gmnpst] ...
  ```

  Continue the usage information on a second line, if necessary. For example:

  ```
  Usage: get [-e|-k] [-c Cutoff] [-i List] [-r SID] [-w String]
           [-x List] [-b] [-gmnpst] [-l[p]] File ...
  ```

Writing Style for Messages

Clear writing aids in message translation. The following guidelines on the writing style of messages include terminology, punctuation, mood, voice, tense, capitalization, format, and other usage questions.

- Write concise messages. One-sentence messages are preferable.
- Use complete-sentence format.
- Add articles (a, an, the) when necessary to eliminate ambiguity.
- Capitalize the first word of the sentence, and use a period at the end of the sentence.
- Use the present tense. Do not use future tense in a message. For example, use the sentence:

  ```
  The cal command displays a calendar.
  ```

  Instead of:

  ```
  The cal command will display a calendar.
  ```

- Do not use the first person (I or we) in messages.
- Avoid using the second person (you) except in help and interactive text.
- Use active voice. The following example shows how a message written in passive voice can be turned into an active voice message.

  ```
  Passive: Month and year must be entered as numbers.
  Active:  Enter month and year as numbers.
  ```

- Use the imperative mood (command phrase) and active verbs such as specify, use, check, choose, and wait.
- State messages in a positive tone. The following example shows a negative message made more positive.

  ```
  Negative: Don't use the f option more than once.
  Positive: Use the -f flag only once.
  ```

- Use words only in the grammatical categories shown in a dictionary. If a word is shown only as a noun, do not use it as a verb. For example, do not *solution* a problem or *architect* a system.
- Do not use prefixes or suffixes. Translators may not know what words beginning with re-, un-, in-, or non- mean, and the translations of messages that use prefixes or suffixes may not have the meaning you intended. Exceptions to this rule occur when the prefix is an integral part of a commonly used word. For example, the words `previous` and `premature` are acceptable; the word `nonexistent` is not acceptable.
- Do not use parentheses to show singular or plural, as in `error(s)`, which cannot be translated. If you must show singular and plural, write *error or errors*. You may also be able to revise the code so that different messages are issued depending on whether the singular or plural of a word is required.
- Do not use contractions.
- Do not use quotation marks, both single and double quotation marks. For example, do not use quotation marks around variables such as %**s**, %**c**, and %**d** or around commands. Users might interpret the quotation marks literally.
- Do not hyphenate words at the ends of lines.

- Do not use the standard highlighting guidelines in messages, and do not substitute initial or all caps for other highlighting practices. (Standard highlighting includes such guidelines as boldface for commands, subroutines, and files; italics for variables and parameters; typewriter or courier font for examples and displayed text.)
- Do not use the and/or construction. This construction does not exist in other languages. Usually it is better to say `or` to indicate that it is not necessary to do both.
- Use the 24-hour clock. Do not use a.m. or p.m. to specify time. For example, write `1:00 p.m.` as `1300`.
- Avoid acronyms. Only use acronyms that are better known to your audience than their spelled-out version. To make a plural of an acronym, add a lowercase s without an apostrophe. Verify that the acronym is not a trademark before using it.
- Do not construct messages from clauses. Use flags or other means within the program to pass on information so that a complete message may be issued at the proper time.
- Do not use hard-coded text as a variable for a **%s** string in a message.
- End the last line of the message with **\n** (indicating a new line). This applies to one-line messages also.
- Begin the second and remaining lines of a message with **\t** (indicating a tab).
- End all other lines with **\n** (indicating a new line).
- Force a newline on word boundaries where needed so that acceptable message strings display. The **printf** subroutine, which often is used to display the message text, disregards word boundaries and wraps text whenever necessary, sometimes splitting a word in the middle.
- If, for some reason, the message should not end with a newline character, leave writers a comment to that effect.
- Precede each message with the name of the command that called the message, followed by a colon. The following example is a message containing a command name:

  ```
  OPIE "foo: Opening the file."
  ```
- Tell the user to `Press the —— key` to select a key on the keyboard, including the specific key to press. For example:

  ```
  Press the Ctrl-D key
  ```
- Do not tell the user to `Try again later`, unless the system is overloaded. The need to try again should be obvious from the message.
- Use the word ″parameter″ to describe text on the command line, the word ″value″ to indicate numeric data, and the words ″command string″ to describe the command with its parameters.
- Do not use commas to set off the one-thousandth place in values. For example, use `1000` instead of `1,000`.
- If a message must be set off with an * (asterisk), use two asterisks at the beginning of the message and two at asterisks at the end of the message. For example:

  ```
  ** Total **
  ```
- Use the words ″log in″ and ″log off″ as verbs. For example:

  ```
  Log in to the system; enter the data; then log off.
  ```
- Use the words ″user name,″ ″group name,″ and ″login″ as nouns. For example:

  ```
  The user is sam.
  The group name is staff.
  The login directory is /u/sam.
  ```
- Do not use the word ″superuser.″ Note that the root user may not have all privileges.
- Use the following frequently occurring standard messages where applicable:

Preferred Standard Message	Less Desirable Message
`Cannot find or open the file.`	Can't open filename.
`Cannot find or access the file.`	Can't access
`The syntax of a parameter is not valid.`	syntax error

Chapter 8. Culture-Specific Data Handling

Culture-specific data handling may be part of a program, and such a program may supply different data for different locales. In addition, a program may use different algorithms to process character data based on the language and culture. For example, recognition of the start and end of a word and the method of hyphenation of a word across two lines varies depending on the locale. Programs that deal with such functionality need access to these tables or algorithms based on the current locale setting at run time. You can handle such programs in the following ways:

- Compile all the algorithms and tables, and load them with the program.

  This method makes it difficult to add or modify the algorithms and tables. Whenever a new algorithm or table is added, the entire program must be relinked.
- Keep the locale-specific algorithms and tables in a file, and load them at run time, depending on the current locale setting.

  This method makes it easier to modify and add algorithms and tables. However, there is no standard defined way to load algorithms. In AIX, you can achieve this using the **load** subroutine, but programs that use the **load** subroutine might not be portable to other systems.

Culture-Specific Tables

If the culture-specific data can be processed by accessing tables based on the current locale setting, then this can be accomplished by using the standard file I/O subroutines (**fopen**, **fread**, **open**, **read**, and so on). Such tables must be provided in the directory defined in **/usr/lpp/***Name* where *Name* is the name of the particular application under the appropriate locale name.

Standard path prefix
: **/usr/lpp/***Name* (AIX-specific pathname)

Culture-specific directory
: Obtain the current locale for the appropriate category that describes the tables. Concatenate it to the above prefix.

Access
: Use standard file access subroutines (**fopen**, **fread**, and so on) as appropriate.

Culture-Specific Algorithms

The culture-specific algorithms reside in the **/usr/lpp**/*Name***/%L** directory. Here **%L** represents the current locale setting for the appropriate category.

Use the **load** subroutine to access program-specific algorithms from an object module.

Standard path prefix
: **/usr/lpp/***Name*

Culture-specific directory
: Obtain the current locale for the appropriate category. Concatenate it to the above prefix.

Method
: Concatenate the method name to it.

Example of Loading a Culture-Specific Module for Arabic Text for an Application

Header File

The **methods.h** include file has one structure as follows:

```
struct Methods {
        int     version;
        char    *(*hyphen)();
        char    *(*wordbegin)();
        char    *(*wordend)();
} ;
```

Main Program

In this example, the program name is **textpr**.

The main program determines the module to load and invokes it. Note that the **textpr.h** include file is used to specify the path name of the load object. This way, the path name, which is system-specific, can be changed easily.

```
#include    <stdio.h>
#include    <errno.h>
#include    "methods.h"
#include    "textpr.h"    /* contains the pathname where
                             the load object can be found */

extern     int   errno;

main()
{
     char libpath[PATH_MAX];   /* stores the full pathname of the
                                  load object */
      char *prefix_path=PREFIX_PATH;        /* from textpr.h */
      char *method=METHOD;                  /* from textpr.h */
      int (*func)();
      char *path;
      /* Methods */
      int   ver;
      char  *p;
      struct Methods *md;


      setlocale(LC_ALL, "");

      path = setlocale(LC_CTYPE, 0);    /* obtain the locale
                                           for LC_CTYPE category */
      /* Construct the full pathname for the */
      /* object to be loaded                 */
      strcpy(libpath, prefix_path);
      strcat(libpath, path);
      strcat(libpath, "/");
      strcat(libpath, method);

      func = load(conv, 1, libpath);        /* load the object */
      if(func==NULL){
            strerror(errno);
            exit(1);
      }
      /* invoke the loaded module ");
      md =(struct Methods *) func();       /* Obtain the methods
                                              structure */
      ver = md->version;
      /* Invoke the methods as needed */
      p = (md->hyphen)();
      p = (md->wordbegin)();
      p = (md->wordend)();
}
```

Methods

This module contains culture-specific algorithms. In this example, it provides the Arabic method. The `method.c` program follows:

```
#include "methods.h"

char *Arabic_hyphen(char *);
char *Arabic_wordbegin(char *);
char *Arabic_wordend(char *);

static struct Methods ArabicMethods= {
        1,
        Arabic_hyphen,
        Arabic_wordbegin,
        Arabic_wordend
} ;

struct Methods *start_methods()
{
        /* startup methods */
        return ( &ArabicMethods);
}

char *Arabic_hyphen(char *string)
{
        /* Arabic hyphen */
        return( string );
}
char *Arabic_wordbegin(char *string)
{
        /*Arabic word begin */);
        return( string );
}
char *Arabic_wordend(char *string)
{
        /* Arabic word end */;
        return( string);
}
```

Include File

The **textpr** include file contains the path name of the module to be loaded.

