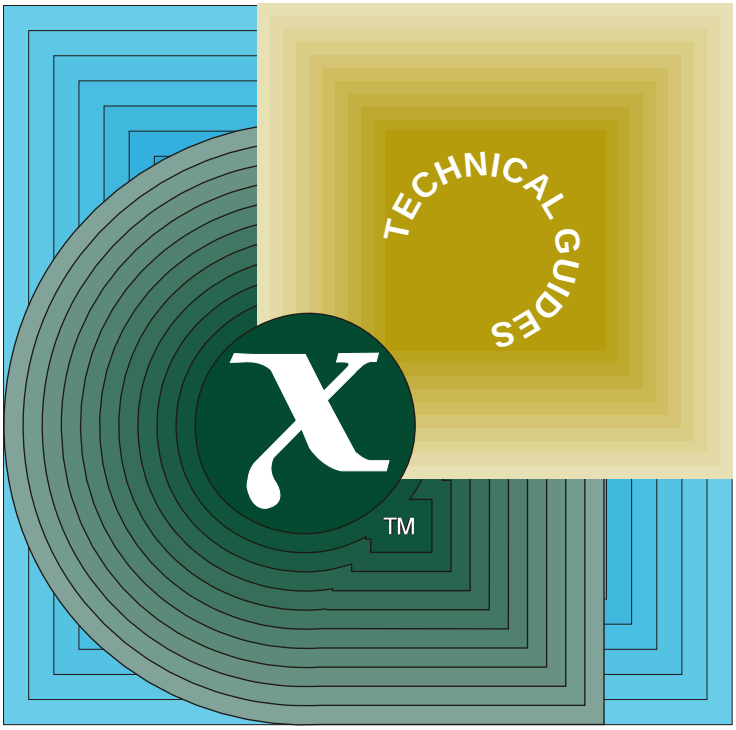
Guide

# Internationalization Guide
# Version 2




THE Open GROUP

[This page intentionally left blank]

*X/Open Guide*

# Internationalisation Guide

## Version 2

*X/Open Company Ltd.*

© *July 1993, X/Open Company Limited*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

X/Open Guide

Internationalisation Guide Version 2

ISBN: 1-85912-002-4

X/Open Document Number: G304

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

# Contents

# Preface

## X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and allows users to move between systems with a minimum of retraining.

The components of the Common Applications Environment are defined in X/Open CAE Specifications. These contain, among other things, an evolving portfolio of practical application programming interfaces (APIs), which significantly enhance portability of application programs at the source code level, and definitions of, and references to, protocols and protocol profiles, which significantly enhance the interoperability of applications.

The X/Open CAE Specifications are supported by an extensive set of conformance tests and a distinct X/Open trademark - the XPG brand - that is licensed by X/Open and may be carried only on products that comply with the X/Open CAE Specifications.

The XPG brand, when associated with a vendor's product, communicates clearly and unambiguously to a procurer that the software bearing the brand correctly implements the corresponding X/Open CAE Specifications. Users specifying XPG-conformance in their procurements are therefore certain that the branded products they buy conform to the CAE Specifications.

X/Open is primarily concerned with the selection and adoption of standards. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organisations to assist in the creation of formal standards covering the needed functions, and to make its own work freely available to such organisations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

## X/Open Specifications

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the long-life specifications that form the basis for conformant and branded X/Open systems. They are intended to be used widely within the industry for product development and procurement purposes.

Developers who base their products on a current CAE Specification can be sure that either the current specification or an upwards-compatible version of it will be referenced by a future XPG brand (if not referenced already), and that a variety of compatible, XPG-branded systems capable of hosting their products will be available, either immediately or in the near future.

CAE Specifications are not published to coincide with the launch of a particular XPG brand, but are published as soon as they are developed. By providing access to its specifications in this way, X/Open makes it possible for products that conform to the CAE (and hence are eligible for a future XPG brand) to be developed as soon as practicable, enhancing the value of the XPG brand as a procurement aid to users.

- *Preliminary Specifications*

  These are specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for the purpose of validation through practical implementation or prototyping. A Preliminary Specification is not a ''draft'' specification. Indeed, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE Specification.

  Preliminary Specifications are analogous with the ''trial-use'' standards issued by formal standards organisations, and product development teams are intended to develop products on the basis of them. However, because of the nature of the technology that a Preliminary Specification is addressing, it is untried in practice and may therefore change before being published as a CAE Specification. In such a case the CAE Specification will be made as upwards-compatible as possible with the corresponding Preliminary Specification, but complete upwards-compatibility in all cases is not guaranteed.

In addition, X/Open periodically publishes:

- *Snapshots*

  Snapshots are ''draft'' documents, which provide a mechanism for X/Open to disseminate information on its current direction and thinking to an interested audience, in advance of formal publication, with a view to soliciting feedback and comment.

  A Snapshot represents the interim results of an X/Open technical activity. Although at the time of publication X/Open intends to progress the activity towards publication of an X/Open Preliminary or CAE Specification, X/Open is a consensus organisation, and makes no commitment regarding publication.

  Similarly, a Snapshot does not represent any commitment by any X/Open member to make any specific products available.

### X/Open Guides

X/Open Guides provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant.

X/Open Guides are not normative, and should not be referenced for purposes of specifying or claiming X/Open-conformance.

### This Document

This document is a Guide (see above). It describes Internationalisation facilities provided by systems that conform to the X/Open Common Applications Environment. The Guide contains background material on how X/Open facilities have evolved through various issues of the X/Open Portability Guide (XPG). It also identifies which features are common with other standards (for example, the ISO POSIX-1 standard and the ISO C standard), and which are unique to X/Open. It is intended for use by system and application programmers developing internationalised software, and by end-users requiring to use the multi-language features of a system that conforms to XPG4. There is also a lot to be learned from the Guide about good programming style for international portability and maintainability.

This Guide assumes the reader has a working knowledge of:

- system interfaces and headers presented in the X/Open CAE Specification **System Interfaces and Headers, Issue 4**
- commands and utilities presented in the X/Open CAE Specification **Commands and Utilities, Issue 4**
- the C Programming Language presented in the ISO C standard.

Specific Internationalisation features of the X/Open System Interface (XSI) are re-introduced in the Guide and no prior knowledge is assumed in this area.

This issue of the Guide is aligned with the XPG4 set of X/Open CAE specifications that define the XSI Operating System requirements. When these specifications are referred to collectively, the term **XPG4** base specifications is used.

**Note:** This Guide is intended as a tutorial rather than a specification. Details of system interfaces are as accurate as possible but readers are referred to the **XPG4** base specifications for a definitive description of these interfaces.

This document is structured as follows:

- Chapter 1 provides background information and an introduction to internationalisation and localisation.
- Chapter 2 discusses how to develop internationalised software.
- Chapter 3 explains the features of the message system.
- Chapter 4 gives details of the interfaces available in XPG4 systems for internationalisation support.
- Chapter 5 is about using internationalised software.
- Chapter 6 describes the utilities available for defining and displaying information about locales.
- Appendix A discusses migration issues up to XPG3. XPG3 to XPG4 migration is covered by the referenced **Migration Guide**.
- Appendix B gives three *example* locales; they are provided to illustrate locale definition files and are not necessarily provided on systems that conform to XPG4.

A glossary and index are also provided.

### Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — utility names
  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*( ); names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. Brackets shown in this font, `[ ]`, are part of the syntax and do *not* indicate optional items. In syntax the `|` symbol is used to separate alternatives, and ellipses (`. . .`) are used to show that additional arguments are optional.
- **`Bold fixed width`** font is used to identify brackets that surround optional items in syntax, **`[ ]`**, and to identify system output in interactive examples.
- Variables within syntax statements are shown in *`italic fixed width font`*.
- Ranges of values are indicated with parentheses or brackets as follows:
  - — (a,b) means the range of all values from a to b, including neither a nor b
  - — [a,b] means the range of all values from a to b, including a and b
  - — [a,b) means the range of all values from a to b, including a, but not b
  - — (a,b] means the range of all values from a to b, including b, but not a.

# *Trade Marks*

AT&T® is a registered trade mark of American Telephone & Telegraph Company in the U.S.A. and other countries.

UNIX® is a registered trade mark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.

X/Open™ and the ‘‘X’’ device are trade marks of X/Open Company Limited in the U.K. and other countries.

# Referenced Documents

The following standards are referenced in this guide:

ANSI C
: ANS X3.159-1989, Programming Language C.

ISO / IEC 646: 1991
: Information processing — ISO 7-bit coded character set for information interchange.

ISO 6937: 1983
: Information processing — Coded character sets for text communication.

ISO 8859
: ISO 8859-*: 1987 Information processing — 8-bit single-byte coded graphic character sets — Parts 1 to 10 inclusive.

ISO C
: ISO / IEC 9899: 1990, Programming languages — C (which is technically identical to ANS X3.159-1989, Programming Language C).

ISO POSIX-1
: ISO / IEC 9945-1: 1990, Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (which is identical to IEEE Std 1003.1-1990).

ISO POSIX-2 DIS
: ISO / IEC DIS 9945-2: 1992, Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities (which is identical to IEEE Std 1003.2-Draft).

JIS X 0201-1976
: Japanese Industrial Standard, Code for Information Interchange.

JIS X 0208-1990
: Japanese Industrial Standard, Code of the Japanese graphic character set for information interchange.

The following X / Open documents are referenced in this guide:

Curses
: X / Open Specification, February 1992, Supplementary Definitions, Issue 3 (ISBN: 1-872630-38-3, C213), Chapters 9 to 14 inclusive, Curses Interface; this specification was formerly X / Open Portability Guide, Issue 3, Volume 3, January 1989, XSI Supplementary Definitions (ISBN: 0-13-685850-3, XO / XPG / 89 / 004).

Languages
: Chapters 1 to 4 of X / Open Specification, 1988, 1989, February 1992, Programming Languages, Issue 3 (ISBN: 1-872630-39-1, C214); this specification was formerly X / Open Portability Guide, Volume 4, August 1988 (ISBN: 0-13-685868-6, XO / XPG / 89 / 004).

Migration Guide
: X / Open Guide, July 1992, XPG3-XPG4 Base Migration Guide (ISBN: 1-872630-49-9, G204).

XPG3
: X / Open Portability Guide Issue 3, December 1988
ISBN: 0-13-685819-8 (set of seven volumes)

XPG4
: A document set defining XSI Operating System requirements comprising:

  XBD
  : X/Open CAE Specification, July 1992, System Interface Definitions, Issue 4 (ISBN: 1-872630-46-4, C204).

  XCU
  : X/Open CAE Specification, July 1992, Commands and Utilities, Issue 4 (ISBN: 1-872630-48-0, C203).

  XSH
  : X/Open CAE Specification, July 1992, System Interfaces and Headers, Issue 4 (ISBN: 1-872630-47-2 C202).

Chapter 1

# Introduction

This chapter describes the background to the internationalisation of computer systems. Internationalisation is the development of language-independent systems and software. Localisation is the definition of a specific environment such that an internationalised program performs in the manner expected by a user, in terms of language, cultural data and so on.

## 1.1 Background

X/Open members market systems in many different countries. Customers and users speak different languages, conform to different cultural conventions and business practices, and in many cases are required to cope with multiple language variations on a single system.

It was in response to these requirements that X/Open set out to define a set of system facilities that would enable:

- programmers to develop internationalised software
- system suppliers to define multiple instances of localisation data
- users to select, either individually or collectively, the specific language operation of system commands and applications

Previously, UNIX systems and most systems derived from them had been based on ASCII[1] and American English. No facilities were provided for dealing with other coded character sets, nor for supporting different languages or cultural conventions.

The internationalisation facilities defined in the **XSH** specification provide enhancements to the system that enable character-based data to be processed in a way that is independent of the underlying character encoding. This allows complete flexibility in the choice of coded character sets supported by an implementation. X/Open systems also allow program messages to be handled in the native language[2] of each user, as well as supporting different formats for cultural data such as date and time formats, currency values and so on.

### 1.1.1 External Influences

Various initiatives have been under way for some time to provide the tools for developing internationalised software in industry-standard operating environments. X/Open has been cognisant of these developments and has aimed to maintain the X/Open definition in line with standards laid down by external bodies.

In particular, the ISO C standard defines functions for locale-dependent character classification, case conversion and string collation. It also defines functions for processing cultural data, and an initialisation function that binds the required localisation data to a program's operational environment. The ISO C standard further defines a basic set of multi-byte functions for handling coded character sets larger than eight bits.

---

1. American Standard Code for Information Interchange. ASCII defines 128 characters, including both control characters and graphic characters, represented by 7-bit binary values.
2. A computer user's spoken or written language, as opposed to a computer programming language.

The ISO POSIX-1 standard extends the ISO C provisions by defining a set of environment variables and locale categories common to POSIX-based systems.

The UniForum Technical Subcommittee on Internationalisation has, in collaboration with X/Open, developed a formal definition for internationalised regular expressions. This has been submitted to the IEEE and is published in the ISO POSIX-2 DIS.

In 1985, the Japan UNIX Advisory Group, mindful that additional interfaces were needed to support codesets larger than eight bits, proposed a set of multi-byte specifications which were subsequently adopted by SIGMA and various leading system suppliers. The **Worldwide Portability Interfaces** published in the **XSH** specification were developed primarily from the SIGMA specifications and include most of the interfaces proposed to ISO by ITSCJ SC22/WG14 C/SWG. This emerging standard is formally known as the MSE working draft. The ISO MSE standard when finalised will be incompatible with XPG4. Where the documents do not agree, X/Open's intention is to adhere to ISO MSE.

X/Open incorporates all of the above, as well as defining a native language message system and extended C library capability in other key areas.

### 1.1.2 Historical Development

#### X/Open Portability Guide, Issue 1 (July 1985)

No Internationalisation facilities were published in this issue of the **X/Open Portability Guide** (XPG). However, it did include a statement of direction to the effect that this area of system operation would be addressed in future issues.

#### X/Open Portability Guide, Issue 2 (January 1987)

This issue contained a trial-use definition of the **X/Open Native Language System** interfaces, which were derived from the **Native Language Support** system developed by the Hewlett-Packard Company of Palo Alto, California. This was enhanced by X/Open and modified in strategic areas to resemble more closely facilities proposed in the ANSI C standard.

Specifically, XPG2 provided a definition of the following facilities:

- transparent single-byte codeset operation
- internationalised versions of library functions
- new functions that allowed applications to determine the format and setting of locale-specific cultural data items
- message catalogues
- an announcement mechanism

#### X/Open Portability Guide, Issue 3 (December 1988)

For the first time Internationalisation facilities were defined as mandatory in the **X/Open System Interface**. The XPG was also fully aligned with the POSIX.1-1988 standard and ANSI C standard, except in the area of multi-byte codeset operation and *localeconv*( ).

XPG3 included all the facilities presented in XPG2, modified for alignment with the above standards, plus:

- an improved announcement mechanism
- internationalised regular expressions
- an optional Internationalised utility environment
- a new utility for codeset conversion

**XPG4 Base CAE Specifications (July 1992)**

The main changes in this issue are support for Asian languages and a more comprehensive definition of the **Internationalised Utility Environment**. Interfaces have also been added that provide codeset conversion facilities at the program level.

In summary, the **XPG4** base specifications include all the facilities published in XPG3, plus:

- **Worldwide Portability Interfaces**, which enable applications to work with either single- or multi-byte codesets
- extended support of the Internationalised Utility Environment
- additional system interfaces for date and time conversion (*strptime*( )), monetary value conversion (*strfmon*( )), and codeset conversion (*iconv**( ))
- full conformance to the ISO C standard
- the *localedef* and *locale* utilities
- changes to the *catopen*( ) *oflag* argument.

## 1.2 Internationalisation

Internationalisation refers to the process of developing programs without prior knowledge of the language, cultural data or character encoding schemes they are expected to handle. In system terms, it refers to the provision of interfaces that enable internationalised programs to modify their behaviour at run time for specific language operation.

### 1.2.1 Language

An internationalised program should make no assumptions about the language of character data (text) that it is designed to handle. This refers to data generated internally, extracted from or written to file store, and message text used for communication with a user.

In particular, language has implications for the processing of text for such things as character handling and word ordering. X/Open systems provide interfaces that enable internationalised programs to manipulate text strings according to the language requirements of individual users (see Chapter 4).

With regards to message text, facilities are provided that enable program messages to be separated from the code, translated into different languages, and accessed by the program at run time. These facilities are described in Chapter 3.

### 1.2.2 Cultural Data

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

An internationalised program cannot assume these formats in advance and must use facilities provided by the underlying operating system to determine their setting at run time. X/Open systems provide this capability by defining a language information database that applications can interrogate for the required format and settings of cultural data items. This facility and associated functional interfaces are described in Section 4.5 on page 56.

### 1.2.3 Character Sets

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

For a program to be able to handle text recorded in different coded character sets, it must not make assumptions about the size or bit assignment of character encodings. In particular it should not be assumed that any part of an area used to store a character code is available for other uses.

X/Open systems provide codeset independent facilities for character classification and case conversion as described in Section 4.2 on page 48 and Section 4.3 on page 51.

## 1.3 Localisation

Localisation refers to the process of establishing information within a computer system specific to each supported language, cultural data and coded character set combination. Each such combination gives rise to the definition of one locale.

How and where localisation data is stored is not defined by X/Open, nor are the permitted settings of a locale name. However, there are implications for what type of information a locale should contain, and the interface functions that utilise this information are defined.

### 1.3.1 Collating Sequence

Each locale contains collating sequence information that informs string compare functions (see Section 4.4 on page 52) about the relative ordering of characters defined in the associated coded character set. This ordering may be defined by underlying hardware, or it may be implemented totally at the software level. Internationalised regular expressions also use this information for implementing character ranges, collating symbols and equivalence classes (see Section 5.3 on page 97).

### 1.3.2 Character Classification

Character classification information provides details about the type of character associated with each legal character code; that is, whether it is an alphabetic, upper-case, lower-case, punctuation, control or space character. This information is used by character classification functions (see Section 4.2 on page 48) and by internationalised regular expressions to implement character classes.

### 1.3.3 Case Conversion

This refers to information identifying the possible other case of each legal character code. It is used by case conversion functions (see Section 4.3 on page 51) to shift from upper-case to lower-case, and vice versa.

### 1.3.4 Language Information

Language information refers to localisation data describing the format and setting of locale-specific cultural data. This data and the functions provided for interrogating it are described in Section 4.5 on page 56.

### 1.3.5 Message Catalogues

A message catalogue is a file or storage area containing program messages, command prompts, and responses to prompts for a particular language. The X/Open messaging system is described in Chapter 3.

## 1.4 Language Announcement

Internationalisation and localisation facilities are provided primarily for use by system and application developers. All a user requires to see of these is the end result; that is, programs working in his or her own language and culture.

X/Open defines an announcement mechanism whereby language, cultural data and codeset requirements can be set by users individually. This is done by setting the required locale name in a set of reserved environment variables. These are read by internationalised applications whenever they are executed, and are used to attach the associated instance of localisation data to the program's operational environment.

Thus the announcement mechanism provides the means by which users can bind specific language information, message text, and so on to each invocation of an internationalised application. This binding can be general and direct that an application should work exclusively with one set of localisation data, or it can be selective and request that different aspects of program operation use different localisations.

Facilities and details of the announcement mechanism are described in Section 5.1 on page 87.

## 1.5 Terms and Definitions

At this point it is worth defining a number of fundamental terms used extensively in this guide. Other, less common terms are defined when they are first encountered in the text.

A *character* is a sequence of one or more bytes representing a single graphic symbol or control code. This should not be confused with the **char** data type in the C programming language, which is defined as an object large enough to store any member of the basic execution character set, usually mapped as an 8-bit value. A character can be represented by single-byte or multi-byte values.

The terms character and *multi-byte* character are synonymous; that is, they both refer to character values of any length, including single-byte.

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

A *wide character* is an integral type large enough to hold any member of the extended execution character set. In program terms, it is an object of type **wchar_t**, which is an implementation-defined integral type defined in the header **<wchar.h>**. It is permitted for implementations supporting only single-byte codesets to define this as a byte value.

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

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

### 1.5.1 Portable Character Set

The **XPG4** base specifications define a Portable Character Set, which is supported in both the source (compile-time) and execution (run-time) environments. The Portable Character Set contains the 26 upper-case letters of the English alphabet:

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

the 26 lower-case letters of the English alphabet:

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

the 10 decimal digits:

```
0 1 2 3 4 5 6 7 8 9
```

the following 32 graphic characters:

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

and the space character, and control characters representing alert, backspace, tab, newline, vertical-tab, form-feed, carriage return and the null character, NUL.

The **Portable Character Set** is similar to the basic source and basic execution character sets defined in the ISO C standard, except that the X/Open version also includes the dollar-sign, commercial-at and grave-accent characters.

In cases where substitutions are made for any of the above characters (for example, in ISO 646 variants), the substituted character has the same syntactic meaning as the character it replaces in the Portable Character Set.

*Chapter 2*

# *Developing Internationalised Software*

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

Programs intending to support multi-language operation must also take into account the fact that languages can be represented within the computer system by one or more coded character sets. Because of the requirements of different languages, these may vary in both size (8-bit, 16-bit, and so on) and binary representation.

Provision for satisfying the above requirements can be made by writing programs that make no hard-coded assumptions about language, cultural data or character encodings. Such a program is said to be internationalised. Data specific to each supported language, territory and codeset combination are held separate from the program code and can be bound to its run-time environment by language initialisation functions.

X/Open systems provide facilities for developing internationalised software, for defining localisation data, and for announcing specific language requirements.

The following facilities are provided for program development:

- library functions that provide language and codeset independent character classification, case conversion, number format conversion and string collation
- library functions that enable programs to determine cultural and language specific data dynamically
- a message system that allows program messages to be held apart from the program code, translated into different languages, and retrieved by a program at run time
- an initialisation function that initialises a program locale according to the stated language requirements of each user
- library functions that allow applications to handle extended character codes

The remainder of this chapter describes each of the above provisions in greater detail. This is done from a programming viewpoint, highlighting areas of program operation that need to be considered when developing internationalised software, and drawing the reader's attention to specific facilities offered by X/Open systems.

## 2.1 Character Sets

In the past most UNIX and UNIX-like systems have been based on the 7-bit ASCII coded character set. Most languages other than English require characters additional to those catered for by ASCII, and some require an alternative (for example, Arabic) or very large (for example, Japanese) set of characters.

There is a set of ISO codesets covering the major European languages. Several of these cater for the mixing of major languages within a single codeset. These codesets also have the advantage that they include the ASCII codeset within them, making it possible for systems to support languages other than English without invalidating existing, non-internationalised software.

All the ISO 8859 codesets include ASCII as a proper codeset within them:

ISO 8859-1: Latin 1 - Western European Languages.
ISO 8859-2: Latin 2 - Eastern European Languages.
ISO 8859-3: Latin 3 - Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish and Turkish.
ISO 8859-4: Latin 4 - Danish, English, Estonian, Finnish, German, Greenlandic, Lappish and Latvian.
ISO 8859-5: Latin/Cyrillic - Bulgarian, Byelorussian, English, Macedonian, Russian, Serbo-Croat and Ukranian.
ISO 8859-6: Latin/Arabic - Arabic.
ISO 8859-7: Latin/Greek - Greek.
ISO 8859-8: Latin/Hebrew - Hebrew.
ISO 8859-9: Latin 5 - Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish and Turkish.
ISO 8859-10: Latin 6 - Danish, English, Estonian, Finnish, German, Greenlandic, Icelandic, Sami (Lappish), Latvian, Lithuanian, Norwegian, Faroese and Swedish.

Another codeset that may be supported on many systems is that defined in the ISO 6937:1983 standard[3]. This is the normal transmission codeset for X.400 and ISO mail systems.

The **XPG4** base specifications make no specific recommendations about which codesets must be supported on conforming systems; that is, implementations are free to support whichever codeset or codesets they choose.

### 2.1.1 Data Transparency

Given the above rationale, it is evident that internationalised software must be written to cater for a potentially wide variety of character encoding schemes. It must not be assumed that a particular codeset is supported on all X/Open systems, nor that individual character encodings occupy a fixed number of bits.

Another legacy of the historical dependence of UNIX systems on 7-bit ASCII is that some programs use the most significant bit of a byte for their own internal purposes. Arguably this was always a dubious programming practice, although it was quite safe while characters in the underlying codeset were always mapped to 7-bits. In the 8-bit or larger world of international codesets, this practise is clearly untenable and should be avoided at all costs.

---

3. ISO 7-bit or 8-bit coded character set for text communication using public communication networks, private communication networks, or interchange media such as magnetic tape and disks.

### 2.1.2 Code Literals

Another area to be wary of when writing internationalised software is the use of in-code literals. The danger of statements such as:

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

are obvious. This piece of code assumes that lower-case **a** is always represented by a fixed octal value, which may not be true for all supported codesets. A better way to make the above test is to use a character constant,

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

although even here there is a need for caution.

By using the *getchar*( ) function, which operates on bytes, the above code would not work correctly if the next character in the input stream was a multi-byte value.