```
#define PREFIX_PATH "/usr/lpp/textpr"
                    /* This is an AIX-specific pathname */
```

Layout (Bidirectional Text and Character Shaping) Overview

Bidirectional (BIDI) text results when texts of different direction orientation appear together. For example, English text is read from left to right. Arabic and Hebrew texts are read from right to left. If both English and Hebrew texts appear on the same line, the text is bidirectional. For further information about directional text and character shaping, including a list of available publications, see the following web address:

http://www.opengroup.org

Write bidirectional text according to the following guidelines:

- Arabic and Hebrew words are written from right to left. (A character string is considered a word for the purposes of sequencing in an alphanumeric environment.)
- Numbers and English quotations are written from left to right.
- Digits and their punctuation marks are written from left to right.

Bidirectional script is read from right to left and from top to bottom.

If the embedded text is contained in one line, the text is written from left to right and embedded in the bidirectional text. However, if the embedded text is split between two or more lines, the correct order must be maintained in the left-to-right portions to allow top-to-bottom reading.

For example, right-to-left text embedded in left-to-right text that is contained in one line is written as follows:

```
THERE IS txet lanoitceridib deddebme IN THIS SENTENCE.
```

Right-to-left text embedded in left-to-right text that is split between two lines is written as follows:

```
THERE IS senil owt neewteb tilps si taht txet lanoitceridib deddebme IN THIS SENTENCE.
```

Both texts maintain readability even though the embedded text is split.

Data Streams

Bidirectional text environments use the following data streams:

Visual Data Streams

The system organizes characters in the sequence in which they are presented on the screen.

If a visual data stream is presented from left to right, the first character of the data stream is on the left side of the viewport (screen, window, line, field, and so on). If the same data stream is presented on a right-to-left viewport, the initial character of the data stream is on the right.

If a language of opposite writing orientation is embedded in the visual data stream, the sequence of each text is preserved when the viewport orientation is reversed. For example, (the lowercase text represents bidirectional text) if the keystroke order is :

```
THERE IS bidirectional text IN THIS SENTENCE.
```

then the visual data stream is:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
```

This visual data stream's presentation on a left-to-right viewport is left-justified, as follows:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
-------> <----------------- ---------------->
```

The arrows indicate reading direction.

If you change the viewport orientation to right-to-left, the visual data stream is reversed, right-justified, and unreadable, as follows:

```
.ECNETNES SIHT NI bidirectional text SI EREHT
<---------------- -----------------> <-------
```

Thus, if English text is embedded in Arabic or Hebrew text, both texts are in proper reading order only on a left-to-right viewport. The same is true for Arabic or Hebrew embedded in English. Reversing the viewport orientation makes both texts unreadable.

Logical Data Streams

The system organizes characters in a readable sequence. The bidirectional presentation-management functions arrange text strings in a readable order.

If a logical data stream is presented on a left-to-right viewport, the initial character of the data stream is presented on the left side. If the same data stream is presented on a right-to-left viewport, the initial character of the data stream is presented on the right side, though it is still presented in a readable order.

If a language of opposite writing orientation is embedded in the logical data stream, the orientations of each text are preserved by the bidirectional presentation-management functions. For example, if the keystroke order is:

```
THERE IS bidirectional text IN THIS SENTENCE.
```

then the logical data stream is the same. For example:

```
THERE IS bidirectional text IN THIS SENTENCE.
```

This logical data stream's presentation on a left-to-right viewport (left-justified) is as follows:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
-------> <----------------- ---------------->
```

The logical data stream's presentation on a right-to-left viewport (right-justified) is as follows:

```
IN THIS SENTENCE. txet lanoitceridib THERE IS
----------------> <----------------- ------->
```

The logical data stream is readable on both viewport orientations.

Cursor Movement

Cursor movement on a screen containing bidirectional text is as follows:

Visual
The cursor moves from its current position left or right to the next character, or up or down to the next row. For example, if the cursor is located at the end of the first left-to-right part of a mixed sentence:

```
THERE IS_txet lanoitceridib IN THIS SENTENCE.
```

then, moving the cursor visually to the right causes it to move one character to the right, as follows:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
```

The cursor moves without regard to the contents of the text.

Logical
The cursor moves from its current position to the next or previous character in the data stream. The character may be adjacent to the cursor's position, elsewhere in the same line, or on another line on the screen. Logical cursor movement requires scanning the data stream to find the next logical character. For example, if the cursor is located at the end of the first left-to-right part of a mixed sentence:

```
THERE IS_txet lanoitceridib IN THIS SENTENCE.
```

then, moving the cursor logically to the next character causes the data stream to be scanned to find the next logical character. The cursor moves to the next logical part of the sentence, as follows:

```
THERE IS txet lanoitceridib_IN THIS SENTENCE.
```

The cursor moves according to content.

Character Shaping

Character shaping occurs when the shape of a character is dependent on its position in a line of text. In some languages, such as Arabic, characters have different shapes depending on their position in a string and on the surrounding characters.

The following characteristics determine character shaping in Arabic script:

- The written language has no equivalent to capital letters.
- The characters have different shapes, depending on their position in a string and on the surrounding characters.
- The written language is cursive. Most characters of a word are connected, as in English handwriting.
- Joined characters can form nonspacing characters. Additionally, a character can have a vowel or diacritic mark written over or under it.
- Characters can vary in length, resulting in an output of two coded shapes.

Methods of Character Shaping

Implement character shaping separately from other system components. However, character shaping should be accessible as a utility by other system components. The system may use character shaping in the following ways:

- As the user enters data into the computer, the system uses character shaping to shape the characters. The system stores these characters in their shaped format.

  This method avoids the need to use character shaping every time these characters are displayed. This method is meant for static data such as menus and help. This method requires preprocessing for correct sorting, searching, or indexing of the characters.

  The characters may need reshaping after processing for proper presentation.
- As the user enters data into the computer, the system stores the characters in their unshaped format.

  This method allows for sorting, searching or indexing of the characters. However, the system must use character shaping every time the characters are displayed.

Base shapes are isolated shapes that were not generated by character shaping. Use base shapes during editing, searching for character strings, or other text operations. Use shaping only when the text is displayed or printed. If characters are stored in their shaped form, the system must deshape them before sorting, collating, searching, or indexing. Character shapes that are not shape-determined according to their position in a string are needed for specific character-handling applications, as well as for communication with different coding environments.

Contextual Character Shaping

In general, contextual character shaping is the selection of the required shape of a character in a given font depending on its position in a word and its surrounding characters. The following shapes are possible:

Isolated	A character that is connected to neither a preceding nor succeeding character
Final	A character that is connected to a preceding character but not with a succeeding character
Initial	A character connected to a succeeding character but not with a preceding character
Middle	A character connected to both a preceding and succeeding character

A character may also have any of the following characteristics:

- Connecting to a preceding character
- Connecting to a succeeding character
- Allowing surrounding characters' connections to pass through it

Acronyms, part numbers, and graphic characters do not need contextual character shaping. To properly enter these characters, turn off the contextual character shaping and use a specific keyboard interface for exact selection of the desired shape. Tag these characters by field, line, or control character for later presentation.

Appendix A. National Language Support (NLS) Reference

This reference provides the following information:

- “National Language Support Checklist”
- “List of National Language Support Subroutines” on page 174

National Language Support Checklist

The National Language Support (NLS) Checklist provides a way to analyze a program for NLS dependencies. By going through this list, one can determine what, if any, NLS functions must be considered. This is useful for both programming and testing. If you identify a set of NLS items that a program depends on, a test strategy can be developed. This facilitates a common approach to testing all programs.

All major NLS considerations have been identified. However, this list is not all-encompassing. There may be other NLS questions that are not listed.

Program Operation Checklist

1. Does the program display translatable messages to the user, either directly or indirectly? An example of indirect messages are those that are stored in libraries.

   If yes:

   - Are these messages externalized from the program by way of the message facility subroutines?
   - Have you provided message source files for all such messages?
   - What is the locale under which the program runs?
     - If it runs in the locale determined by the locale environment variables, did you invoke the **setlocale** subroutine in the following manner?

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

       Note: See “Setting the Locale” on page 15 for **setlocale** subroutine examples. The locale categories, in their predefined hierarchical order, are: **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME**. See “Understanding Locale Environment Variables” on page 9 for more information on the **LC_ALL** category.
     - If the program runs in the ″C″ locale, except for displaying messages in the locale specified by the locale environment variables, did you invoke the **setlocale** subroutine in the following manner?

       ```
       setlocale(LC_MESSAGES, "")
       ```
   - After invoking the **setlocale** subroutine, did you invoke the **catopen** subroutine in the following manner?

     ```
     catopen(catalog_name, NL_CAT_LOCALE)
     ```
   - Did you invoke the **catopen** subroutine with the proper catalog name?
   - See the Chapter 7, “Message Facility,” on page 151 for more information about translatable messages.
2. Does the program compare text strings?

   If yes:

   - Are the strings compared to check equality only?

     If yes:

     - Use the **strcmp** or **strncmp** subroutine.
     - Do not use the **strcoll** or **strxfrm** subroutine.

- Are the strings compared to see which one sorts before the other, as defined in the current locale?

  If yes:

  – Invoke the **setlocale** subroutine in the following manner:

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

  – Use the **strcoll**, **strxfrm**, **wcscoll**, or **wcsxfrm** subroutine.

  – Do not use the **strcmp** or **strncmp** subroutine.

3. Does the program parse path names of files?

   If yes:

   - If looking for / (slash), use the **strchr** subroutine.
   - If looking for characters, be aware that the file names can include multibyte characters. In such cases, invoke the **setlocale** subroutine in the following manner and then use appropriate search subroutines:

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

4. Does the program use system names, such as node names, user names, printer names, and queue names?

   If yes:

   - System names can have multibyte characters.
   - To identify a multibyte character, first invoke the **setlocale** subroutine in the following manner and then use appropriate subroutines in the library.

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

5. Does the program use character class properties, such as uppercase, lowercase, and alphabetic?

   If yes:

   - Invoke the **setlocale** subroutine in the following manner:

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

   - Do not make assumptions about character properties. Always use system subroutines to determine character properties.
   - Are the characters restricted to single-byte code sets?

     If yes:

     – Use one of the **ctype** subroutines: **isalnum**, **isalpha**, **iscntrl**, **isdigit**, **isgraph**, **isprint**, **isspace**, or **isxdigit**.

       If not, the characters may be multibyte characters:

     – Use the **iswalnum**, **iswalpha**, **iswcntrl**, **iswdigit**, **iswgraph**, **iswlower**, **iswprint**, **iswpunct**, **iswspace**, **iswupper**, or **iswxdigit** subroutine. See “Wide Character Classification Subroutines” on page 28 for more information.

6. Does the program convert the case (upper or lower) of characters?

   If yes:

   - Invoke the **setlocale** subroutine in the following manner:

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

   - Are the characters restricted to single-byte code sets?

     If yes:

     – Use these **conv** subroutines: **_tolower**, **_toupper**, **tolower**, or **toupper**.

     If not, the characters may be multibyte characters:

     – Use the **towlower** or **towupper** subroutine. See “Wide Character Classification Subroutines” on page 28 for more information.

7. Does the program keep track of cursor movement on a tty terminal?

   If yes:

   - Invoke the **setlocale** subroutine in the following manner:

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

- You may need to determine the display column width of characters. Use the **wcwidth** or **wcswidth** subroutine.

8. Does the program perform character I/O?

   If yes:

   - Invoke the **setlocale** subroutine in the following manner:

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

   - Are the characters restricted to single-byte code sets?

     If yes:

     – Use following subroutine families:
       - **fgetc**, **getc**, **getchar**, **getw**
       - **fgets**, **gets**
       - **fputc**, **putc**, **putchar**, **putw**
       - **printf**, **scanf**

     If not:

     – Use following subroutine families:
       - **fgetwc**, **getwc**, **getwchar**
       - **fgetws**, **getws**
       - **fputwc**, **putwc**, **putwchar**

9. Does the program step through an array of characters?

   If yes:

   - Is the array limited to single-byte characters only?

     If yes:

     – Does not require **setlocale(LC_ALL,** *""***)**

     – If p is the pointer to this array of single-byte characters, step through this array using p++.

   If not:

   - Invoke the **setlocale** subroutine in the following manner:

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

   - Use the **mblen** or **wcslen** subroutine.

10. Does the program need to know the maximum number of bytes used to encode a character within the code set?

    If yes:

    - Invoke the **setlocale** subroutine in the following manner:

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

    - Use the **MB_CUR_MAX** macro.

11. Does the program format date or time numeric quantities?

    If yes:

    - Invoke the **setlocale** subroutine in the following manner:

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

    - Use the **nl_langinfo** or **localeconv** subroutine to obtain the locale-specific information.
    - Use the **strftime** or **strptime** subroutine.
    - See “Setting the Locale” on page 15 and “Euro Currency Support” on page 21 for more information.

12. Does the program format numeric quantities?

    If yes:

- Invoke the **setlocale** subroutine in the following manner:

  ```
  setlocale(LC_ALL, "")
  ```
- Use the **nl_langinfo** or **localeconv** subroutine to obtain the locale-specific information.
- Use the following pair of subroutines, as needed: **printf**, **scanf.**

13. Does the program format monetary quantities?

    If yes:

    - Invoke the **setlocale** subroutine in the following manner:

      ```
      setlocale(LC_ALL, "")
      ```
    - Use the **nl_langinfo** or **localeconv** subroutine to obtain the locale-specific information.
    - Use the **strfmon** subroutine to format monetary quantities.
    - See "Setting the Locale" on page 15 and "Euro Currency Support" on page 21 for more information.
14. Does the program search for strings or locate characters?

    If yes:

    - Are you looking for single-byte characters in single-byte text?
      - Does not require **setlocale(LC_ALL,** *""***)**
      - Use standard **libc** string subroutines such as the **strchr** subroutine.
    - Are you looking for characters in the range 0x00-0x3F (the unique code-point range)?
      - Does not require **setlocale(LC_ALL,** *""***)**
      - Use standard **libc** string subroutines such as the **strchr**, **strcspn**, **strpbrk**, **strrchr**, **strspn**, **strstr**, **strtok**, and **memchr** subroutines.
    - Are you looking for characters in the range 0x00-0xFF?
      - Invoke the **setlocale** subroutine in the following manner:

        ```
        setlocale(LC_ALL, "")
        ```
      - Two methods are available:

        Use the **mblen** subroutine to skip multibyte characters. Then, on encountering single-byte characters, check for equality. See checklist item 2.

        OR

        Convert the search character and the searched string to wide character form, and then use wide character search subroutines. See "Wide Character String Search Subroutines" on page 37 for more information.
15. Does the program perform regular-expression pattern matching?

    If yes:

    - Invoke the **setlocale** subroutine in the following manner:

      ```
      setlocale(LC_ALL, "")
      ```
    - Use the **regcomp**, **regexec**, or **regerror** subroutine.
16. Does the program ask the user for affirmative/negative responses?

    If yes:

    - Invoke the **setlocale** subroutine in the following manner:

      ```
      setlocale(LC_ALL, "")
      ```
    - Put the prompt in the message catalog. Use the **catopen** and **catgets** subroutines to retrieve the catalog and display the prompt.
    - Use the **rpmatch** subroutine to match the user's response.
    - See the Chapter 7, "Message Facility," on page 151 for more information.
17. Does the program use special box-drawing characters?

    If yes:

- Do not use code set-specific box-drawing characters.
- Instead use the box-drawing characters and attributes specified in the **terminfo** file.

18. Does the program perform culture-specific or locale-specific processing that is not addressed here?

    If yes:

    - Externalize the culture-specific modules. Do not make them part of the executable program.
    - Load the modules at run time using subroutines provided by the system, such as the **load** subroutine.
    - If the system does not provide such facilities, link them statically but provide them in a modular fashion.

AIXwindows Checklist

The remaining checklist items are specific to the AIXwindows systems.

1. Does your program use the font set specification in order to be code-set independent in X applications?
2. Does your client use labels, buttons, or other output-only widgets to display translatable messages? If yes:
   - Invoke the *****XtSetLanguageProc** subroutine in the following manner:

     ```
     XtSetLanguageProc(NULL, NULL, NULL);
     ```
   - Messages can be placed in either message catalogs or localized resource files. See checklist items 1 or 20, respectively.
   - To make the widgets code set-independent, specify fonts that use font sets.
3. Does your client use X resource files to define the text of labels, buttons, or text widgets?

   If yes:

   - Put all resources that need translation in one place.
   - Consider using message catalogs for the text strings. See the Chapter 7, "Message Facility," on page 151 for more information.
   - Do not use translated color names, since color names are restricted to one encoding. The only portable names are encoded in the portable character set.
   - Put language-specific resource files in **/usr/lib/X11/%L/app-defaults/%N**, where **%L** is the name of the locale, such as fr_FR, and **%N** is the name of the client.
4. Is keyboard input localized by language?

   If yes:

   - Invoke the *****XtSetLanguageProc** subroutine in the following manner:

     ```
     XtSetLanguageProc(NULL, NULL, NULL);
     ```
   - Use the **XmText** or **XmTextField** widgets for all text input.

     Some of the **XmText** widgets' arguments are defined in terms of character length instead of byte length. The cursor position is maintained in character position, not byte position.
   - Are you using the **XmDrawingArea** widget to do localized input?
     - Use the input method subroutines to do input processing in different languages. See the Chapter 6, "Input Methods," on page 123 and the **IMAuxDraw** Callback subroutine for more information.
5. Does your client present lists or labels consisting of localized text from user files rather than from X resource files?

   If yes:

   - Invoke the *****XtSetLanguageProc** subroutine in the following manner:

     ```
     XtSetLanguageProc(NULL, NULL, NULL);
     ```
   - Use the **XmStringCreateSimple** subroutine to create the **XmString** data type for localized text. The **XmStringCreate** subroutine can be used, but **XmSTRING_DEFAULT_CHARSET** is preferable.

- To make the widgets code-set independent, specify fonts by using font sets. Font resources (for example, *****fontList:** instead) in the app-defaults files should use the upper case and class form rather than the lower case form (for example, *****FontList:** instead). This allow the desktop style manager to affect the application font selection.

6. Does your program do any presentation operations (Xlib drawing, printing, formatting, or editing) on bidirectional text?

   If yes:

   - Use the **XmText** or **XmTextField** in the Xm (Motif) library. These widgets are enabled for bidirectional text. See ″Layout (Bidirectional) Support in Xm (Motif) Library″ in *AIX 5L Version 5.2 AIXwindows Programming Guide* for more information.
   - If the Xm library can not be used, use the layout subroutines to perform any re-ordering and shaping on the text.
   - Store and communicate the text in the implicit (logical) form. Some utilities (for example, **aixterm**) support the visual form of bidirectional text, but most NLS subroutines can not process the visual form of bidirectional text.

If the response to all the above items is no, then the program probably has no NLS dependencies. In this case, you may not need the locale-setting subroutine **setlocale** and the catalog facility subroutines **catopen** and **catgets**.

List of National Language Support Subroutines

The National Language Support (NLS) subroutines are used for handling locale-specific information, manipulating wide characters and multibyte characters, and using regular expressions.

For more information about NLS subroutines, see Chapter 3, “Subroutines for National Language Support,” on page 15.

List of Locale Subroutines

The following subroutines are provided to obtain and process locale-specific data:

localeconv	Retrieves locale-dependent conventions of a program locale.
nl_langinfo	Returns information on language or cultural area in a program locale.
rpmatch	Determines whether a response is affirmative or negative in the current locale.
setlocale	Changes or queries a program’s current locale.

For more NLS subroutines see “List of National Language Support Subroutines.”

List of Time and Monetary Formatting Subroutines

strfmon	Formats monetary strings according to the current locale.
strftime	Formats time and date according to the current locale.
strptime	Converts a character string to a time value according to the current locale.
wcsftime	Converts time and date into a wide character string according to the current locale.

For more information about NLS subroutines see Chapter 3, “Subroutines for National Language Support,” on page 15.

For more NLS subroutines see “List of National Language Support Subroutines.”

List of Multibyte Character Subroutines

mblen Determines the length of a multibyte character.
mbstowcs Converts a multibyte character string to a wide character string.
mbtowc Converts a multibyte character to a wide character.

For more information about multibyte character subroutines see Chapter 3, "Subroutines for National Language Support," on page 15.

For more NLS subroutines see "List of National Language Support Subroutines" on page 174.

List of Wide Character Subroutines

The following subroutines process characters in process-code form:

fgetwc Gets a wide character or word from an input stream.
fgetws Gets a wide character string from a stream.
fputwc Writes a wide character or a word to a stream.
fputws Writes a wide character string to a stream.
getwc Gets a wide character or word from an input stream.
getwchar Gets a wide character or word from an input stream.
getws Gets a wide character string from a stream.
iswalnum Determines if the wide character is alphanumeric.
iswalpha Determines if the wide character is alphabetic.
iswcntrl Determines if the wide character is a control character.
iswctype Determines the property of a wide character.
iswdigit Determines if the wide character is a digit.
iswgraph Determines if the wide character (excluding ″space characters″) is a printing character.
iswlower Determines if the wide character is lowercase.
iswprint Determines if the wide character (including ″space characters″) is a printing character.
iswpunct Determines if the wide character is a punctuation character.
iswspace Determines if the wide character is a blank space.
iswupper Determines if the wide character is uppercase.
iswxdigit Determines if the wide character is a hexadecimal digit.
putwc Writes a wide character or a word to a stream.
putwchar Writes a wide character or a word to a stream.
putws Writes a wide character string to a stream.
strcoll Compares two strings based on their collation weights in the current locale.
strxfrm Transforms a string into locale collation values.
towlower Converts an uppercase wide character to a lowercase wide character.
towupper Converts a lowercase wide character to an uppercase wide character.
ungetwc Pushes a wide character onto a stream.
wcsid Returns the charsetID of a wide character.
wcscat Concatenates wide character strings.
wcschr Searches for a wide character.
wcscmp Compares wide character strings.
wcscoll Compares the collation weights of wide character strings.
wcscpy Copies a wide character string.
wcscspn Searches for a wide character string.
wcslen Determines the number of characters in a wide character string.
wcsncat Concatenates a specified number of wide characters.
wcsncmp Compares a specified number of wide characters.
wcsncpy Copies a specified number of wide characters.
wcspbrk Locates the first occurrence of wide characters in a wide character string.
wcsrchr Locates the last occurrence of wide characters in a wide character string.

wcsspn	Returns the number of wide characters in the initial segment of a string.
wcstod	Converts a wide character string to a double-precision floating point value.
wcstok	Breaks a wide character string into a sequence of separate wide character strings.
wcstol	Converts a wide character string to a long integer value.
wcstombs	Converts a sequence of wide characters to a sequence of multibyte characters.
wcstoul	Converts a wide character string to an unsigned, long integer value.
wcswcs	Locates the first occurrence of a wide character sequence in a wide character string.
wcswidth	Determines the display width of a wide character string.
wcsxfrm	Converts a wide character string to values representing character collation weights.
wctomb	Converts a wide character to a multibyte character.
wctype	Gets a handle for valid property names as defined in the current locale.
wcwidth	Determines the display width of a wide character.

For more information about wide character subroutines see Chapter 3, "Subroutines for National Language Support," on page 15.

For more NLS subroutines see "List of National Language Support Subroutines" on page 174.

List of Layout Library Subroutines

The following subroutines of the Layout library (**libi18n.a**) transform bidirectional and context-dependent text to different formats:

layout_object_create	Initializes a layout context.
layout_object_free	Frees a **LayoutObject** structure.
layout_object_editshape	Edits the shape of the context text.
layout_object_getvalue	Queries the current layout values of a **LayoutObject** structure.
layout_object_setvalue	Sets the layout values of a **LayoutObject** structure.
layout_object_shapeboxchars	Shapes box characters.