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

uses the wide-character function *getwchar*( ), which works correctly with any codeset because 'a' is a member of the portable character set and is transformed into the same wide character code in all locales.

The **X/Open Portability Guide** further defines that each source character set member and escape sequence in character constants and string literals is converted to the same member of the execution character set in all locales. Thus it is safe to use any of the characters in the Portable Character Set, either as character constants or within string literals, but it is not guaranteed that non-English characters are translated correctly.

As an example, in a statement of the form:

```
if ((c = getwchar()) == L'á')
```

the accented character **á** may not be representable in either or both of the source and execution character sets, or its binary value may not be translatable from one to the other. The results of specifying such a constant in a source program are undefined.

To make such a test, programmers are advised to use the message system and read the character value from localisation data, for example:

```
char     *schar;
wchar_t  wchar;

...
schar = catgets(catd, NL_SETD, MSG_ID, "a");

if (mbtowc(&wchar, schar, MB_CUR_MAX) == -1)
         error();

if ((c = getwchar()) == wchar)
...
```

where MSG_ID identifies a message string containing the single character **á**. The conversion to wide character is necessary because *catgets*( ) may return a multi-byte sequence.

Message catalogues are bound to a program at run time and are locale-specific. Also in the above example, if the accented character is not available in the current codeset, the test is made against the base character.

### 2.1.3 Multi-byte Characters

X/Open systems may support multi-byte codesets for use with Asian languages. No requirement is made that all X/Open systems must support this feature, although they must all provide the interfaces described in Chapter 4, even if on some implementations these are merely synonyms for the associated byte functions.

Programmers should be aware that both the source and execution character sets may contain multi-byte characters. The encoding need not be the same, but both must observe the following rules:

- The characters defined in Section 2.1.2 on page 11, must be present in both codesets.
- The existence, meaning and encoding of any additional members are locale-specific.
- A character may have a state-dependent encoding. Each sequence of characters begins in an initial shift state and enters other implementation-defined shift states when specific characters are encountered in the sequence.
- While in the initial shift state, all characters from the basic character set retain their usual interpretation and do not alter the shift state.
- The interpretation for subsequent bytes in the sequence is a function of the current shift state.
- A byte with all bits zero is interpreted as a null character, independent of shift state.
- A byte with all bits zero must not occur in the second or subsequent bytes of a multi-byte character.

A source character set must also observe the following:

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

### 2.1.4 Trigraphs

X/Open C Language compilers also support trigraph sequences. These are designed to allow users to input the full range of basic characters even if their keyboards do not support the full source character set. The following trigraph sequences are currently defined, each of which is replaced by the corresponding single character:

| trigraph sequence | single character |
|---|---|
| ??= | # |
| ??( | [ |
| ??/ | \ |
| ??) | ] |
| ??' | ^ |
| ??< | { |
| ??! | \| |
| ??> | } |
| ??- | ~ |

### 2.1.5 Classification

Another feature of program operation that is dependent on the codeset is character classification; that is, determining whether a particular character code refers to an upper-case alphabetic, lower-case alphabetic, digit, punctuation, control or space character.

In the past, many programs classified characters according to whether the character's value fell between certain numerical limits. For example:

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

tests for all upper-case characters. This statement is perfectly valid in the ASCII world, where all upper-case letters have values in the range 0x41 to 0x5a (A-Z). However, it is not valid in the ISO 8859-1: 1987 standard world for example, where upper-case letters occupy the ranges 0x41 to 0x5a, 0xc0 to 0xd6, and 0xd8 to 0xdf. In EBCDIC, character values are different again, and in this case even the upper-case English letters have a different encoding.

In internationalised software, all character classification should be done by calling the appropriate internationalisation function, for example:

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

These functions classify wide character code values according to **type** information in the program's locale, and thus are independent of the language and codeset. The character classification functions currently defined by X/Open are described in Section 4.2 on page 48.

### 2.1.6 Conversion

Similarly, case conversion of ASCII characters can be done with a statement of the form

```
a |= 0x20;
```

which converts the character in *a* to lower-case. Alternatively,

```
a &= 0xdf;
```

converts the character in *a* to upper-case.

This code is unsafe in internationalised software because:

- It assumes ASCII-coded character values.
- It does not check for valid input.

The correct way to perform these operations is to call

```
a = towlower(a);
```

for conversion to lower case, and

```
a = towupper(a);
```

for conversion to upper case. Both functions use information defined in the program locale and are independent of the codeset. They also check that the input is valid, returning the argument unchanged if it is not. The case conversion functions are described in Section 4.3 on page 51.

### 2.1.7 String Comparison

UNIX systems have always provided functions for comparing character strings, for example:

```
int strcmp(s1, s2)
```

which compares strings *s1* and *s2*, returning an integer greater than, equal to or less than zero, depending on whether *s1* is greater than, equal to or less than *s2* in the machine collating sequence.

However, certain languages also require that X/Open systems support multi-pass collation algorithms,

- for ordering accented characters within a **character class** (for example, **a**, **á**, **à**, and so on)
- for collating certain n-character sequences as a single character (for example, **ch** in Spanish which collates after **c** and before **d**)
- for collating certain single characters as a 2-character sequence (for example, a sharp **s** in German which collates as **ss**)
- for ignoring certain characters altogether for collation purposes (for example, hyphens in dictionary words).

String comparison in an international environment is thus dependent on the codeset and language, requiring that additional functions be defined that compare strings according to collating sequence information in the program's locale.

A number of such functions are defined in the **XSH** specification, including:

- *wcscoll*( ), which provides similar capability to *strcmp*( ), except that it operates on wide characters and uses locale-specific collating information
- *wcsxfrm*( ), which transforms a wide-character string using collating sequence information in the program's locale such that the resulting string can be compared using the *wcscmp*( ) function

  Note, however, that if the two strings are compared only for *equality*, the function *strcmp*( ) or *wcscmp*( ) can still be used, and they are faster in most environments than *strcoll*( ) or *wcscoll*( ).

A comprehensive description of these functions, complete with examples, is given in Section 4.4 on page 52.

## 2.2 Cultural Data

Cultural data refers to items of information that can vary from one language to another, or between geographic areas. For example:

- In the U.K. a period is used to represent the radix character in decimal numbers, and a comma is used as the thousands separator. The same two characters are used in Germany, but with exactly the opposite meaning. Thus, 2,345.77 in the U.K. is the same as 2.345,77 in Germany.
- Even in countries where the same language is spoken, conventions for representing certain cultural data items may vary. In the U.S.A., October 7, 1986 is represented as 10/7/86, whereas in the U.K. it is represented as 7/10/86.
- As well as year, month and day, separating delimiters may also vary between countries. In Germany, the above date is represented as 7.10.86.
- Currency symbols vary both in terms of the character or characters used and with regards to the position these characters occupy within a currency value; that is, they can precede, follow or appear within the value.

An internationalised program cannot make assumptions about this type of data. Instead, it must localise its operation according to the language and local customs of each user.

### 2.2.1 The langinfo Database

X/Open systems define a language information database (**langinfo**) which contains details of cultural data items for each supported locale. At least the following data is available on all X/Open systems:

- codeset name
- date and time formats
- names of the days of the week
- names of the months of the year
- abbreviated day names
- abbreviated month names
- radix character
- thousands separator character
- affirmative and negative responses for yes/no queries
- currency symbol and its position within a currency value
- Emperor/Era name and year.

### 2.2.2 The nl_langinfo() Function

Cultural data items can be extracted from the language information database by calling the *nl_langinfo*( ) function. This takes an argument *item*, which is a constant defined in **<langinfo.h>**, and returns a pointer to a string containing the associated data.

For example:

```
nl_langinfo(D_T_FMT);
```

extracts the string for formatting date and time information.

### 2.2.3 The strftime( ) Function

Other, higher level functions are also provided that allow internationalised programs to make indirect use of the language information database. The *strftime*( ) function, for example, allows applications to generate date and time strings that observe the local customs of the caller.

In the following example, the *strftime*( ) function is used to generate a date string as defined by item D_FMT in the current language information database.

```
setlocale(LC_ALL, "");

clock = time((long*) NULL);
tm    = localtime(&clock);

strftime(buf, size, "%x", tm);
puts(buf);
```

The *buf* argument is a pointer to a string, of up to *size* bytes, in which the date string is returned. The next argument is a format string containing conversion specifications. In the above example, **%x** is replaced in the output string by the locale's appropriate date representation. Finally, *tm* is a pointer to a time structure.

The following example is similar to the previous example. The capability illustrated is how one might use *nl_langinfo*( ) and *strftime*( ) in combination:

```
setlocale(LC_ALL, "");

clock = time((long*) NULL);
tm    = localtime(&clock);

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

where *strftime*( ) is used to generate a date and time string obtained by *nl_langinfo*( ) from the D_T_FMT item in the current locale.

### 2.2.4 The strfmon( ) Function

Similarly, the *strfmon*( ) function is provided for the formatting of monetary values, according to language information in the current locale.

For example:

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

formats the double argument *value* according to the locale's national currency format (for example, $1,234.56).

### 2.2.5 The localeconv( ) Function

The *localeconv*( ) function is a more primitive interface that provides formatting details of numeric quantities, monetary or otherwise, sufficient to enable applications to perform their own conversions.

```
struct lconv *lconv = localeconv();
```

returns a pointer to a structure containing separate fields that describe all aspects of number formatting in the current locale. This information includes:

- radix character
- thousands separator character
- digit grouping size
- international currency symbol
- local currency symbol
- radix character for monetary values
- thousands separator character for monetary values
- digit grouping size for monetary values
- positive sign
- negative sign
- number of fractional digits to be displayed
- monetary symbol placement
- parenthesis symbols for negative-valued monetary quantities.

### 2.2.6 Other Functions

Various other functions make use of the language information database to determine the setting of specific items of cultural data. For example, the *scanf*( ), *printf*( ) and *wcstod*( ) functions use it to determine the current radix character.

## 2.3 Language

There are various implications for internationalised software with respect to the language of program messages, text processed by an application, and the presentation of information to a user.

### 2.3.1 Messages

As already mentioned, one of the major requirements for internationalisation is that applications should communicate with users of the system in their own language. This places a number of constraints on the development of internationalised programs. For example:

- To cater for multi-language operation, program messages should be defined independently of the program source and should not be compiled into object programs.
- Once extracted, program messages can be translated into different languages and stored in a form suitable for linkage to the program at run time.
- Programs can then retrieve appropriate message text translations according to the stated language requirements of each user.

X/Open systems support a native language message system that contains a definition of message text source files, a command *gencat* to generate message catalogues, and a set of C library functions to retrieve individual messages from one or more message catalogues at run time.

As an example, the following code provides a simple illustration of how message text can be retrieved by an internationalised program:

```
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>
#include "prog.h"

main( )
{
       nl_catd catd;

       setlocale(LC_ALL, "");

       catd = catopen("prog", NL_CAT_LOCALE);

       puts(catgets(catd, SETN,
              HELLO_MSG, "hello world"));

       catclose(catd);
}
```

NL_CAT_LOCALE is defined in **<nl_types.h>**. Note that the *catgets*( ) function allows one set of message localisations to be held with the program source. The constants SETN and HELLO_MSG are set and message identifiers assumed to be defined in the header **prog.h**.

The *catopen*( ) function opens a message catalogue for use by *catgets*( ). Access to the required message catalogue is steered by the value of the *NLSPATH* environment variable, which contains a search path and templates for locating and naming message catalogues. If this environment variable does not exist, or if a message catalogue cannot be found a default path is used. The *oflag* argument affects the way *NLSPATH* substitutions for %L, %l, %t and %c are performed (see Section 3.5.1 on page 33) as follows:

- If *oflag* equals 0, the setting of the *LANG* environment variable is used for *NLSPATH* substitutions.
- If *oflag* equals NL_CAT_LOCALE, the setting of the LC_MESSAGES locale category is used for these substitutions.

In the latter case, *setlocale*( ) must be called before *catopen*( ) to ensure the locale is initialised correctly.

The settings of these are used in conjunction with the first argument to *catopen*( ) to generate a message catalogue name. Once opened, individual messages are identified by means of a set identifier (*set_id* argument) and a message identifier (*msg_id* argument).

Features and capabilities of the message system are explained in Chapter 3.

### 2.3.2 Formatted Output

Not only does internationalised software need to be aware that the language of program messages may vary, but also that the construction of message strings may be affected by local customs. For example, the statement,

```
printf("%s owned by %s\n", s1, s2);
```

assumes both an English language locale and a particular language construction. Translations into different languages can be handled by locating the message string in a message catalogue and reading the required translation at run time, as follows:

```
printf(catgets(catd, set_id, msg_id,
       "%s owned by %s\n"), s1, s2);
```

However there is still a problem. Suppose that, in certain languages, translation requires the words pointed to by *s1* and *s2* to be included in the output string in a different order (that is, *s2* then *s1*). The argument list in the above example imposes a fixed ordering on substitutions in the output string, which cannot be overcome by the use of message catalogues alone.

To overcome this type of problem, X/Open has extended the definition of *printf*( ) format specifiers such that conversion can be applied to the **n**th argument in an argument list, rather than to the next unused argument. In this case, the % conversion character is replaced by the sequence %*digit*$, where *digit* is a decimal value giving the position of the required argument in the argument list.

Using this feature, localisations of the above format string can be produced for any language and cultural-data combination. For example,

```
"%1$s owned by %2$s\n", or
"%2$s owns %1$s\n"
```

In format strings containing the %*digit*$ form of a conversion specification, a field width or precision may be indicated by the sequence **digit*$.

### 2.3.3 Formatted Input

Similar facilities are defined for the *scanf*( ) family of functions. The conversion character % is again replaced by the sequence %*digit*$, where *digit* is a decimal value giving the position of the associated argument in the argument list.

For example,

```
int day, month, year;

scanf(catgets(catd, NL_SETD, MSG_ID,
       "%d/%d/%d"),
       &month, &day, &year);
```

where NL_SETD is a default message set number defined in **<nl_types.h>** and the setting of the message identified by MSG_ID is:

```
American English:  "%1$d/%2$d/%3$d"   (for example, 10/7/89)
British English:   "%2$d/%1$d/%3$d"   (for example, 7/10/89)
German:            "%2$d.%1$d.%3$d"   (for example, 7.10.89)
```

## 2.4 Initialisation

To recap, Internationalisation is the process of developing software such that it can be adapted to meet the needs of different languages, cultures and coded character sets. Localisation is the process of establishing information in the computer system specific to each supported language, territory and codeset combination.

For an internationalised program to function correctly, the required localisation data must therefore be bound to its operational environment at run time. On X/Open systems, this is achieved by calling the *setlocale*( ) function as follows,

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

where *category* names all or part of the program locale, and *locale* specifies the required locale name.

### 2.4.1 General Locale Setting

The announcement mechanism defines a set of reserved environment variables in which users can identify their localisation requirements. This is done by assigning a locale name to the appropriate environment variable, for example:

```
LANG=locale-name
```

Note that settings of *locale-name* are not fixed by X/Open and may vary from one system to another.

To initialise its locale from the above, an internationalised application would call *setlocale*( ) as follows:

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

The empty string in the second argument instructs *setlocale*( ) to initialise the program's locale from the current settings of the international environment variables. The constant LC_ALL names the entire locale (see Section 4.1 on page 42).

### 2.4.2 Internal Locale Settings

Alternatively, programs may want to prompt the user for a locale name directly, or allow users to change localisation data in mid-program. In this case, the locale can be initialised or reinitialised by calling *setlocale*( ) as follows:

```
nl_catd  catd;
char     buf[BUFSIZ];

setlocale(LC_ALL, "");
catd = catopen(NAME, NL_CAT_LOCALE);

printf(catgets(catd, NL_SETD, MSG,
         "Give locale-name: "));
gets(buf);

setlocale(LC_ALL, buf);
```

In this example the second argument contains the *locale-name* directly. NAME and MSG are constants assumed to be defined elsewhere in the program.

**Note:** The important thing for programmers to remember is that *setlocale*( ) is the only means by which localisation data is bound to the program locale. Thus, internationalised

programs do not function correctly, in terms of language, territory and codeset operation, until *setlocale*( ) has been called successfully.

*Chapter 3*

# *The Message System*

This chapter explains the facilities offered by the X/Open Message System. This is done by describing the actions that need to be taken to internationalise a non-trivial application, including:

- message extraction
- message translation
- generating message catalogues
- message catalogue access
- reading program messages.

The application used to illustrate this process is a simple database program, comprising several source modules and a menu-driven front end. The application makes extensive use of the *curses* library package and is designed to operate with character-based terminals.

Before describing functional capabilities of the message system however, it is first necessary for readers to understand the syntax and semantics of message text source files.

## 3.1 Message Text Source Files

A message text source file is made up of a series of numbered messages and numbered message sets, containing native language translations of the message text associated with an application. Each localisation of message text is normally held in a separate source file.

### 3.1.1 General

The fields of a message text source line are separated by a single space or tab character, as defined by the source codeset. Any other space or tab characters are considered to be part of the subsequent field, although use of the **quote** directive may modify this rule.

Empty lines in a message text source file are ignored. The effects of lines starting with an invalid character are implementation-defined.

Text strings can contain ordinary characters, and the special characters and escape sequences listed in the following table:

| Description | Symbol | Sequence |
|---|---|---|
| newline | NL(LF) | \n |
| horizontal tab | HT | \t |
| vertical tab | VT | \v |
| backspace | BS | \b |
| carriage return | CR | \r |
| form feed | FF | \f |
| backslash | \ | \\ |
| bit pattern | ddd | \ddd |

The escape sequence **\ddd** consists of backslash followed by one, two or three octal digits, which are taken to specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored.

Backslash followed by a newline character is also used to continue a string on the following line, for example:

```
1      This line continues \
to the next line
```

describes a single message equivalent to,

```
1      This line continues to the next line
```

### 3.1.2 The set Directive

```
$set n comment
```

This directive specifies the set identifier of subsequent messages until another **$set**, **$delset** or end-of-file appears. The *n* denotes the set identifier, which is a number in the range [1-{NL_SETMAX}] (defined in **<limits.h>**. Set identifiers must be presented in ascending order within a single source file but need not be contiguous. Any characters following *n* are treated as a comment.

If no **set** directive is specified in a source file, all messages are assigned to a default message set. The identifier of the default message set is defined by the constant NL_SETD in the header **<nl_types.h>**.

The set mechanism is provided to aid the development of large, multi-module applications. A common use is to assign a set identifier for each source module, thus allowing message numbers in each module to run consecutively from the first. This reduces the amount of time and effort required to administer the allocation of message numbers.

### 3.1.3 The delset Directive

```
$delset n comment
```

This directive causes message set *n* to be deleted from an existing message catalogue. The *n* denotes the set identifier and is a number in the range [1-{NL_SETMAX}]. Any string following the set number is treated as a comment.

### 3.1.4 Message Directives

```
m message-text
```

The *m* denotes the message identifier and is a number in the range [1-{NL_MSGMAX}]. The *message-text* is stored in a message catalogue with the set identifier specified by the last **$set** directive, or with set identifier NL_SETD if none has been specified, message number *m*.

If *message-text* is empty, and a space or tab field separator is present, an empty string is stored in the message catalogue. If *message-text* is empty and *m* is not followed by a space or tab character, the existing message with that number (if any) is deleted from an existing message catalogue.

Message identifiers must be presented in ascending order within a single source file but need not be contiguous. The total length of *message-text* must not exceed {NL_TEXTMAX} bytes.

### 3.1.5 Comments

```
$ comment
```

A line beginning with **$**, followed by a space or tab character, is treated as a comment. No further interpretation of the line takes place.

### 3.1.6 Quoting

```
$quote c
```

This directive allows a quote character to be specified, which can be used to surround *message-text* so that trailing spaces or empty (null) strings are visible in a message text source file.

By default, or if an empty **$quote** directive is specified, no quoting of *message-text* is recognised.

## 3.2 Programming Using Message Files

The first action a programmer should perform is to identify all string literals in the program source that are in some way language-specific. These should be moved from the program source to a message text source file, where they can be edited or translated to different languages.

Some implementations may provide tools to assist in this process, although no such tools are currently defined by X/Open. This section describes how to build message text source files manually. It is applicable to both the development of new applications and to the internationalising of existing applications.

### 3.2.1 Example of Program Referencing Separate Texts

The example database application using the curses library contains numerous messages that are displayed on the user's terminal, depending on the current state of operation. In particular, the main program loop contains code that displays a top-level menu:

```
clear( );

standout( );
printw("\t\t      Inquire,  Update,  Add");
printw("      \t\t");
standend( );

printw("\n\n\n\n");
printw("\t\t\t    1 - Add Record\n");
printw("\t\t\t    2 - Select Record\n");
printw("\t\t\t    3 - Search by Record\n");
printw("\t\t\t    4 - Search by Field\n");
printw("\t\t\t    5 - Sort by Record\n");
printw("\t\t\t    6 - Read TEXT File\n");
printw("\t\t\t    7 - Define Screens\n");
printw("\t\t\t    8 - Define Formulae\n");
printw("\t\t\t    9 - Recalculate\n");
printw("\t\t\t    ! - Execute command\n");
printw("\t\t\t    X - Quit\n\n");
printw("\t\t\t    >   Enter Selection ");
refresh( );
```

As can be seen, the program first clears the screen, displays a screen title in standout mode, and then builds the menu via calls to the *printw*( ) function. Each call to *printw*( ) includes a format string containing control characters, for placing the text, and ordinary characters making up the message.

To internationalise this code, the programmer must at least separate out the message text. The programmer could move the entire format string to a message catalogue, but as the control characters are unlikely to vary from one language to another, this might be wasteful.

### 3.2.2 Example of Related Texts (English Version)

Continuing with the database example, the programmer chose to extract only message text from the program source, giving rise to the following message text source file:

```
$      English message localisation

$quote "

$set 1
1      "Inquire, Update, Add "
2      "Add Record"
3      "Select Record"
4      "Search by Record"
5      "Search by Field"
6      "Sort by Record"
7      "Read File"
8      "Define Screens"
9      "Define Formulae"
10     "Recalculate"
11     "Execute command"
12     "Quit"
13     "Enter Selection "
14     "Database Name: "
15     "Filename: "
16     "Command: "
17     "[Hit any key to continue]"
18     "Sorting, please wait ..."
19     "File %s locked, please try later ..."

$set 2
1      "Record Number: "

$set 3
1      " U - Update, ENTER - Select Next, \
BREAK - Return"
2      "Insert record? "
3      "Delete record? "

$set 4
1      "Record number: "
2      "Selector: "
3      "Defined Fields"
4      "Field number: "
5      "Selector: "
```

There are a number of things worth noting about this example:

- The **$quote** directive has been used to declare double-quote marks as the quoting character. Thus all subsequent messages are delimited explicitly. This was deemed necessary in this case as some messages contain trailing blanks (for example, messages 1, 14, 15 and 16 in set number 1), which the programmer required to be visible in the source file.
- Multiple sets have been used, one per source file. (Note that no fixed usage of the set mechanism is defined by X/Open.)
- In message lines, the field separator between the message number and message text is a single tab character, not multiple spaces.
- A backslash (\) at the end of line 1, set 3, has been used to continue the message on another line.

### 3.2.3 Editing Message Files

Message text source files can be edited using any text editor, provided that:

- The input device is capable of generating the necessary characters.
- If 8-bit or multi-byte characters are required, the editor is capable of transparently handling this data.

The **XPG4** base specifications define that *ed*, *ex* and *vi* satisfy the second of these requirements, although it should be remembered that *ex* and *vi* may not be available to users working on synchronous or network-connected terminals.

## 3.3 Message Translation

Once message text has been separated from the program source, it can be translated into different language and codeset combinations. For example, the following is a German language translation of message text for the database program:

```
$      German message localisation

$quote "

$set 1
1      "Suchen, Aktualisieren, Neuaufnehmen"
2      "Satz neuaufnehmen"
3      "Satz auswählen"
4      "nach Satz suchen"
5      "nach Feld suchen"
6      "Sätze sortieren"
7      "Datei einlesen"
8      "Bildschirm definieren"
9      "Formel definieren"
10     "Neu berechnen"
11     "Kommando ausführen"
12     "Beenden"
13     "Auswahl angeben "
14     "Name der Datenbasis: "
15     "Name der Datei: "
16     "Kommando: "
17     "[um weiterzumachen, drücken Sie Bitte \
irgendeine Taste ... ]"
18     "Ein bischen Geduld, es wird sortiert ..."
19     "Die Datei %s ist gelockt, \
versuchen Sie später ..."

$set 2
1      "Nummer des Satzes: "

$set 3
1      " U - Aktualisieren, \
ENTER - nächste Auswahl, \
BREAK - Rückkehr"
2      "Satz einfügen? "
3      "Satz löschen? "

$set 4
1      "Satznummer: "
2      "Auswahlkriterium: "
3      "Definierte Felder"
4      "Feldnummer: "
5      "Auswahlkriterium: "
```

This source should be located in a different file from the original English text, and ultimately is compiled into a message catalogue associated with one or more German-language locales.

The following is a French language translation of message text for the database program:

```
$      French message localisation

$quote "

$set 1
1      "Recherche, Mise à jour, Ajout "
2      "Ajout d'un enregistrement"
3      "Choix d'un enregistrement"
4      "Recherche par enregistrement"
5      "Recherche par zone"
6      "Tri par enregistrement"
7      "Lecture du fichier"
8      "Définition d'écrans"
9      "Définition de formules"
10     "Recalcul"
11     "Exécution d'une commande"
12     "Fin"
13     "Votre choix "
14     "Nom de la base de données : "
15     "Nom du fichier : "
16     "Commande : "
17     "[Pressez une touche quelconque, \
pour continuer]"
18     "Tri en cours, attendez SVP ..."
19     "Le fichier %s est bloqué, \
essayez plus tard ..."

$set 2
1      "Numéro de l'enrégistrement : "

$set 3
1      " U - Mise à jour, \
ENTER - Choix du suivant, \
BREAK - Retour"
2      "Désirez-vous insérer l'enregistrement ? "
3      "Désirez-vous supprimer l'enregistrement ? "

$set 4
1      "Numéro de l'enregistrement : "
2      "Critère de sélection : "
3      "Champs définis"
4      "Numéro de zone : "
5      "Critère de sélection : "
```

## 3.4 Generating Message Catalogues

A message catalogue is a file or storage area containing program messages, command prompts and responses to prompts, one per supported language, territory and codeset combination. A message catalogue is thus an example of localisation data.

The location and format of message catalogues is implementation-defined. Generically they are direct access storage areas, individual elements of which (messages) are indexed by **set** and **message** number.

Message catalogues are created using the *gencat* command, which takes one or more message text source files and produces or modifies a message catalogue.

### 3.4.1 The gencat Command

```
gencat catfile msgfile...
```

The *gencat* command merges the message text source file or files *msgfile* into the message catalogue identified by *catfile*. The file *catfile* is created if it does not already exist. Otherwise, if it does exist, its messages are included in the new *catfile*. If set and message numbers collide, the message text defined in *msgfile* replaces existing message text currently contained in *catfile*.

For example, the command to compile the three message text source files for the database application is:

```
gencat /nlslib/english/prog.cat english/prog.msg
gencat /nlslib/german/prog.cat german/prog.msg
gencat /nlslib/french/prog.cat french/prog.msg
```

Note that the location of message catalogues, and conventions for naming them, are not defined by X/Open. The names used above are specific to a particular implementation and vary from system to system.

### 3.4.2 Multiple Source Files

The multiple source file capability of *gencat* allows message catalogues to be created from a related set of message text source files. For example, instead of locating all the messages for our database application in one source file, we could have used one file per message set. This can be useful when more than one programmer is working on a development.

To process multiple source files, use a command of the form:

```
gencat /nlslib/german/prog.cat german/*.msg
```

where ***.msg** expands into a list of file names.

### 3.4.3 Changing an Existing Message Catalogue

Another feature of *gencat* is that it can be used to modify an existing message catalogue. For example, if we wanted to delete message set number 4 from one of our database message catalogues:

```
gencat /nlslib/german/prog.cat /dev/tty
$delset 4
<EOF>
```

Considerable caution needs to be exercised when using this feature as the original version of the message catalogue is lost.

A more practical use of the facility is that it allows individual program messages to be updated on site. For certain types of application, this can provide a useful aid to system administration.

### 3.4.4 Portability

It should be noted that message catalogues produced by *gencat* are binary encoded and may not be portable between different types of systems. Message text source files should be portable, provided the X/Open recommendations for text transmission are followed.

## 3.5 Message Catalogue Access

Message catalogues are opened ready for use by calling the *catopen*( ) function, which locates the identified message catalogue according to search and naming rules defined in the *NLSPATH* environment variable.

### 3.5.1 NLSPATH

The names associated with message catalogues, and their location in file store, can vary from one system to another. Filename affixes are not standardised, and individual applications can choose catalogue names suited to their own specific needs. For example:

- Standard commands and utilities
  Message catalogues for these are probably located in a well-known system directory, one per supported locale (for example, in:

  **/<path-prefix>/<locale-name>/*.cat**

  where ***** is the command name).

- Applications
  The above structure may be inconvenient for applications, not least because there are no standards in this area. Instead, it may be easier for an application to locate its message catalogues in a single directory, with one message catalogue for each supported locale, for example:

  **/<path-prefix>/progname/<locale-name>**

Recognising both these requirements, X/Open defines an environment variable *NLSPATH*, which gives both the location of message catalogues, in the form of a search path, and naming conventions associated with message catalogue files.

*NLSPATH* contains a sequence of templates which the *catopen*( ) function uses when opening a message catalogue. Each template consists of an optional prefix, one or more substitution fields, a filename, and an optional suffix.

For example:

```
NLSPATH=/system/nlslib/%N.cat
```

defines that *catopen*( ) should look for all message catalogues in the directory **/system/nlslib**, where the catalogue name should be constructed from the *name* argument passed to *catopen*( ), with suffix **.cat**.

Substitution fields consist of a **%** symbol, followed by a single letter keyword. The following keywords are currently defined:

| Key | Meaning |
|---|---|
| %N | The value of the *name* argument passed to *catopen*( ) |
| %L | The value of the LC_MESSAGES category or *LANG* environment variable. |
| %l | The **language** element from the LC_MESSAGES category or *LANG* environment variable. |
| %t | The **territory** element from the LC_MESSAGES category or *LANG* environment variable. |
| %c | The **codeset** element from the LC_MESSAGES category or *LANG* environment variable. |
| %% | A single **%** character |

If the substitution value is not defined in the caller's environment (see Chapter 5 on page 87), an empty string is substituted. The field separator characters **_** (underscore) and **.** (period) are not included in %t and %c substitutions.

Templates defined in *NLSPATH* are separated by colons (**:**). A leading colon or two adjacent colons (**::**) is equivalent to specifying %N. For example:

```
NLSPATH=:%N.cat:/nlslib/%L/%N.cat
```

indicates that *catopen*( ) should search in

- **name**
- **name.cat**
- **/nlslib/<category>/name.cat**

*NLSPATH* is normally set up on a system-wide basis and should not need to be altered by users. Application developers need to be aware that they may have to add templates if private naming conventions are employed.

### 3.5.2 The catopen( ) Function

The *catopen*( ) function and the type **nl_catd** are defined in the header **<nl_types.h>**.

```
nl_catd catopen(const char *name, int oflag);
```

If successful, *catopen*( ) opens the message catalogue identified by the *name* argument and returns a valid message catalogue descriptor. This is passed as an argument to subsequent calls of *catgets*( ) and *catclose*( ).

If *name* contains a slash (*/*) character, it is assumed to contain a complete message catalogue name. Otherwise, *NLSPATH* is used with *name* substituted for %N. If *NLSPATH* is not defined in the environment, or if a message catalogue cannot be opened in any of the specified elements, an implementation-defined default path may be used.

If the value of the *oflag* argument is 0, the *LANG* environment variable is used to locate the catalogue without regard to the LC_MESSAGES category. If the *oflag* argument is NL_CAT_LOCALE, the LC_MESSAGES category is used to locate the message catalogue.

The *catopen*( ) function should be called immediately after having initialised the program locale. Until a message catalogue has been attached to the locale, program messages continue to be produced in the default language of the program (if any).

Programmers should note that a successful call to *catopen*( ) may result in the allocation of a file descriptor. This is taken from the pool of file descriptors available to the process and may have implications for other aspects of program operation. It also means that *catopen*( ) may fail if {OPEN_MAX} file descriptors are already allocated to the process.

How to use *catopen*( ) is largely a matter of personal preference, although one common usage is to pass the program name given in the argument list as the *name* argument:

```
catd = catopen(argv[0], NL_CAT_LOCALE);
```

although some caution is needed when doing this. The value pointed to by **argv[0]** normally contains a filename, but may contain a pathname complete with path-prefix.

If a call to *catopen*( ) is unsuccessful, the value (**nl_catd**)–1 is returned. Applications may choose to ignore this condition and allow processing to continue in the default language. In any event, terminating the program is problematic, as it is not apparent how to inform the user of the failure, given that message translations into the user's native language have not been attached to the program's locale.

### 3.5.3 The catclose( ) Function

This function and the type **nl_catd** are defined in the header **<nl_types.h>**.

```
int catclose(nl_catd catd);
```

The *catclose* function closes the message catalogue identified by *catd*. If a file descriptor is associated with an underlying catalogue file, this is also closed.

Open message catalogues are also closed by the *exit*( ) function when a process terminates.

## 3.6 Reading Program Messages

So far we have seen how to extract messages from the program source, how to generate message catalogues, and how to locate and open a message catalogue ready for use. All that remains is to describe how individual messages are read into a program.

### 3.6.1 The catgets( ) Function

This function and the type **nl_catd** are defined in the header **<nl_types.h>**.

```
char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

The *catgets*( ) function attempts to read message *msg_id*, from set *set_id*, in the message catalogue identified by *catd*.

The *catd* argument is a message catalogue descriptor returned by a previous call to *catopen*( ). The *set_id* and *msg_id* are integer values giving the set number and message number respectively of the required program message. The *s* argument is a pointer to a default message string.

If the call is successful, *catgets*( ) returns a pointer to the null terminated message string. If the call is unsuccessful for any reason, *s* is returned. This provides a mechanism whereby one set of localisation data can be held with the program source, thus allowing the program to operate meaningfully even if locale-specific message text is not defined.

One thing that programmers need to be wary of is that, if successful, *catgets*( ) reads the message string into an internal buffer area. This area may be overwritten by subsequent calls to *catgets*( ). If it is required to reuse a message, and there are intervening calls to *catgets*( ), the string should be preserved by moving it to another location.

The *set_id* and *msg_id* are defined as integer values for maximum portability. However, programmers are recommended to use named constants to identify messages rather than having integer values hardcoded into source programs, particularly in cases where a message is used from different places in the program, for example:

```
$ cat prog.h
#define      HELLO_MSG    1
...
#define TERM_MSG    42

$ cat prog.c
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>
#include "prog.h"

main()
{
    nl_catd catd;

    setlocale(LC_ALL, "");

    catd = catopen("prog", NL_CAT_LOCALE);
```

```
    puts(
       catgets(catd, NL_SETD, HELLO_MSG,
           "signing on - continue"));

    ...

    puts(
       catgets(catd, NL_SETD, TERM_MSG,
           "signing off - terminate"));

    exit(0);
}
```

The constant NL_SETD is defined in **<nl_types.h>** and gives the system's default set number.

### 3.6.2 Further Examples

The database application code presented in Section 3.2 on page 26 after internationalisation is as follows:

```
setlocale(LC_ALL, "");
catopen("prog", NL_CAT_LOCALE);

clear();

standout();
printw("\t\t      %s",
    catgets(catd, MSET, TITLE,
       "Inquire,  Update,  Add "));
printw("      \t\t");
standend();

printw("\n\n\n\n");
printw("\t\t\t    1 - %s\n",
    catgets(catd, MSET, ITEM1, "Add Record"));
printw("\t\t\t    2 - %s\n",
    catgets(catd, MSET, ITEM2, "Select Record"));
printw("\t\t\t    3 - %s\n",
    catgets(catd, MSET, ITEM3, "Search by Record"));

...

printw("\t\t\t    >   %s",
    catgets(catd, MSET, ITEM12, "Enter Selection "));
```

Note that all the string literals have been embodied in calls to *catgets*( ). This allows the English translation to be held with the program source, thus making it unnecessary to provide a message catalogue in English-language locales.

The default localisation does not have to be in English. However, it is worth bearing in mind that accented characters, for example, are not defined in the base character set and may not be supported on all implementations. Thus if any characters from the extended character set are used in string literals, the code may not be portable to all X/Open systems.

### 3.6.3 Performance

Another area that should be considered is program performance. Having to fetch character strings from a message catalogue is likely to be slower than having them defined as in-code literals. Programmers should weigh possible performance degradations that might result against the added usability that message localisation brings to an application.

In the preceding example, thirteen separate calls to *catgets*( ) are required to build the menu screen. This was measured and, although slower than using literals, was still found to be acceptable in performance terms. Had this not been the case, an alternative implementation could have been to load the message translations into a static array and use this in the above loop rather than going to the message catalogue every time. For example:

```
char *msgs [MAX_ITEM];

for(i = ITEM1; i <= ITEM12; ++i)
    if((s = catgets(catd, MSET, i, NULL)) != NULL) {
       if((msgs[i] = malloc(strlen(s)+1)) != NULL)
           strcpy(msgs[i], s);
    } else
       msgs[i] = NULL;
...
```

would require changes in the main loop of the program as follows:

```
...
printw("\n\n\n\n");
printw("\t\t\t    1 - %s\n",
       (msgs[ITEM1] ? msgs[ITEM1] : "Add Record"));
printw("\t\t\t    2 - %s\n",
       (msgs[ITEM2] ? msgs[ITEM2] : "Select Record"));
...
```

It is also worth remembering that if message catalogues are implemented using file descriptors, it is highly probable that some form of buffer caching is done by the operating system kernel. This may reduce the number of I/O operations required, depending on how the message catalogue is accessed. Also some implementations provide further caching within the message catalogue.