layout_object_transform	Transforms the text according to the current layout values of a **LayoutObject** structure.

For more information about Layout library subroutines see Chapter 3, "Subroutines for National Language Support," on page 15.

For more NLS subroutines see "List of National Language Support Subroutines" on page 174.

List of Message Facility Subroutines

The Message Facility consists of standard defined subroutines and commands, and manufacturer value-added extensions to support externalized message catalogs. These catalogs are used by an application to retrieve and display messages, as needed. The following Message Facility subroutines get messages for an application:

catopen	Opens a catalog.
catgets	Gets a messages from a catalog.
catclose	Closes a catalog.
strerror	Maps an error number to an error-message string appropriate for the current locale.

For more information about multibyte character subroutines see Chapter 3, "Subroutines for National Language Support," on page 15.

For more NLS subroutines see "List of National Language Support Subroutines" on page 174.

List of Converter Subroutines

In an internationalized environment, it is often necessary to convert data from one code set to another. The following converter subroutines are supported for this purpose:

iconv_open	Performs the initialization required to convert characters from the code set specified by the *FromCode* parameter to the code set specified by the *ToCode* parameter.
iconv	Invokes the converter function using the descriptor obtained from the **iconv_open** subroutine.
iconv_close	Closes the conversion descriptor specified by the *cd* variable and makes it usable again.
ccsidtocs	Returns the code-set name of the corresponding coded character set IDs (CCSID).
cstoccsid	Returns the CCSID of the corresponding code-set name.

For more information about multibyte character subroutines see Chapter 3, “Subroutines for National Language Support,” on page 15.

For more NLS subroutines see “List of National Language Support Subroutines” on page 174.

List of Input Method Subroutines

The Input Method is a set of subroutines that translate key strokes into character strings in the code set specified by a locale. The Input Method subroutines include logic for locale-specific input processing and keyboard controls (for example, Ctrl, Alt, Shift, Lock, and Alt-Graphic). The following subroutines support this Input Method:

IMAIXMapping	Translates a pair of *KeySymbol* and *State* parameters to a string and returns a pointer to that string.
IMAuxCreate	Tells the application program to create an auxiliary area.
IMAuxDestroy	Notifies the callback to destroy any knowledge of the auxiliary area.
IMAuxDraw	Tells the application program to draw the auxiliary area.
IMAuxHide	Tells the application program to hide the auxiliary area.
IMBeep	Tells the application program to emit a beep sound.
IMClose	Closes the input method.
IMCreate	Creates one instance of a particular input method.
IMDestroy	Destroys an input method instance.
IMFilter	Checks whether a keyboard event is used by the input method for its internal processing.
IMFreeKeymap	Frees resources allocated by the **IMInitialzieKeymap** subroutine.
IMIndicatorDraw	Tells the application program to draw the indicator.
IMIndicatorHide	Tells the application program to hide the indicator.
IMInitialize	Initializes the input method for a particular language.
IMInitializeKeymap	Initializes the input method for a particular language.
IMIoctl	Performs a variety of control or query operations on the input method.
IMLookupString	Maps a keyboard-symbol/state pair to a string defined by the user.
IMProcessAuxiliary	Notifies the input method of input for an auxiliary area.
IMQueryLanguage	Checks to see if the specified language is supported.
IMSimpleMapping	Translates a pair of *KeySymbol* and *State* parameters to a string a returns a pointer to that string.
IMTextCursor	Sets the new display cursor position.
IMTextDraw	Asks the application program to draw the next string.
IMTextHide	Tells the application program to hide the text area.
IMTextStart	Notifies the application program of the length of the pre-editing space.
IMTextStart	Notifies the application program of the length of the pre-editing space.

List of Regular Expression Subroutines

The following subroutines handle regular expressions:

regcomp Compiles a regular expression for comparison by the **regexec** subroutine.

For more information about multibyte character subroutines see Chapter 3, “Subroutines for National Language Support,” on page 15.

For more NLS subroutines see “List of National Language Support Subroutines” on page 174.

Appendix B. Character Maps

This appendix contains textual representations of the following character maps discussed in Chapter 4, “Code Sets for National Language Support,” on page 49:

- “ISO Code Sets”
- “IBM Code Sets” on page 197

ISO Code Sets

The following ISO code sets are described:

- “ISO8859–1”
- “ISO8859–2” on page 182
- “ISO8859–5” on page 184
- “ISO8859–6” on page 187
- “ISO8859–7” on page 188
- “ISO8859–8” on page 190
- “ISO8859–9” on page 192
- “ISO8859–15” on page 194

ISO8859–1

Table 1. ISO8859–1 Code set

Symbolic Name	Hex Value
no break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
currency sign	A4
yen sign	A5
broken bar	A6
section sign	A7
diaeresis	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
acute accent	B4

Table 1. ISO8859–1 Code set (continued)

Symbolic Name	Hex Value
micro sign	B5
pilcrow sign	B6
middle dot	B7
cedilla	B8
superscript one	B9
masculine ordinal indicator	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter E with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter eth	D0
latin capital letter n with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circumflex	DB
latin capital letter U with diaeresis	DC

Table 1. ISO8859–1 Code set (continued)

Symbolic Name	Hex Value
latin capital letter Y with acute	DD
latin capital letter thorn	DE
latin small letter sharp S	DF
latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter eth	F0
latin small letter n with tilde	F1
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter thorn	FE
latin small letter y with diaeresis	FF

ISO8859–2

Table 2. ISO8859–2 Code set

Symbolic Name	Hex Value
no break space	A0
latin capital letter A with ogonek	A1
bleve	A2
capital letter L with stroke	A3
currency sign	A4
latin capital letter L with caron	A5
latin capital letter S with acute	A6
section sign	A7
diaeresis	A8
latin capital letter S with caron	A9
latin capital letter S with cedilla	AA
latin capital letter T with caron	AB
latin capital letter Z with acute	AC
soft hyphen	AD
latin capital letter Z with caron	AE
latin capital letter Z with dot above	AF
degree sign	B0
latin small letter A with ogenek	B1
ogenek	B2
latin small letter L with stroke	B3
acute accent	B4
latin small letter L with caron	B5
latin small letter S with acute	B6
caron	B7
cedilla	B8
latin small letter S with caron	B9
latin small letter S with cedilla	BA
latin small letter T with caron	BB
latin small letter Z with acute	BC
double acute accent	BD
latin small letter Z with caron	BE
latin small letter Z with dot above	BF
latin capital letter R with acute	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with breve	C3
latin capital letter A with diaeresis	C4
latin capital letter L with acute	C5

Table 2. ISO8859–2 Code set (continued)

Symbolic Name	Hex Value
latin capital letter C with acute	C6
latin capital letter C with cedilla	C7
latin capital letter C with caron	C8
latin capital letter E with acute	C9
latin capital letter E with ogonek	CA
latin capital letter E with diaeresis	CB
latin capital letter E with caron	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter D with caron	CF
latin capital letter D with stroke	D0
latin capital letter N with acute	D1
latin capital letter N with caron	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with double acute	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter R with caron	D8
latin capital letter U with ring above	D9
latin capital letter U with acute	DA
latin capital letter U with double acute	DB
latin capital letter U with diaeresis	DC
latin capital letter Y with acute	DD
latin capital letter T with cedilla	DE
latin small letter sharp S	DF
latin small letter R with acute	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with breve	E3
latin small letter A with diaeresis	E4
latin small letter L with acute	E5
latin small letter C with acute	E6
latin small letter C with cedilla	E7
latin small letter C with caron	E8
latin small letter E with acute	E9
latin small letter E with ogonek	EA
latin small letter E with diaeresis	EB
latin small letter E with caron	EC
latin small letter I with acute	ED

Table 2. ISO8859–2 Code set (continued)

Symbolic Name	Hex Value
latin small letter I with circumflex	EE
latin small letter D with caron	EF
latin small letter D with stroke	F0
latin small letter N with acute	F1
latin small letter N with caron	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with double acute	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter R with caron	F8
latin small letter U with ring above	F9
latin small letter U with acute	FA
latin small letter Uwith double acute	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter T with cedilla	FE
dot above	FF

ISO8859–5

Table 3. ISO8859–5 Code set

Symbolic Name	Hex Value
no break space	A0
cyrillic capital letter io	A1
cyrillic capital letter dje	A2
cyrillic capital letter gje	A3
cyrillic capital letter ukrainian ie	A4
cyrillic capital letter dze	A5
cyrillic capital letter byelorussian-ukrainian I	A6
cyrillic capital letter yi	A7
cyrillic capital letter je	A8
cyrillic capital letter lje	A9
cyrillic capital letter nje	AA
cyrillic capital letter tshe	AB
cyrillic capital letter kje	AC
soft hyphen	AD
cyrillic capital letter short U	AE
cyrillic capital letter dzhe	AF
cyrillic capital letter A	B0

Table 3. ISO8859–5 Code set (continued)

Symbolic Name	Hex Value
cyrillic capital letter be	B1
cyrillic capital letter ve	B2
cyrillic capital letter ghe	B3
cyrillic capital letter de	B4
cyrillic capital letter ie	B5
cyrillic capital letter zhe	B6
cyrillic capital letter ze	B7
cyrillic capital letter I	B8
cyrillic capital letter short I	B9
cyrillic capital letter ka	BA
cyrillic capital letter el	BB
cyrillic capital letter em	BC
cyrillic capital letteren	BD
cyrillic capital letter O	BE
cyrillic capital letter pe	BF
cyrillic capital letter er	C0
cyrillic capital letter es	C1
cyrillic capital letter te	C2
cyrillic capital letter U	C3
cyrillic capital letter ef	C4
cyrillic capital letter ha	C5
cyrillic capital letter tse	C6
cyrillic capital letter che	C7
cyrillic capital letter sha	C8
cyrillic capital letter shcha	C9
cyrillic capital letter hard sign	CA
cyrillic capital letter yeru	CB
cyrillic capital letter soft sign	CC
cyrillic capital letter E	CD
cyrillic capital letter tu	CE
cyrillic capital letter ya	CF
cyrillic small letter A	D0
cyrillic small letter be	D1
cyrillic small letter ve	D2
cyrillic small letter ghe	D3
cyrillic small letter de	D4
cyrillic small letter ie	D5
cyrillic small letter zhe	D6
cyrillic small letter ze	D7
cyrillic small letter I	D8

Table 3. ISO8859–5 Code set (continued)

Symbolic Name	Hex Value
cyrillic small letter short I	D9
cyrillic small letter ka	DA
cyrillic small letter el	DB
cyrillic small letter em	DC
cyrillic small letter en	DD
cyrillic small letter O	DE
cyrillic small letter pe	DF
cyrillic small letter er	E0
cyrillic small letter es	E1
cyrillic small letter te	E2
cyrillic small letter U	E3
cyrillic small letter ef	E4
cyrillic small letter ha	E5
cyrillic small letter tse	E6
cyrillic small letter che	E7
cyrillic small letter sha	E8
cyrillic small letter shcha	E9
cyrillic small letter hard sign	EA
cyrillic small letter yeru	EB
cyrillic small letter soft sign	EC
cyrillic small letter E	ED
cyrillic small letter yu	EE
cyrillic small letter ta	EF
numero sign	F0
cyrillic small letter io	F1
cyrillic small letter dje	F2
cyrillic small letter gje	F3
cyrillic small letter ukrainian ie	F4
cyrillic small letter dze	F5
cyrillic small letter byelorussian-ukrainian I	F6
cyrillic small letter yi	F7
cyrillic small letter je	F8
cyrillic small letter lje	F9
cyrillic small letter nje	FA
cyrillic small letter tshe	FB
cyrillic small letter kje	FC
selection sign	FD
cyrillic small letter short U	FE
cyrillic small letter dzhe	FF

ISO8859–6

Table 4. ISO8859–6

Symbolic Name	Hex Value
no-break space	A0
currency sign	A4
Arabic comma	AC
soft hyphen	AD
Arabic semicolon	BB
Arabic question mark	BF
Arabic letter hamza	C1
Arabic letter alef with madda above	C2
Arabic letter alef with hamza above	C3
Arabic letter waw with hamza above	C4
Arabic letter alef with hamza below	C5
Arabic letter yeh with hamza above	C6
Arabic letter alef	C7
Arabic letter beh	C8
Arabic letter teh marbuta	C9
Arabic letter teh	CA
Arabic letter theh	CB
Arabic letter jeem	CC
Arabic letter hah	CD
Arabic letter khah	CE
Arabic letter dal	CF
Arabic letter thal	D0
Arabic letter reh	D1
Arabic letter zain	D2
Arabic letter seen	D3
Arabic letter sheen	D4
Arabic letter sad	D5