Chapter 4

# Internationalisation Support Interfaces

This chapter describes the functions that provide internationalisation support in XPG4 systems. They are grouped into:

- locale initialisation
- character classification
- case conversion
- character collation
- language information
- date and time
- printing
- scanning
- number conversion
- ISO C multi-byte functions
- I/O
- string handling
- error handling
- codeset conversion.

The ISO C standard includes a number of library functions that have been defined to operate in an international environment. The **XSH** specification includes the following functions from the ISO C standard:

| **ISO C Compatible Functions** | |
|---|---|
| *atof*( ) | *perror*( )† |
| *fprintf*( )† | *printf*( )† |
| *fscanf*( )† | *scanf*( )† |
| *isalnum*( ) | *setlocale*( ) |
| *isalpha*( ) | *sprintf*( )† |
| *iscntrl*( ) | *sscanf*( )† |
| *isdigit*( ) | *strcoll*( ) |
| *isgraph*( ) | *strerror*( ) |
| *islower*( ) | *strftime*( ) |
| *isprint*( ) | *strtod*( ) |
| *ispunct*( ) | *strxfrm*( ) |
| *isspace*( ) | *tolower*( ) |
| *isupper*( ) | *toupper*( ) |
| *isxdigit*( ) | *vfprintf*( )† |
| *localeconv*( ) | *vprintf*( )† |
| *mblen*( ) | *vsprintf*( )† |
| *mbstowcs*( ) | *wcstombs*( ) |
| *mbtowc*( ) | *wctomb*( ) |

Items marked with a dagger (†) in the above list have been enhanced by X/Open.

In addition, X/Open defines functions for compiling and matching internationalised regular expressions (see Chapter 5), native language message handling functions (see Chapter 3), codeset conversion functions (see Section 4.14 on page 83), functions for processing locale-dependent date and time strings (see Section 4.6 on page 62), a function for converting monetary values to strings (see Section 4.5 on page 56), and a function for reading the current setting of cultural data items (see Section 4.5 on page 56).

| **X/Open Additional Functions** | |
|---|---|
| *catclose*( ) | *regcomp*( ) |
| *catgets*( ) | *regerror*( ) |
| *catopen*( ) | *regexec*( ) |
| *iconv*( ) | *regfree*( ) |
| *iconv_close*( ) | *strfmon*( ) |
| *iconv_open*( ) | *strptime*( ) |
| *nl_langinfo*( ) | |

All other functions defined in the **XSH** specification provide at least single-byte transparency on all XPG4 systems.

The problem with many of the above interfaces, and other interfaces published in the **XSH** specification, is that they are defined to work with byte values, which limits them in terms of multi-byte codeset support. Ideally, these interfaces should be redefined to work with wide character types (see Section 1.5 on page 7). However, for reasons of compatibility with earlier systems, this approach is untenable.

X/Open's solution to this problem has been to define a parallel set of interfaces, known collectively as the **Worldwide Portability Interfaces** (WPI), functionally equivalent to the standard interfaces but which work on wide character values. The byte versions of these are retained in the interface definition for backward compatibility, but it is the WPI versions that are recommended for use by character-based portable applications.

The following such interfaces are defined in the **XPG4** base specifications:

| **Worldwide Portability Interfaces** | |
|---|---|
| *fgetwc*( ) | *wcscat*( ) |
| *fgetws*( ) | *wcschr*( ) |
| *fputwc*( ) | *wcscmp*( ) |
| *fputws*( ) | *wcscoll*( ) |
| *getwc*( ) | *wcscpy*( ) |
| *getwchar*( ) | *wcscspn*( ) |
| *iswctype*( ) | *wcsftime*( ) |
| *iswalnum*( ) | *wcslen*( ) |
| *iswalpha*( ) | *wcsncat*( ) |
| *iswcntrl*( ) | *wcsncmp*( ) |
| *iswdigit*( ) | *wcsncpy*( ) |
| *iswgraph*( ) | *wcspbrk*( ) |
| *iswlower*( ) | *wcsrchr*( ) |
| *iswprint*( ) | *wcsspn*( ) |
| *iswpunct*( ) | *wcstod*( ) |
| *iswspace*( ) | *wcstok*( ) |
| *iswupper*( ) | *wcstol*( ) |
| *iswxdigit*( ) | *wcstoul*( ) |
| *putwc*( ) | *wcswcs*( ) |
| *putwchar*( ) | *wcswidth*( ) |
| *towlower*( ) | *wcsxfrm*( ) |
| *towupper*( ) | *wctype*( ) |
| *ungetwc*( ) | *wcwidth*( ) |

These functions are described in detail in the following sections. For a complete description of all interfaces, see the **XSH** specification and the **Curses** specification.

## 4.1 Locale Initialisation

Localisation data is bound to the locale of an internationalised application by calling the *setlocale*( ) function. This should be the first action performed by such an application and should be called prior to using any of the internationalisation functions listed above.

### 4.1.1 The setlocale( ) Function

This function is defined in the header **<locale.h>**.

```
char *setlocale(int category, const char *locale);
```

The *setlocale*( ) function selects the appropriate piece of the program's locale, as defined by the setting of *category* and *locale*, and can be used to change or query the program's international environment. The value LC_ALL for *category* names the entire locale. Other values name a specific part of the locale, as follows:

LC_COLLATE refers to collating sequence information and affects the behaviour of internationalised regular expressions and the string collation functions described in Section 4.4 on page 52.

LC_CTYPE refers to character type information and affects the behaviour of internationalised regular expressions, the character classification functions described in Section 4.2 on page 48, the character conversion functions described in Section 4.3 on page 51, the wide character conversion functions described in Section 4.9 on page 72 and Section 4.10 on page 74.

LC_MESSAGES refers to message translations, including yes/no responses required by certain commands and utilities defined in the **XCU** specification.

LC_MONETARY refers to monetary formatting information and affects the behaviour of the *strfmon*( ) and *localeconv*( ) functions described in Section 4.5 on page 56.

LC_NUMERIC refers to number formatting information and affects the behaviour of the *localeconv*( ) function, the printing functions described in Section 4.7 on page 68 and the scanning functions described in Section 4.8 on page 70.

LC_TIME refers to date and time information and affects the *strftime*( ), *wcsftime*( ) and *strptime*( ) functions described in Section 4.6 on page 62.

The *nl_langinfo*( ) function can also be used to obtain certain information in the program's locale; see Section 4.5 on page 56.

The *locale* argument is a pointer to a character string containing the required setting of *category*, in the form of a locale name. The actual settings of this argument are implementation-defined, with the exception of the following which are supported on all XPG4 systems:

"POSIX" specifies the minimal environment for C language translation. If *setlocale*( ) is not called by an application, the POSIX locale is applied as a default.

"C" same as "POSIX".

"" specifies that the locale should be initialised from the corresponding environment variables.

NULL this setting is used to direct *setlocale*( ) to query the current locale setting and return its name to the caller.

If successful, *setlocale*( ) returns a pointer to a locale-name string; otherwise it returns a null pointer. A null pointer for *locale* causes *setlocale*( ) to return a pointer to the locale-name string associated with category.

The string returned by *setlocale*( ) is such that a subsequent call with that string and its associated *category* restores that part of the program's locale. Note that this value may be overwritten by subsequent calls to *setlocale*( ) and should be saved by a program if it is to be reused.

**Note:** The format of the return string from *setlocale*( ) is non-standard across vendors' platforms and should not be parsed by applications.

### 4.1.2 Setting the Program Locale

Basically there are two ways to set the program locale using the *setlocale*( ) function:

`setlocale(`*`category`*`, `*`string`*`)`

This usage sets a specific *category* in the program locale to the value of *string*, as follows:

```
setlocale(LC_ALL, "fr_FR.8859");
```

This example sets all categories of the program locale to localisation data associated with the specified locale name (fr_FR.8859). Remember, however, that this is only an example and values of locale name are implementation-defined.

If *string* does not contain a valid locale name, *setlocale*( ) returns a null pointer and the program locale is not changed. Otherwise, it returns the name of the locale.

This special case sets the minimal environment for C translation and is the one locale guaranteed to exist on all X/Open systems:

```
setlocale(category, "C")
```

`setlocale(`*`category`*`, "")`

This usage sets *category* to the setting of the associated international environment variables. This can be further subdivided as follows:

- setting a specific category from the environment
- setting all categories from the environment

The behaviour of these is described in the following sections.

### 4.1.3 Setting a Specific Category from the Environment

This allows one of LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, or LC_TIME to be set individually, for example:

```
setlocale(LC_COLLATE, "");
```

In this case, *setlocale*( ) takes the name of the new locale for the specified category from the environment, as determined by the first condition met below:

i. *$LC_ALL* is defined in the environment and is not null.

ii. A variable is defined in the environment with the same name as the category (for example, *$LC_COLLATE*) and is not null.

iii. *$LANG* is defined in the environment and is not null.

If the resulting value is a supported locale, *setlocale*( ) sets the supplied category to that value and returns its name. If the value does not name a supported locale (and is not null), *setlocale*( ) returns a null pointer and the program locale is not changed. Otherwise, the behaviour of *setlocale*( ) is implementation-defined.

### 4.1.4 Setting all Categories from the Environment

This is similar to the previous usage, except that when *$LC_ALL* is not defined or is null, *setlocale*( ) examines all the environment variables with the same name as the categories to determine which values to set in the program locale. In this case, *setlocale*( ) is called as follows:

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

To satisfy this request, *setlocale*( ) first checks all the environment variables. If the setting of any variable is invalid, *setlocale*( ) returns a null pointer and the program locale is not changed. If they are valid, *setlocale*( ) proceeds as if it had been called for each category, using the value from the associated environment variable, or from an implementation-defined default if there is no such value.

### 4.1.5 The POSIX Locale

As already mentioned, the POSIX locale is the one locale guaranteed to exist on all X/Open systems. This is useful for verification purposes or if an application needs to establish operation in a known environment.

#### Coded Character Set

Although implementations are free to use any coded character set in the POSIX locale, all characters defined in the Portable Character Set and the Control Code Character Set must be supported. These characters are collated in the following order (reading from left to right and top to bottom):

```
<NUL>     <SOH>     <STX>     <ETX>
<EOT>     <ENQ>     <ACK>
```

```
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>

<SO>     <SI>     <DLE>    <DC1>
<DC2>    <DC3>    <DC4>    <NAK>
<SYN>    <ETB>    <CAN>    <EM>
<SUB>    <ESC>    <IS4>    <IS3>
<IS2>    <IS1>

<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen>
<period>
<slash>

<zero>   <one>    <two>    <three>
<four>   <five>   <six>    <seven>
<eight>  <nine>

<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>

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

```
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underline>
<grave-accent>

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

<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
<DEL>
```

No equivalence classes, N-to-1, 1-to-2 or don’t-care character mappings are defined in this locale.

**Cultural Data**

The following items of cultural data are defined in the POSIX locale:

```
LC_TIME
abday          "Sun";"Mon";"Tue";"Wed";"Thu";"Fri";"Sat"
day            "Sunday";"Monday";"Tuesday";"Wednesday";"Thursday";\
               "Friday";"Saturday"
abmon          "Jan";"Feb";"Mar";"Apr";"May";"Jun";"Jul";"Aug";"Sep";\
               "Oct";"Nov";"Dec"
mon            "January";"February";"March";"April";"May";"June";"July";\
               "August";"September ";"October";"November";"December"
d_t_fmt        "%a %b %d %H:%M:%S %Y"
d_fmt          "%m/%d/%y"
t_fmt          "%H:%M:%S"
am_pm          "AM";"PM"
t_fmt_ampm     "%I:%M:%S %p"
END LC_TIME

LC_MONETARY
int_curr_symbol          ""
currency_symbol          ""
mon_decimal_point        ""
mon_thousands_sep        ""
mon_grouping             -1
positive_sign            ""
negative_sign            ""
int_frac_digits          -1
frac_digits              -1
p_cs_precedes            -1
p_sep_by_space           -1
n_cs_precedes            -1
n_sep_by_space           -1
p_sign_posn              -1
n_sign_posn              -1
LC_MONETARY

LC_NUMERIC
decimal_point    "<period>"
thousands_sep    ""
grouping          -1
END LC_NUMERIC

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

Note that date and time settings are defined in terms of *strftime*( ) format strings.