Arabic letter dad	D6
Arabic letter tah	D7
Arabic letter zah	D8
Arabic letter ain	D9
Arabic letter ghain	DA
Arabic letter tatweel	E0
Arabic letter feh	E1
Arabic letter qaf	E2
Arabic letter kaf	E3
Arabic letter lam	E4
Arabic letter meem	E5

Table 4. ISO8859–6 (continued)

Symbolic Name	Hex Value
Arabic letter noon	E6
Arabic letter heh	E7
Arabic letter waw	E8
Arabic letter alef maksura	E9
Arabic letter yeh	EA
Arabic letter fathatan	EB
Arabic letter dammatan	EC
Arabic letter kasratan	ED
Arabic letter fatha	EE
Arabic letter damma	EF
Arabic letter kasra	F0
Arabic letter shadda	F1
Arabic letter sukun	F2

ISO8859–7

Table 5. ISO8859–7 Code set

Symbolic Name	Hex Value
no break space	A0
left single quotation mark	A1
right single quotation mark	A2
puond sign	A3
euro sign	A4
broken bar	A6
section sign	A7
diaeresis	A8
copyright sign	A9
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
horizontal bar	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
greek tonos	B4
greek dialytika tonos	B5
greek capital letter alpha with tonos	B6
middle dot	B7
greek capital letter epsilon with tonos	B8

Table 5. ISO8859–7 Code set (continued)

Symbolic Name	Hex Value
greek capital letter eta with tonos	B9
greek capital letter iota with tonos	BA
right-pointing double angle quotation mark	BB
greek capital letter omicron with tonos	BC
vulgar fraction one half	BD
greek capital letter upsilon with tonos	BE
greek capital letter omega with tonos	BF
greek small letter iota with dialytika and tonos	C0
greek capital letter alpha	C1
greek capital letter beta	C2
greek capital letter gamma	C3
greek capital letter delta	C4
greek capital letter epsilon	C5
greek capital letter zeta	C6
greek capital letter eta	C7
greek capital letter theta	C8
greek capital letter iota	C9
greek capital letter kappa	CA
greek capital letter lambda	CB
greek capital letter mu	CC
greek capital letter nu	CD
greek capital letter xi	CE
greek capital letter omicron	CF
greek capital letter pi	D0
greek capital letter rho	D1
greek capital letter sigma	D3
greek capital letter tau	D4
greek capital letter upsilon	D5
greek capital letter phi	D6
greek capital letter chi	D7
greek capital letter psi	D8
greek capital letter omega	D9
greek capital letter iota with dialytika	DA
greek capital letter upsilon with dialytika	DB
greek small letter alpha with tonos	DC
greek small letter epsilon with tonos	DD
greek small letter eta with tonos	DE
greek small letter iota with tonos	DF
greek small letter upsilon with dialytika and tonos	E0
greek small letter alpha	E1

Table 5. ISO8859–7 Code set (continued)

Symbolic Name	Hex Value
greek small letter beta	E2
greek small letter gamma	E3
greek small letter delta	E4
greek small letter epsilon	E5
greek small letter zeta	E6
greek small letter eta	E7
greek small letter theta	E8
greek small letter iota	E9
greek small letter kappa	EA
greek small letter lambda	EB
greek small letter mu	EC
greek small letter nu	ED
greek small letter xi	EE
greek small letter omicron	EF
greek small letter pi	F0
greek small letter rho	F1
greek small letter final sigma	F2
greek small letter sigma	F3
greek small letter tau	F4
greek small letter upsilon	F5
greek small letter phi	F6
greek small letter chi	F7
greek small letter psi	F8
greek small letter omega	F9
greek small letter iota with dialytika	FA
greek small letter upsilon with dialytika	FB
greek small letter omicron with tonos	FC
greek small letter upsilon with tonos	FD
greek small letter omega with tonos	FE

ISO8859–8

Table 6. ISO8859–8 Code set

Symbolic Name	Hex Value
no-break space	A0
cent sign	A2
pound sign	A3
currency sign	A4
yen sign	A5
broken bar	A6

Table 6. ISO8859–8 Code set (continued)

Symbolic Name	**Hex Value**
section sign	A7
diaeresis	A8
copyright sign	A9
multiplication sign	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
overline	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
acute accent	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
cedilla	B8
superscript one	B9
division sign	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vilgar fraction three quarters	BE
double low line	DF
hebrew letter alef	EO
hebrew letter bet	E1
hebrew letter gimel	E2
hebrew letter dalet	E3
hebrew letter he	E4
hebrew letter vav	E5
hebrew letter zayin	E6
hebrew letter het	E7
hebrew letter tet	E8
hebrew letter yod	E9
hebrew letter final kaf	EA
hebrew letter kaf	EB
hebrew letter lamed	EC
hebrew letter final mem	ED
hebrew letter mem	EE

Table 6. ISO8859–8 Code set (continued)

Symbolic Name	Hex Value
hebrew letter final nun	EF
hebrew letter nun	F0
hebrew letter samekh	F1
hebrew letter ayin	F2
hebrew letter final pe	F3
hebrew letter pe	F4
hebrew letter final tsadi	F5
hebrew letter tsadi	F6
hebrew letter qof	F7
hebrew letter resh	F8
hebrew letter shin	F9
hebrew letter tav	FA

ISO8859–9

Table 7. ISO8859–9 Code set

Symbolic Name	Hex Value
no-break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
currency sign	A4
yen sign	A5
broken bar	A6
section sign	A78
diaeresis	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double quotation mark	AB
not sign	AC
sofy hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
acute accent	B4
micro sign	B5
pilcrow sign	B6

Table 7. ISO8859–9 Code set (continued)

Symbolic Name	Hex Value
middle dot	B7
cedilla	B8
superscript one	B9
masculine ordinal indicator	BA
right pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter E with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter G with breve	D0
latin capital letter N with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circumflex	DB
latin capital letter U with diaeresis	DC
latin capital letter I with dot above	DD
latin capital letter S with cedilla	DE

Table 7. ISO8859–9 Code set (continued)

Symbolic Name	Hex Value
latin small letter sharp S	DF
latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diseresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter G with breve	F0
latin small letter N with tilde	F1
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter dotless I	FD
latin small letter S with cedilla	FE
latin small letter Y with diaeresis	FF

ISO8859–15

Table 8. ISO8859–1 Code set

Symbolic Name	Hex Value
no-break space	A0
inverted exclamation mark	A1

Table 8. ISO8859–1 Code set (continued)

Symbolic Name	Hex Value
cent sign	A2
pound sign	A3
euro sign	A4
yen sign	A5
latin capital letter S with caron	A6
section sign	A7
letin small letter S with caron	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
latin capital letter Z with caron	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
latin small letter Z with caron	B8
superscript one	B9
masculine ordinal indicator	BA
right-pointing bouble angle quotation marks	BB
latin capital ligature oe	BC
latin small ligature oe	BD
latin capital letter Y with diaeresis	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9

Table 8. ISO8859–1 Code set (continued)

Symbolic Name	Hex Value
latin capital letter W with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter eth	D0
latin capital letter N with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circumflex	DB
latin capital letter U with diaeresis	DC
latin capital letter Y with acute	DD
latin capital letter thorn	DE
latin small letter sharp S	DF
latin small letter A with grave	EO
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter eth	F0
latin small letter N with tilde	F1

Table 8. ISO8859–1 Code set (continued)

Symbolic Name	Hex Value
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter thorn	FE
latin small letter Y with diaeresis	FF

IBM Code Sets

The following IBM PC code sets are described:

- "IBM-856"
- "IBM-921" on page 200
- "IBM-922" on page 202
- "IBM-1046" on page 205
- "IBM-1124" on page 208
- "IBM-1129" on page 210
- "TIS-620" on page 213

IBM-856

Table 9. IBM–856 Code set

Symbolic Name	Hex Value
hebrew letter alef	80
hebrew letter bet	81
hebrew letter gimel	82
hebrew letter dalet	83
hebrew letter he	84
hebrew letter vav	85
hebrew letter zayin	86
hebrew letter het	87
hebrew letter tet	88
hebrew letter yod	89
hebrew letter final kaf	8A
hebrew letter kaf	8B

Table 9. IBM–856 Code set (continued)

Symbolic Name	Hex Value
hebrew letter lamed	8C
hebrew letter final mem	8D
hebrew letter mem	8E
hebrew letter final nun	8F
hebrew letter nun	90
hebrew letter samekh	91
hebrew letter ayin	92
hebrew letter final pe	93
hebrew letter pe	94
hebrew letter final tsadi	95
hebrew letter tsadi	96
hebrew letter qof	97
hebrew letter resh	98
hebrew letter shin	99
hebrew letter tav	9A
pound sign	9C
multiplication sign	9E
registered sign	A9
not sign	AA
vulgar fraction one half	AB
vulgar fraction one quarter	AC
left pointing double angle quotation mark	AE
right pointing double angle quotation mark	AF
light shade	B0
medium shade	B1
dark shade	B2
box drawings light vertical	B3
box drawings light vertical and left	B4
copyright sign	B8
box drawings double vertival and left	B9
box drawings double vertical	BA
box drawings double down and left	BB
box drawings double up and left	BC
cent sign	BD
yen sign	BE
box drawings light down and left	BF
box drawings light up and right	C0
box drawings light up and horizontal	C1
box drawings light down and horizontal	C2
box drawings light vertical and right	C3

Table 9. IBM–856 Code set (continued)

Symbolic Name	Hex Value
box drawings light horizontal	C4
box drawings light vertical and horizontal	C5
box drawings double up and right	C8
box drawings double down and right	C9
box drawings double up and horizontal	CA
box drawings double down and horizontal	CB
box drawings double vertical and right	CC
box drawings double horizontal	CD
box drawings double vertical and horizontal	CE
currency sign	CF
box drawings light up and left	D9
box drawings light down and right	DA
full block	DB
lower half block	DC
broken bar	DD
upper half block	DF
micro sign	E6
overline	EE
acute accent	EF
soft hyphen	F0
plus-minus sign	F1
double low line	F2
vulgar fraction three quarters	F3
pilcrow sign	F4
section sign	F5
division sign	F6
cedilla	F7
degree sign	F8
diaeresis	F9
middle dot	FA
superscript one	FB
superscript three	FC
superscript two	FD
black square	FE
no-break space	FF

IBM-921

Table 10. IBM–921 Code set

Symbolic Name	Hex Value
no-break space	A0
right double quotation mark	A1
cent sign	A2
pound sign	A3
euro sign	A4
double low-9 quotation mark	A5
broken bar	A6
section sign	A7
latin capital letter O with stroke	A8
copyright sign	A9
latin capital letter R with cedilla	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
latin capital letter AE	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
left double quotation mark	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
latin small letter O with stroke	B8
superscript one	B9
latin small letter R with cedilla	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
latin small letter AE	BF
latin capital letter A with ogonek	C0
latin capital letter I with ogonek	C1
latin capital letter A with macron	C2
latin capital letter C with acute	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5

Table 10. IBM–921 Code set (continued)

Symbolic Name	Hex Value
latin capital letter E with ogonek	C6
latin capital letter E with macron	C7
latin capital letter C with caron	C8
latin capital letter E with acute	C9
latin capital letter Z with acute	CA
latin capital letter E with dot above	CB
latin capital letter G with cedilla	CC
latin capital letter K with cedilla	CD
latin capital letter I with macron	CE
latin capital letter L with cedilla	CF
latin capital letter S with caron	D0
latin capital letter N with acute	D1
latin capital letter N with cedilla	D2
latin capital letter O with acute	D3
latin capital letter O with macron	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter U with ogonek	D8
latin capital letter L with stroke	D9
latin capital letter S with acute	DA
latin capital letter U with macron	DB
latin capital letter U with diaeresis	DC
latin capital letter Z with dot above	DD
latin capital letter Z with caron	DE
latin small letter sharp S	DF
latin small letter A with ogonek	E0
latin small letter I with ogonek	E1
latin small letter A with macron	E2
latin small letter C with acute	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter E with ogonek	E6
latin small letter E with macron	E7
latin small letter C with caron	E8
latin small letter E with acute	E9
latin small letter Z with acute	EA
latin small letter E with dot above	EB
latin small letter G with cedilla	EC
latin small letter K with cedilla	ED

Table 10. IBM–921 Code set (continued)

Symbolic Name	Hex Value
latin small letter I with macron	EE
latin small letter L with cedilla	EF
latin small letter S with caron	F0
latin small letter N with acute	F1
latin small letter N with cedilla	F2
latin small letter O with acute	F3
latin small letter O with macron	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter U with ogonek	F8
latin small letter L with stroke	F9
latin small letter S with acute	FA
latin small letter U with macron	FB
latin small letter U with diaeresis	FC
latin small letter Z with dot above	FD
latin small letter Z with caron	FE
right single quotation mark	FF

IBM-922

Table 11. IBM–922 Code set

Symbolic Name	Hex Value
no break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
euro sign	A4
yenb sign	A5
broken bar	A6
section sign	A7
diaeresis	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0

Table 11. IBM–922 Code set (continued)

Symbolic Name	Hex Value
plus-minus sign	B1
superscript two	B2
superscript three	B3
acute accent	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
cedilla	B8
superscript one	B9
masculine ordinal indicator	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter E with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter S with caron	D0
latin capital letter N with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8

Table 11. IBM–922 Code set (continued)

Symbolic Name	Hex Value
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circuflex	DB
latin capital letter U with diaeresis	DC
latin capital letter Y with acute	DD
latin capital letter Z with caron	DE
latin small letter sharp S	DF
latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED
latin small letter I with curcumflex	EE
latin small letter I with diaeresis	EF
latin small letter S with caron	F0
latin small letter N with tilde	F1
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter Z with caron	FE
latin small letter Y with diaeresis	FF

IBM-1046

Table 12. IBM–1046 Code set

Symbolic Name	Hex Value
arabic letter alef with hamza below final form	80
multiplication sign	81
division sign	82
arabic letter seen first part of final form	83
arabic letter sheen first part of final form	84
arabic letter sad first part of final form	85
arabic letter dadfirst part of final form	86
arabic tatweel with fathatan above	87
full block	89
box drawings light vertical	8A
box drawings light horizontal	8B
box drawings light down and left	8C
box drawings light down and right	8D
box drawings light up and right	8E
box drawings light up and left	8F
arabic damma medial form	90
arabic kasra medial form	91
arabic shadda medial form	92
arabic sukun medial form	93
arabic fatha medial form	94
arabic letter yeh with hamza above final form	95
arabic letter alef maksura final form	96
arabic letter yeh initial form	97
arabic letter yeh final form	98
arabic letter ghain final form	99
arabic letter ghain initial form	9A
arabic letter ghain medial form	9B
arabic ligature lam with alef with madda above final form	9C
arabic ligature lam with alef with hamza above final form	9D
arabic ligature lam with alef with hamza below final form	9E
arabic ligature lam with alef final form	9f
no-break space	A0
arabic letter alef with madda above after lam	A1
arabic letter alef with hamza above after lam	A2
arabic letter alef with hamza below after lam	A3
currency sign	A4
arabic letter alef after lam	A5
arabic letter yeh with hamza above initial form	A6

Table 12. IBM–1046 Code set (continued)

Symbolic Name	Hex Value
arabic letter beh with initial form	A7
arabic letter teh with initial form	A8
arabic letter theh with initial form	A9
arabic letter jeem with initial form	AA
arabic letter hah with initial form	AB
arabic comma	AC
soft hyphen	AD
arabic letter khan initial form	AE
arabic letter seen initial form	AF
arabic-indic digit zero	B0
arabic-indic digit one	B1
arabic-indic digit two	B2
arabic-indic digit three	B3
arabic-indic digit four	B4
arabic-indic digit five	B5
arabic-indic digit six	B6
arabic-indic digit seven	B7
arabic-indic digit eight	B8
arabic-indic digit nine	B9
arabic letter sheen initial form	BA
arabic semicolon	BB
arabic letter sad initial form	BC
arabic letter dad initial form	BD
arabic letter ain initial form	BE
arabic question mark	BF
arabic letter ain initial form	C0
arabic letter hamza	C1
arabic letter alef with madda above	C2
arabic letter alef with hamza above	C3
arabic letter waw with hamza above	C4
arabic letter alef with hamza below	C5
arabic letter yeh with hamza above	C6
arabic letter alef	C7
arabic letter beh	C8
arabic letter teh marbuta	C9
arabic letter teh	CA
arabic letter theh	CB
arabic letter jeem	CC
arabic letter hah	CD
arabic letter khah	CE

Table 12. IBM–1046 Code set (continued)

Symbolic Name	Hex Value
arabic letter dal	CF
arabic letter thal	D0
arabic letter reh	D1
arabic letter zain	D2
arabic letter seen	D3
arabic letter sheen	D4
arabic letter sad	D5
arabic letter dad	D6
arabic letter tah	D7
arabic letter zah	D8
arabic letter ain	D9
arabic letter ghain	DA
arabic letter ain medial form	DB
arabic letter alef with madda above final form	DC
arabic letter alef with hamza above final form	DD
arabic letter alef with final form	DE
arabic letter feh initial form	DF
arabic tatweel	E0
arabic letter feh	E1
arabic letter qaf	E2
arabic letter kaf	E3
arabic letter lam	E4
arabic letter meem	E5
arabic letter noon	E6
arabic letter heh	E7
arabic letter waw	E8
arabic letter alef maksura	E9
arabic letter yeh	EA
arabic fathatan	EB
arabic dammatan	EC
arabic kasratan	ED
arabic fatha	EE
arabic damma	EF
arabic kasra	F0
arabic shadda	F1
arabic sukun	F2
arabic letter qar initial form	F3
arabic letter kaf initial form	F4
arabic letter lam initial form	F5
arabic kasseh	F6

Table 12. IBM–1046 Code set (continued)

Symbolic Name	Hex Value
arabic ligature lam with alef with madda above isolated form	F7
arabic ligature lam with alef with hamza above isolated form	F8
arabic ligature lam with alef with madda below isolated form	F9
arabic ligature lam with alef isolated form	FA
arabic letter meem initial form	FB
arabic letter noon initial form	FC
arabic letter heh initial form	FD
arabic letter heh final form	FE
euro sign	FF

IBM-1124

Table 13. IBM–1124 Code set

Symbolic Name	Hex Value
no-break space	A0
cyrillic capital letter io	A1
cyrillic capital letter dje	A2
cyrillic capital letter ghe with upturn	A3
cyrillic capital letter ukrainian ie	A4
cyrillic capital letter dze	A5
cyrillic capital letter byelorussian-ukranian i	A6
cyrillic capital letter yi	A7
cyrillic capital letter je	A8
cyrillic capital letter lje	A9
cyrillic capital letter nje	AA
cyrillic capital letter tshe	AB
cyrillic capital letter kje	AC
soft hyphen	AD
cyrillic capital letter short U	AE
cyrillic capital letter dzhe	AF
cyrillic capital letter A	B0
cyrillic capital letter be	B1
cyrillic capital letter ve	B2
cyrillic capital letter ghe	B3
cyrillic capital letter de	B4
cyrillic capital letter ie	B5
cyrillic capital letter zhe	B6
cyrillic capital letter ze	B7
cyrillic capital letter I	B8
cyrillic capital letter short I	B9

Table 13. IBM–1124 Code set (continued)

Symbolic Name	Hex Value
cyrillic capital letter ka	BA
cyrillic capital letter el	BB
cyrillic capital letter em	BC
cyrillic capital letter en	BD
cyrillic capital letter O	BE
cyrillic capital letter pe	BF
cyrillic capital letter er	C0
cyrillic capital letter es	C1
cyrillic capital letter te	C2
cyrillic capital letter U	C3
cyrillic capital letter ef	C4
cyrillic capital letter ha	C5
cyrillic capital letter tse	C6
cyrillic capital letter che	C7
cyrillic capital letter sha	C8
cyrillic capital letter shcha	C9
cyrillic capital letter hard sign	CA
cyrillic capital letter yeru	CB
cyrillic capital letter soft sign	CC
cyrillic capital letter E	CD
cyrillic capital letter yu	CE
cyrillic capital letter ya	CF
cyrillic small letter A	D0
cyrillic small letter be	D1
cyrillic small letter ve	D2
cyrillic small letter ghe	D3
cyrillic small letter de	D4
cyrillic small letter ie	D5
cyrillic small letter zhe	D6
cyrillic small letter ze	D7
cyrillic small letter I	D8
cyrillic small letter short I	D9
cyrillic small letter ka	DA
cyrillic small letter el	DB
cyrillic small letter em	DC
cyrillic small letter en	DD
cyrillic small letter O	DE
cyrillic small letter pe	DF
cyrillic small letter er	E0
cyrillic small letter es	E1

Table 13. IBM–1124 Code set (continued)

Symbolic Name	Hex Value
cyrillic small letter te	E2
cyrillic small letter u	E3
cyrillic small letter ef	E4
cyrillic small letter ha	E5
cyrillic small letter tse	E6
cyrillic small letter che	E7
cyrillic small letter sha	E8
cyrillic small letter shcha	E9
cyrillic small letter hard sign	EA
cyrillic small letter yeru	EB
cyrillic small letter soft sign	EC
cyrillic small letter E	ED
cyrillic small letter yu	EE
cyrillic small letter ya	EF
numero sign	F0
cyrillic small letter io	F1
cyrillic small letter dje	F2
cyrillic small letter ghe with upturn	F3
cyrillic small letter ukrainian ie	F4
cyrillic small letter dze	F5
cyrillic small letter byelorussian-ukrainian	F6
cyrillic small letter yi	F7
cyrillic small letter je	F8
cyrillic small letter lje	F9
cyrillic small letter nje	FA
cyrillic small letter tshe	FB
cyrillic small letter kje	FC
section sign	FD
cyrillic small letter short u	FE
cyrillic small letter dzhe	FF

IBM-1129

Table 14. IBM–1129 Code set

Symbolic Name	Hex Value
no-break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
euro sign	A4

Table 14. IBM–1129 Code set (continued)

Symbolic Name	Hex Value
yen sign	A5
broken bar	A6
section sign	A7
latin small ligature OE	A8
copyright sign	A9
feminine ordinal indicator	AA
left pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
latin capital Y with diaeresis	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
latin capital ligature OE	B8
superscript one	B9
masculine ordinal indicator	BA
right pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with breve	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter E with circumflex	CA
latin capital letter E with diaeresis	CB
combining grave accent	CC

Table 14. IBM–1129 Code set (continued)

Symbolic Name	**Hex Value**
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter D with stroke	D0
latin capital letter N with tilde	D1
combining hook above	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with horn	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circuflex	DB
latin capital letter U with diaeresis	DC
latin capital letter U with horn	DD
combining tilde	DE
latin small letter sharp S	DF
latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with breve	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
combining acute accent	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter D with stroke	F0
latin small letter N with tilde	F1
combining dot below	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4

Table 14. IBM–1129 Code set (continued)

Symbolic Name	Hex Value
latin small letter O with horn	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter U with horn	FD
dong sign	FE
latin small letter Y with diaeresis	FF

TIS-620

Table 15. TIS–620 Code set

Symbolic Name	Hex Value
thai character ko kai	A1
thai character kho khai	A2
thai character kho khuat	A3
thai character kho khwai	A4
thai character kho khon	A5
thai character kho rakhang	A6
thai character ngo ngu	A7
thai character cho chan	A8
thai character cho ching	A9
thai character cho chang	AA
thai character so so	AB
thai character cho choe	AC
thai character yo ying	AD
thai character do chada	AE
thai character to patak	AF
thai character tho than	B0
thai character tho nangmontho	B1
thai character tho phuthao	B2
thai character no nen	B3
thai character do dek	B4
thai character to tao	B5
thai character tho thung	B6
thai character tho thahan	B7
thai character tho thong	B8

Table 15. TIS–620 Code set (continued)

Symbolic Name	Hex Value
thai character no nu	B9
thai character bo baimai	BA
thai character po pla	BB
thai character pho phung	BC
thai character fo fa	BD
thai character pho phan	BE
thai character fo fan	BF
thai character pho samphao	C0
thai character mo ma	C1
thai character yo yak	C2
thai character ro rua	C3
thai character ru	C4
thai character lo ling	C5
thai character lu	C6
thai character wo waen	C7
thai character so sala	C8
thai character so rusi	C9
thai character so sua	CA
thai character ho hip	CB
thai character lo chula	CC
thai character o ang	CD
thai character ho nokhuk	CE
thai character paiyannoi	CF
thai character sara a	D0
thai character mai han-akat	D1
thai character sara aa	D2
thai character sara am	D3
thai character sara i	D4
thai character sara ii	D5
thai character sara ue	D6
thai character sara uee	D7
thai character sara u	D8
thai character uu	D9
thai character phinthu	DA
thai currency symbol baht	DF
thai character sara e	E0
thai character sara ae	E1
thai character sara O	E2
thai character sara ai maimuan	E3
thai character sara ai maimalai	E4

Table 15. TIS–620 Code set (continued)

Symbolic Name	Hex Value
thai character lakkhangyao	E5
thai character maiyamok	E6
thai character maitaikhu	E7
thai character mai ek	E8
thai character mai tho	E9
thai character mai tri	EA
thai character mai chattawa	EB
thai character thanthakhat	EC
thai character nikhahit	ED
thai character yamakkan	EE
thai character fongman	EF
thai digit zero	F0
thai digit one	F1
thai digit two	F2
thai digit three	F3
thai digit four	F4
thai digit five	F5
thai digit six	F6
thai digit seven	F7
thai digit eight	F8
thai digit nine	F9
that character angkhankhu	FA
thai character khomut	FB

Appendix C. NLS Sample Program

This appendix contains a sample program fragment, `foo.c`, which illustrates internationalization through code set independent programming.

Message Source File for foo

A sample message source file for the `foo` utility is given here. Note we defined only one set and three messages in this catalog for illustration purposes only. A typical catalog contains several such messages.

The following is the message source file for `foo`, **foo.msg**.