**Language**

No specific language is defined, meaning that program messages are produced in the original language of the program.

## 4.2 Character Classification

Character classification is provided by an internationalised version of the character classification functions defined in **<ctype.h>**. These classify wide character values according to the rules of the coded character set defined by **type** information in the program's locale (category LC_CTYPE). Each function returns non-zero for true, zero for false.

It is important to appreciate what is meant by locale-specific character classification. For example, in an English language locale supporting the ASCII codeset, only the characters a-z and A-Z would be classified as alphabetic. In a Japanese locale, alphabetics may also include multi-byte characters.

These functions are defined in the header **<wchar.h>**.

### 4.2.1 The iswalnum( ) Function

```
int    iswalnum(wint_t wc);
```

This function tests whether *wc* is an alphanumeric wide-character code. It returns non-zero if *wc* is an alphanumeric wide-character code, that is if *iswalpha*( ) or *iswdigit*( ) would return non-zero. Otherwise it returns non-zero.

### 4.2.2 The iswalpha( ) Function

```
int    iswalpha(wint_t wc);
```

This function tests whether *wc* is an alphabetic wide-character code. It returns non-zero if *wc* is an alphabetic wide-character code. The characters that are classified as either upper-case (that is *iswupper*( ) returns non-zero) or lower-case (that is *iswlower*( ) returns non-zero) are automatically classified as alphabetic (*iswalpha*( ) returns non-zero). It is worth noting that there are characters which should be classified as alphabetic but have no difference between upper- and lower-case. Otherwise it returns zero.

### 4.2.3 The iswcntrl( ) Function

```
int    iswcntrl(wint_t wc);
```

The *iswcntrl*( ) function tests whether *wc* is a control wide-character code. It returns non-zero if *wc* is a control wide-character code (for example, character codes 0-31 and 127 in a locale that supports the ASCII codeset). Otherwise it returns zero.

### 4.2.4 The iswdigit( ) Function

```
int    iswdigit(wint_t wc);
```

The *iswdigit*( ) function tests whether *wc* is a decimal digit wide-character code in the portable character set; that is, one of the characters

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

The *iswdigit*( ) function returns non-zero if *wc* is a decimal digit wide-character code. Otherwise it returns zero.

### 4.2.5 The iswgraph( ) Function

```
int    iswgraph(wint_t wc);
```

This function tests whether *wc* is a graphic wide-character code. It returns non-zero if *wc* is a wide-character code with a visible representation. Otherwise it returns zero.

### 4.2.6 The iswlower( ) Function

```
int    iswlower(wint_t wc);
```

The *iswlower*( ) function tests whether *wc* is a lower-case letter wide-character code (for example, character codes 97-122 in a locale that supports the ASCII codeset). It returns non-zero if *wc* is a lower-case letter wide-character code. Otherwise it returns zero.

### 4.2.7 The iswprint( ) Function

```
int    iswprint(wint_t wc);
```

This function tests whether *wc* is a printing wide-character code. It returns non-zero if *wc* is a printing wide-character code. Otherwise it returns zero.

The difference between this function and *iswgraph*( ) is that *iswprint*( ) also returns non-zero if *wc* is a space wide-character code. The *iswgraph*( ) function would return zero in this case.

### 4.2.8 The iswpunct( ) Function

```
int    iswpunct(wint_t wc);
```

This function tests whether *wc* is a punctuation wide-character code (for example, character codes 33-47, 58-64, 91-96 and 123-126 in a locale that supports the ASCII codeset). It returns non-zero if *wc* is a punctuation wide-character code. Otherwise it returns zero.

### 4.2.9 The iswspace( ) Function

```
int    iswspace(wint_t wc);
```

This function tests if *wc* is a wide-character code causing white space in displayed text (for example, space, tab, carriage return, newline, vertical tab or form feed in a locale that supports the ASCII codeset). It returns non-zero if *wc* is a white-space wide-character code. Otherwise it returns zero.

### 4.2.10 The iswupper( ) Function

```
int    iswupper(wint_t wc);
```

This function tests whether *wc* is an upper-case wide-character code (for example, character codes 65-90 in a locale that supports the ASCII codeset). It returns non-zero if *wc* is an upper-case wide-character code. Otherwise it returns zero.

### 4.2.11 The iswxdigit() Function

```
int    iswxdigit(wint_t wc);
```

The *iswxdigit*( ) function tests whether *wc* is a hexadecimal digit wide-character code in the portable character set; that is, one of the characters:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

or

```
a, b, c, d, e, f
```

or

```
A, B, C, D, E, F
```

The *iswxdigit*( ) function returns non-zero if *wc* is a hexadecimal digit wide-character code. Otherwise it returns zero.

### 4.2.12 The iswctype() Function

```
int    iswctype(wint_t wc, wctype_t charclass);
```

This function determines whether the wide-character *wc* has the property *charclass*, returning zero (false) or non-zero (true). Thus in the table below, the functions in the left column are equivalent to the functions in the right column.

```
iswalnum(wc)  iswctype(wc, wctype("alnum"))
iswalpha(wc)  iswctype(wc, wctype("alpha"))
iswcntrl(wc)  iswctype(wc, wctype("cntrl"))
iswdigit(wc)  iswctype(wc, wctype("digit"))
iswgraph(wc)  iswctype(wc, wctype("graph"))
iswlower(wc)  iswctype(wc, wctype("lower"))
iswprint(wc)  iswctype(wc, wctype("print"))
iswpunct(wc)  iswctype(wc, wctype("punct"))
iswspace(wc)  iswctype(wc, wctype("space"))
iswupper(wc)  iswctype(wc, wctype("upper"))
iswxdigit(wc) iswctype(wc, wctype("xdigit"))
```

**Note:** The "blank" property is defined in XPG4, but there is no *iswblank*( ) function. Thus, *iswctype*( ) has to be used to test for the "blank" property.

### 4.2.13 The wctype() Function

```
wctype_t      wctype(const char *charclass);
```

This function returns a value that can be passed as the second argument in a call to *iswctype*( ). The values are defined in the LC_CTYPE category. It is a generic function, where *charclass* is a string identifying the character class for which locale-specific type information is required.

## 4.3 Case Conversion

Case conversion facilities are provided by internationalised versions of the case conversion functions defined in **<ctype.h>**. These provide case conversion according to shift information defined in the program's locale (category LC_CTYPE).

Note that character classes **upper** and **lower** are specific to certain alphabets and general use of these in internationalised software is discouraged. When they are used, programs continue to work in all locales but case conversion may be a null function.

All the functions described in this section, the constant WEOF and the data type **wint_t** are defined in the header **<wchar.h>**.

### 4.3.1 The towlower() Function

```
wint_t towlower(wint_t wc);
```

This function takes a **wint_t** data item as its argument, the value of which is representable as a **wchar_t**, and must be a wide-character code corresponding to a valid character in the current locale or the constant WEOF. If there is a lower-case wide-character code mapping for *wc* in the program's locale, *towlower*( ) returns the mapped value. Otherwise it returns the value unchanged.

### 4.3.2 The towupper() Function

```
wint_t towupper(wint_t wc);
```

This function takes a **wint_t** data item as its argument, the value of which is representable as a **wchar_t**, and must be a wide-character code corresponding to a valid character in the current locale or the constant WEOF. If there is an upper-case wide-character code mapping for *wc* in the program's locale, *towupper*( ) returns the mapped value. Otherwise it returns the value unchanged.

## 4.4 Character Collation

Two functions are provided for comparing wide-character strings using **collating sequence** information in the program's locale (category LC_COLLATE). The *wcscoll*( ) function obeys the LC_COLLATE category, whereas *wcscmp*( ) does not. The function *wcsxfrm*( ) allows wide-character strings to be transformed such that they can be compared using the *wcscmp*( ) function.

### 4.4.1 The wcscoll( ) Function

This function is declared in the header **<wchar.h>**.

```
int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

This function compares the wide-character strings pointed to by *ws1* and *ws2* using collating sequence information in the current locale. The *wcscoll*( ) function returns a value greater than 0 if *ws1* collates after *ws2*, zero if the two wide-character strings are equal, or a value less than 0 if *ws1* collates before *ws2*.

On error, *wcscoll*( ) sets *errno* but no return value is reserved to indicate the error condition. In general an error can only arise if the string pointed to by *ws1* or *ws2* contains an invalid wide character. A valid character outside the domain of the collating sequence has no effect on comparisons and is not an error. Applications wishing to check for errors should set *errno* to zero before calling *wcscoll*( ) and check its value on return.

As an example of this interface, the following is a simple sort program that uses a combination of *qsort*( ) and *wcscoll*( ) to order the records of a text file:

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

#define ELEMENTS   512
#define WIDTH      256
#define USE        1
#define FAIL       2
#define SIZE       3

wchar_t     table [ELEMENTS] [WIDTH];
```

```
main(argc, argv)
int   argc;
char  **argv;
{
     FILE    *fp;
     nl_catd catd;
     int     nel, i, wcscoll( );

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

     if(argc != 2) {
         fprintf(stderr,
              catgets(catd, NL_SETD, USE,
               "usage: qsort file\n"));
         exit(1);
     }
     if((fp = fopen(argv [1], "r")) == NULL) {
         fprintf(stderr,
              catgets(catd, NL_SETD, FAIL,
               "can't open %s\n", argv [1]));
         exit(2);
     }

     for(nel = 0; nel < ELEMENTS &&
         fgetws(table [nel], WIDTH, fp); ++nel) ;

     fclose(fp);

     if(nel >= ELEMENTS) {
         fprintf(stderr,
              catgets(catd, NL_SETD, SIZE,
                  "file too large\n"));
         exit(3);
     }
     qsort(table, nel, WIDTH, wcscoll);

     for(i = 0; i < nel; ++i)
         fputws(table[i], stdout);

     exit(0);
}
```

An advantage of *wcscoll*( ) is that the wide-character strings are not modified in any way, and hence remain printable. A disadvantage is that because strings must be transformed each time *wcscoll*( ) is called, performance overheads may be significant.

### 4.4.2 The wcsxfrm( ) Function

This function is defined in the header **<wchar.h>**.

```
size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

The *wcsxfrm*( ) function transforms the wide-character string pointed to by *ws2* and places the result in the array pointed to be *ws1*. The transformation is such that if the *wcscmp*( ) function is applied to two transformed strings, the result is the same as if the strings had been compared using *wcscoll*( ).

No more than *n* elements are placed in the array pointed to by *ws1*, including a terminating null wide character. If *n* is zero, *ws1* can be a null pointer. This form of a call is useful to determine the size of the *ws1* array prior to making the transformation.

The *wcsxfrm*( ) function returns the number of elements in the transformed string, not including the terminating null wide character. If the return value is equal to or greater than *n*, the contents of the array pointed to by *ws1* are indeterminate.

If unsuccessful, *wcsxfrm*( ) returns (**size_t**)–1 and sets *errno* to indicate the error. This can only arise if the string pointed to by *ws2* contains an invalid wide character. A valid character outside the domain of the collating sequence should be ignored and is not an error.

Points worth noting about this function include:

- Where wide-character strings are likely to be compared more than once, it is more efficient to transform them via *wcsxfrm*( ) and then compare them using *wcscmp*( ).
- When using *wcscmp*( ) to compare transformed wide character strings, both *ws1* and *ws2* must have been produced by the *wcsxfrm*( ) function; that is, a transformed string cannot be compared with a non-transformed string.
- No mechanism is defined to return a transformed string to its original state. Thus, assuming sorted output is required in text form, it is necessary for a program using *wcsxfrm*( ) to retain some form of association between a transformed string and its textual representation.

Using the previous example, instead of identifying *wcscoll*( ) as the comparison function in the call to *qsort*( ), we could supply our own function that uses *wcsxfrm*( ), as follows:

```
compare(ws1, ws2)
wchar_t *ws1, *ws2;
{
       wchar_t      *ts1, *ts2;
       int    result;
       size_t n1, n2;

     n1 = wcsxfrm(NULL, ws1, 0);
     if(n1 ==(size_t)-1)
        return(wcscmp(ws1, ws2));

     n2 = wcsxfrm(NULL, ws2, 0);
     if (n2 == (size_t)-1))
         return(wcscmp(ws1, ws2));
```

```
        if ((ts1 = (wchar_t*)
           calloc(n1+1, sizeof(wchar_t)))
               == NULL)
           return(wcscmp(ws1, ws2));

        if ((ts2 = (wchar_t*)
           calloc(n2+1, sizeof(wchar_t)))
               == NULL) {
           free(ts1);
           return(wcscmp(ws1, ws2));
        }

        wcsxfrm(ts1, ws1, n1+1);
        wcsxfrm(ts2, ws2, n2+1);
        result = wcscmp(ts1, ts2);

        free(ts1);
        free(ts2);

        return(result);
  }
```

Note that in the above example *wcsxfrm*( ) is used to determine how much storage space is required to hold the transformed wide-character strings. This is done by passing a null pointer as the the first argument in a call to *wcsxfrm*( ). The return value plus 1 gives the amount of space required in units of type **wchar_t**. This value is then passed as the first argument to a call of *calloc*( ).

The above code demonstrates how the *wcsxfrm*( ) function can be used but is inefficient in that the strings are transformed each time the function is called. A better and more competent implementation would be to retain the transformed strings for subsequent comparisons.

## 4.5 Language Information

Various functions are provided that allow a program to access items of cultural data in the program's locale. These should be used to determine the prevailing setting and format of cultural data, rather than having this information built into the program. The *nl_langinfo*( ) function provides language information, *strfmon*( ) converts monetary value to a string and *localeconv*( ) determines the program's locale.

### 4.5.1 The nl_langinfo() Function

This function, the type **nl_item** and constants used for the setting of *item* are defined in the header **<langinfo.h>**.