```
$quote "
$set MS_FOO
CANTOPEN        "foo: cannot open %s\n"
BYTECNT         "number of bytes: %d\n"
CHARCNT         "number of characters: %d
```

Creation of Message Header File for foo

To generate the run-time catalog, use the **runcat** command as follows:

```
runcat foo foo.msg
```

This generates the **foo_msg.h** header file, as shown in the following section. Note that the set mnemonic is `MS_FOO` and the message mnemonics are `CANTOPEN`, `BYTECNT`, and `CHARCNT`. These mnemonics are used in the programs in this appendix.

```
/*
** The header file: foo_msg.h is as follows:
*/

#ifndef _H_FOO_MSG
#define _H_FOO_MSG
#include <limits.h>
#include <nl_types.h>
#define MF_FOO "foo.cat"

/* The following was generated from wc.msg. */


/* definitions for set MS_FOO */
#define MS_FOO 1

#define CANTOPEN 1
#define BYTECNT  2
#define CHARCNT  3

#endif
```

Single Source, Single Path Code-set Independent Version

The term *single source single path* refers to one path in a single application to be used to process both single-byte and multibyte code sets. The single source single path method eliminates all **ifdef**s for internationalization. All characters are handled the same way, whether they are members of single-byte or multibyte code sets.

Single source single path is desirable, but it can degrade performance. Thus, it is not recommended for all programs. There may be some programs that do not suffer any performance degradation when they are fully internationalized; in those cases, use the single source single path method.

The following fully internationalized version of the foo utility supports all code sets through single source single path, code-set independent programming:

```
/*
 * COMPONENT_NAME:
 *
 * FUNCTIONS: foo
 *
 * The following code shows how to count the number of bytes and
 * the number of characters in a text file.
 *
 * This example is for illustration purposes only. Performance
 * improvements may still be possible.
 *
 */

#include        <stdio.h>
#include        <ctype.h>
#include        <locale.h>
#include        <stdlib.h>
#include        "foo_msg.h"

#define MSGSTR(Num,Str) catgets(catd,MS_FOO,Num,Str)

/*
 * NAME: foo
 *
 * FUNCTION: Counts the number of characters in a file.
 *
 */

main(argc,argv)
int argc;
char **argv;
{
    int     bytesread,   /* number of bytes read */
        bytesprocessed;
    int     leftover;

   int     i;
    int     mbcnt;           /* number of bytes in a character */
    int     f;               /* File descriptor */
    int mb_cur_max;
    int    bytect;           /* name changed from charct... */
    int    charct;           /* for real character count */
    char   *curp, *cure;     /* current and end pointers into
                               ** buffer */
    char         buf[BUFSIZ+1];

    nl_catd      catd;

    wchar_t    wc;

    /* Obtain the current locale */
    (void) setlocale(LC_ALL,"");

    /* after setting the locale, open the message catalog */
    catd = catopen(MF_FOO,NL_CAT_LOCALE);

    /* Parse the arguments if any */

    /*
    ** Obtain  the maximum number of bytes in a character in the
    ** current locale.
    */
    mb_cur_max = MB_CUR_MAX;
    i = 1;
```

```
 /* Open the specified file and issue error messages if any */
 f = open(argv[i],0);
 if(f<0){
     fprintf(stderr,MSGSTR(CANTOPEN,              /*MSG*/
         "foo: cannot open %s\n"), argv[i]);      /*MSG*/
         exit(2);
 }

/* Initialize the variables for the count */
 bytect = 0;
 charct = 0;

 /* Start count of bytes and characters  */

 leftover = 0;

 for(;;) {
     bytesread = read(f,buf+leftover, BUFSIZ-leftover);
     /* issue any error messages here, if needed */
     if(bytesread <= 0)
          break;

     buf[leftover+bytesread] = '\0';
             /* Protect partial reads */
     bytect += bytesread;
     curp=buf;
     cure = buf + bytesread+leftover;
     leftover=0;      /* No more leftover */

     for(; curp<cure ;){
         /* Convert to wide character */
         mbcnt= mbtowc(&wc, curp, mb_cur_max);
         if(mbcnt <= 0){
             mbcnt = 1;
         }else if (cure - curp >=mb_cur_max){
             wc = *curp;
             mbcnt =1;
         }else{
             /* Needs more data */
             leftover= cure - curp;
             strcpy(buf, curp, leftover);
             break;
         }
         curp +=mbcnt;
         charct++;
     }
 }

     /* print number of chars and bytes */
 fprintf(stderr,MSGSTR(BYTECNT, "number of bytes:%d\n"),
         bytect);
 fprintf(stderr,MSGSTR(CHARCNT, "number of characters:%d\n"),
         charct);
 close(f);
 exit(0);
```

Single Source, Dual-Path Version Optimized for Single-Byte Code Sets

The term *single source dual path* refers to two paths in a single application where one of the paths is chosen at run time depending on the current locale setting, which indicates whether the code set in use is single-byte or multibyte.

If a program can retain its performance and not increase its executable file size too much, the single source dual path method is the preferred choice. You should evaluate the increase in the executable file size on a per command or utility basis.

In the single byte dual-path method, the **MB_CUR_MAX** macro specifies the maximum number of bytes in a multibyte character in the current locale. This should be used to determine at run time whether the processing path to be chosen is the single-byte or the multibyte path. Use a boolean flag to indicate the path to be chosen, for example:

```
int mbcodeset ;
/* After setlocale(LC_ALL,"") is done, determine the path to
** be chosen.
*/
if(MB_CUR_MAX == 1)
        mbcodeset = 0;
else    mbcodeset = 1;
```

This way, the current code set is checked to see if it is a multibyte code set and if so, the flag `mbcodeset` is set appropriately. Testing this flag has less performance impact than testing the **MB_CUR_MAX** macro several times.

```
if(mbcodeset){
        /* Multibyte code sets (also supports single-byte
        ** code sets )
        */
        /* Use multibyte or wide character processing
        functions */
}else{
        /* single-byte code sets */
        /* Process accordingly */
}
```

The preceeding approach is appropriate if internationalization affects a small proportion of a module. Excessive tests for providing dual paths may degrade performance. Provide the test at a level that precludes frequent testing for this case.

The following version of the `foo` utility produces one object, yet at run time, the appropriate path is chosen based on the code set to optimize performance for that code set. Note that we distinguish between single-byte and multibyte code sets only.

```
/*
 * COMPONENT_NAME:
 *
 * FUNCTIONS: foo
 *
 * The following code shows how to count the number of bytes and
 * the number of characters in a text file.
 *
 * This example is for illustration purposes only. Performance
 * improvements may still be possible.
 *
 */

#include        <stdio.h>
#include        <ctype.h>
#include        <locale.h>
#include        <stdlib.h>
#include        "foo_msg.h"

#define MSGSTR(Num,Str) catgets(catd,MS_FOO,Num,Str)

/*
 * NAME: foo
 *
 * FUNCTION: Counts the number of characters in a file.
```

```
 *
 */

main(argc,argv)
int argc;
char **argv;
{
    int bytesread,  /* number of bytes read */
        bytesprocessed;
    int   leftover;

    int   i;
    int   mbcnt;   /* number of bytes in a character */
    int   f;       /* File descriptor */
    int   mb_cur_max;
    int    bytect;             /* name changed from charct... */
    int    charct;             /* for real character count */
    char   *curp, *cure; /* current and end pointers into buffer */
    char        buf[BUFSIZ+1];

        nl_catd          catd;

        wchar_t    wc;

        /* flag to indicate if current code set is a
        ** multibyte code set
        */
        int     multibytecodeset;

        /* Obtain the current locale */
        (void) setlocale(LC_ALL,"");

    /* after setting the locale, open the message catalog */
    catd = catopen(MF_FOO,NL_CAT_LOCALE);

   /* Parse the arguments if any */

    /*
    ** Obtain the maximum number of bytes in a character in the
    ** current locale.
    */
    mb_cur_max = MB_CUR_MAX;

    if(mb_cur_max >1)
        multibytecodeset = 1;
    else
        multibytecodeset = 0;

    i = 1;

    /* Open the specified file and issue error messages if any */
    f = open(argv[i],0);
    if(f<0){
        fprintf(stderr,MSGSTR(CANTOPEN,              /*MSG*/
            "foo: cannot open %s\n"), argv[i]);      /*MSG*/
            exit(2);
    }

    /* Initialize the variables for the count */
    bytect = 0;
    charct = 0;

    /* Start count of bytes and characters  */

    leftover = 0;

    if(multibytecodeset){
```

```
    /* Full internationalzation */
    /* Handles supported multibyte code sets */
    for(;;) {
        bytesread = read(f,buf+leftover,
                BUFSIZ-leftover);
        /* issue any error messages here, if needed */
        if(bytesread <= 0)
            break;

        buf[leftover+bytesread] = '\0';
                /* Protect partial reads */
        bytect += bytesread;
        curp=buf;

        cure = buf + bytesread+leftover;
        leftover=0; /* No more leftover */

        for(; curp<cure ;){
            /* Convert to wide character */
            mbcnt= mbtowc(&wc, curp, mb_cur_max);
            if(mbcnt <= 0){
                mbcnt = 1;
            }else if (cure - curp >=mb_cur_max){
                wc = *curp;
                mbcnt =1;

            }else{
                /* Needs more data */
                leftover= cure - curp;
                strcpy(buf, curp, leftover);
                break;
            }
            curp +=mbcnt;
            charct++;
        }
    }
}else {

    /* Code specific to single-byte code sets that
    ** avoids conversion to widechars and thus optimizes
    ** performance for single-byte code sets.
    */

    for(;;) {
        bytesread = read(f,buf, BUFSIZ);
        /* issue any error messages here, if needed */
        if(bytesread <= 0)
                break;

            bytect += bytesread;
            charct += bytesread;
    }

}

    /* print number of chars and bytes */
fprintf(stderr,MSGSTR(BYTECNT, "number of bytes:%d\n"),
        bytect);
fprintf(stderr,MSGSTR(CHARCNT, "number of characters:%d\n"),
        charct);
close(f);
exit(0);
```

Appendix D. Use of the libcur Package

Programs that use the libcur package (extension to AT&T's libcurses package) need to make the following changes:

1. Remove the assumption that the number of bytes need to represent a character in a code set also represents the display column width of the character. Use the **wcwidth** subroutine to determine the number of display columns needed by the wide character code of a character.
2. **NLSCHAR** is redefined to be **wchar_t**.
3. The `win->_y [y][x]` has **wchar_t** encodings.
4. Programs should not assume any particular encodings on the **wchar_t**.
5. Programs should use the **addstr**, **waddstr**, **mvaddstr**, and **mvwaddstr** subroutines rather than the **addch** family of subroutines. All string arguments are in multibyte form.
6. The **addch** and **waddch** subroutines accept a **wchar_t** encoding of the character. Programs that use these subroutines should ensure that **wchar_t** are used in calling these functions. The (x,y) are incremented by the number of columns occupied by the **wchar_t** passed to these subroutines.
7. The **delch**, **wdelch**, **mvdelch**, and **mvwdelch** subroutines support delete and backspace on multibyte characters depending on the current position of (x,y). If the current (x,y) column position points to either the first or second column of a two-column character, the **delch** subroutine deletes both columns and shifts the rest of the line by the number of columns deleted.
8. The **insch**, **winsch**, **mvinsch**, and **mvwinsch** subroutines can be used to insert a **wchar_t** encoding of a character at the current (x,y) position. The line is shifted by the number of columns needed by the **wchar_t**.
9. The libcur package is modified to support box drawing characters as defined in the **terminfo** database and not assume the graphic characters in the IBM-850 code set. The libcur package supports drawing of primary and alternate box characters as defined in the **box_chars_1** and **box_chars_2** entries in the terminfo database. To use this, programs should be modified in the following fashion:

   ```
   Drawing Primary box characters:



           wcolorout(win, Bxa);
           cbox(win);
           wcolorend(win);

           or,
           wcolorout(win, Bxa);
           drawbox(win, y,x, height, width);
           wcolorend(win);

   Drawing Alternate box characters:



           wcolorout(win, Bya)
           cboxalt(win);
           wcolorend(win);

           or,

           wcolorout(win, Bya);
           drawbox(win, y, x, height, width);
           wcolorend(win);
   ```

   `Bxa` and `Bya` refer to the primary and alternate attributes defined in the **terminfo** database.

   The following macros are added in the **cur01.h** file:

```
cboxalt(win)

drawboxalt(win, y,x, height, width)
```

10. Programs that need to support input of multibyte characters should not set **_extended** to TRUE by a call to **extended(TRUE)**. When the **_extended** flag is true, the **wgetch** subroutine returns **wchar_t** encodings of the character. With multibyte characters, this encoding of **wchar_t** may conflict with predefined values for escape sequences or function keys. Avoid this conflict when using multibyte code sets by setting extended to off (**extended(FALSE)**) before input. (The default is TRUE.)

    Programs that do multibyte character input should do the following:

```
    Input routine:

    Example:

        int c, count;
        char buf[];

        extended(FALSE); /* obtain one byte at a time */
        count =0;
        while(1){
            c = wgetch();  /* get one byte at a time */
            buf[count++] = c;
            if(count <=MB_CUR_MAX)
                if(mblen(buf, count) != -1)
                    break; /* character found* /
            else
                /*Error. No character can be found */
                /* Handle this case appropriately */
                break;
        }
/* buf contains the input multibyte sequence */
/* Now handle PF keys, or any escape sequence here */
```

11. The **inch**, **winch**, **mvinch**, and **mvwinch** subroutines return the **wchar_t** at the current (x,y) position. Note that in the case of a double column width character, if the (x,y) point is at the first column, the **wchar_t** code of the double column width character is returned. If the (x,y) point is at the second column, WEOF is returned.

Appendix E. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIX
- AIX 5L
- IBM

Microsoft and Windows are trademarks of the Microsoft Corporation in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Index

Special characters

D

J

K

L

M

N

O

P

R

S

T

U

V

W

Readers' Comments — We'd Like to Hear from You

AIX 5L Version 5.2
National Language Support Guide and Reference

Publication No. SC23-4872-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SC23-4872-00

IBM

Cut or Fold Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department H6DS-905-6C006
11501 Burnet Road
Austin, TX 78758-3493

Fold and Tape | **Please do not staple** | Fold and Tape

SC23-4872-00

Cut or Fold Along Line

Printed in U.S.A.

SC23-4872-00