```
char   *nl_langinfo(nl_item item);
```

The *nl_langinfo*( ) function returns a pointer to a string containing information relevant to the particular language or cultural area currently defined in the program's locale. The required item of language information is identified by the value of *item*, which is a value of type **nl_item**. Permitted settings of this argument are defined by a set of constants in **<langinfo.h>**. At least the following are supported on all XPG4 systems:

| Constant | Category | Meaning |
|---|---|---|
| CODESET | LC_CTYPE | codeset name |
| D_T_FMT | LC_TIME | string for formatting date and time |
| D_FMT | LC_TIME | date format string |
| T_FMT | LC_TIME | time format string |
| T_FMT_AMPM | LC_TIME | a.m. or p.m. time format string |
| AM_STR | LC_TIME | ante meridiem affix |
| PM_STR | LC_TIME | post meridiem affix |
| DAY_1 | LC_TIME | name of the first day of the week (for example, Sunday) |
| ... | | |
| DAY_7 | LC_TIME | name of the seventh day of the week |
| ABDAY_1 | LC_TIME | abbreviated name of the first day of the week |
| ... | | |
| ABDAY_7 | LC_TIME | abbreviated name of the seventh day of the week |
| MON_1 | LC_TIME | name of the first month in the year |
| ... | | |
| MON_12 | LC_TIME | name of the twelfth month |
| ABMON_1 | LC_TIME | abbreviated name of the first month |
| ... | | |
| ABMON_12 | LC_TIME | abbreviated name of the twelfth month |
| ERA | LC_TIME | era display format |
| ERA_D_FMT | LC_TIME | era date format |
| ERA_D_T_FMT | LC_TIME | era date and time format string |
| ERA_T_FMT | LC_TIME | era time format string |
| ALT_DIGITS | LC_TIME | alternative digit symbols |

| Constant | Category | Meaning |
|---|---|---|
| RADIXCHAR | LC_NUMERIC | radix character |
| THOUSEP | LC_NUMERIC | separator for thousands |
| CODESET | LC_CTYPE | implementation-defined (note that this is missing from the **XSH** specification) |
| YESSTR | LC_MESSAGES | affirmative response for yes/no queries (**TO BE WITHDRAWN**) |
| NOSTR | LC_MESSAGES | negative response for yes/no queries (**TO BE WITHDRAWN**) |
| YESEXPR | LC_MESSAGES | affirmative response for yes/no queries |
| NOEXPR | LC_MESSAGES | negative response for yes/no queries |
| CRNCYSTR | LC_MONETARY | currency symbol, preceded by – if the symbol should appear before the value, + if the symbol should appear after the value, or . if the symbol should replace the radix character. |

The value of CODESET is implementation-defined.

In a locale where language information data is not defined, *nl_langinfo*( ) returns a pointer to the corresponding item in the POSIX locale. In all locales, *nl_langinfo*( ) returns a pointer to an empty string if *item* contains an invalid setting. A pointer to an empty string is returned, rather than a null pointer, so that failure conditions set by *nl_langinfo*( ) do not invalidate argument lists.

The array pointed to by the return value should not be modified by the program and may be overwritten by subsequent calls to *nl_langinfo*( ). This value should be copied to another area if it is to be reused.

As an example, the following program outputs day names defined in the current locale:

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

nl_item items[7] = {
     DAY_1, DAY_2, DAY_3, DAY_4,
     DAY_5, DAY_6, DAY_7};

main( )
{
     int i;

     setlocale(LC_ALL, "");

     for(i = 0; i < 7; ++i)
         puts(nl_langinfo(items [i]));

     exit(0);
}
```

### 4.5.2 The strfmon( ) Function

This function is declared in the header **<monetary.h>**.

```
ssize_t strfmon(char *s, size_t maxsize, const char *format, ...)
```

The *strfmon*( ) function places characters into the character array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed in the array.

The *format* string consists of zero or more conversion specifications and plain characters. A conversion specification consists of a **%** character, optional flags, an optional field width and precision, and a conversion character that determines the conversion specifications' behaviour. All plain characters, including the terminating null character, are copied to *s* unchanged.

If successful, the *strfmon*( ) function returns the number of bytes placed into the array pointed to by *s*. Otherwise, –1 is returned and the contents of the array are indeterminate.

#### Conversion Characters

The following conversion characters are defined in the **XSH** specification:

%n the argument of type **double** is formatted according to the locale's national currency format.

%i the argument of type **double** is formatted according to the locale's international currency format.

%% is replaced by a single **%**.

No arguments are converted for %% conversion specifications.

**Field Width and Precision**

Optional field width and precision specifications can immediately follow the % character introducing a conversion. These are specified as follows:

*=f* the fill character for *#n* conversions (space by default).

^ indicates that the output value should not be formatted using the thousands grouping symbol. By default the output is formatted using the thousands grouping symbol defined in the current locale (if defined).

+ or ( indicates the presentation style for positive and negative currency values. If + is specified, the locale's equivalent of + and – are used. If ( is specified, negative values are enclosed in parenthesis. If neither flag is specified, + is used.

! suppresses the currency symbol from being output.

– where – specifies the alignment. If this flag is present all fields are left aligned (padded to the right) rather than the default of right aligned.

*w* where *w* is a decimal digit string specifying the minimum field width.

*.p* where *p* is a decimal digit string specifying the number of digits after the radix (decimal-point) character. If *p* is zero, no radix character appears in the output. If not specified, a default value is taken from the current locale. Rounding to the specified number of digits occurs before the number is formatted.

*#n* where *n* is a decimal digit string specifying a maximum number of digits output to the left of the radix character. This option allows output to be aligned. If the number of output digits is greater than *n*, the option is ignored; if it is less, the number is padded on the left with fill characters (see *=f*).

**Examples**

```
double val;
char   buf[MAXSIZE];
....
setlocale(LC_ALL, "");
....
if(strfmon(buf, MAXSIZE, catgets(catd, setn, msgw, "%#5=*n"),
           val) != -1)
       printf(
              catgets(catd, setn, msgx,
                     "amount = %s\n"),
              buf);
else
       printf(
              catgets(catd, setn, msgy,
                     "value too large \n"))
....
```

which formats a monetary value with five digits to the left of the radix character, filled with asterisks if necessary, for example:

```
$**123.45
```

### 4.5.3 The localeconv() Function

This function is defined in the header **<locale.h>**.

```
struct lconv *localeconv(void);
```

The *localeconv*( ) function sets the components of an object with the type **struct lconv** with the values appropriate for the formatting of numeric quantities (monetary or otherwise) according to the rules of the current locale.

The **lconv** structure contains at least the following members (comments indicate values in the POSIX locale):

```
char   *decimal_point;       /*"."*/
char   *thousands_sep;       /*""*/
char   *grouping;            /*""*/
char   *int_curr_symbol;     /*""*/
char   *currency_symbol;     /*""*/
char   *mon_decimal_point;   /*""*/
char   *mon_thousands_sep;   /*""*/
char   *mon_grouping;        /*""*/
char   *positive_sign;       /*""*/
char   *negative_sign;       /*""*/
char    int_frac_digits;     /*CHAR_MAX*/
char    frac_digits;         /*CHAR_MAX*/
char    p_cs_precedes;       /*CHAR_MAX*/
char    p_sep_by_space;      /*CHAR_MAX*/
char    n_cs_precedes;       /*CHAR_MAX*/
char    n_sep_by_space;      /*CHAR_MAX*/
char    p_sign_posn;         /*CHAR_MAX*/
char    n_sign_posn;         /*CHAR_MAX*/
```

Structure members of type **char*** are pointers to strings, any of which (except **decimal_point**) can point to "", indicating that the associated value is not available in the current locale or is of zero length. Structure members of type **char** are non-negative numbers, any of which can be {CHAR_MAX} (defined in **<limits.h>**), indicating that the associated value is not available in the current locale.

The element **grouping** defines the size of each group of digits in formatted non-monetary quantities.

**Note:** The ISO C string pointed to by the element **grouping** is not the same as the definition of the **grouping** keyword in the locale definition file (see the **XBD** specification).

The element **grouping** points to a string containing a sequence of integers specifying the number of digits in each group; The initial integer defines the size of the group immediately preceding the decimal delimiter; the following integers define the preceding groups. Similarly **mon_grouping** defines the size of each group of digits in formatted monetary quantities. The strings pointed to by *grouping* and *mon_grouping* are interpreted as follows:

{CHAR_MAX} No further grouping is performed.

0 The previous element is to be used repeatedly for the remainder of the digits.

**Note:** This option is valid but redundant because exactly the same result is obtained when the 0 is omitted — in that case the string matches the next option.

*other* The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

For example, if the octal value of {CHAR_MAX} is 177 and a comma is the thousands separator, the following strings would produce formatting as shown:

| ISO C String | Formatted Value |
|---|---|
| "\3\177" | 123456,789 |
| "\3" | 123,456,789 |
| "\3\2\177" | 1234,56,789 |
| "\3\2" | 12,34,56,789 |
| "\177" | 123456789 |

The values of *p_sign_posn* and *n_sign_posn* are interpreted as follows:

0 Parenthesis surround the quantity and currency symbol or int_curr_symbol.

1 The sign string precedes (occurs before with or without a space) the quantity and currency symbol or int_curr_symbol.

2 The sign string succeeds (occurs after with or without a space) the quantity and currency symbol or int_curr_symbol.

3 The sign string immediately (occurs after with no space) precedes the currency symbol or int_curr_symbol.

4 The sign string immediately (occurs before with no space) succeeds the currency symbol or int_curr_symbol.

The *localeconv*( ) function returns a pointer to the completed structure. Note that the structure pointed to by the return value should not be modified by the program. It may be overwritten by a subsequent call to the *localeconv*( ) function, or by calls to the *setlocale*( ) function specifying categories LC_ALL, LC_MONETARY or LC_NUMERIC.

This interface provides all the information necessary for the formatting of numeric quantities. However, it is the responsibility of the application to interpret that information and format numeric values accordingly.

## 4.6 Date and Time Functions

In the past, applications have used interfaces like *ctime*( ) and *asctime*( ) to format date and time strings. These produced printable representations of date and time in the following form:

```
Sun Sep 16 01:03:52 1973\n\0
```

The *ctime*( ) function converted a long integer representing the time in seconds since the Epoch, while *asctime*( ) accepted a **tm** structure as input. The problem with both these interfaces was:

- they had built-in knowledge of language and cultural data formats
- they were too rigid in their definition and provided no facilities for user formatting of date and time information

Both these functions are retained in the interface definition for compatibility with earlier systems. New interfaces such as *strftime*( ), *wcsftime*( ) and *strptime*( ) are also specified, which provide conversion facilities based on language information defined in the program's locale. These functions also allow individual fields of a date and time string to be handled separately.

### 4.6.1 The strftime( ) Function

This function is defined in the header **<time.h>**.

```
size_t strftime(char *s, size_t maxsize, const char *format,
       const struct tm *timptr);
```

The *strftime*( ) function places bytes into the character array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed in the array.

The *format* string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a % character and a terminating conversion character or characters that determine the conversion specification's behaviour. All ordinary characters, including the terminating null character, are copied unchanged to the array.

Local timezone information is used as though *strftime*( ) called the *tzset*( ) function.

If the total number of resulting bytes, including the terminating null byte, is not greater than *maxsize*, the *strftime*( ) function returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise zero is returned and the contents of the array are indeterminate.

#### Conversions

The full list of conversion specifications is given in the **XSH** specification, but at least the following are supported on all conforming implementations:

%a is replaced by the locale's abbreviated weekday name.
%A is replaced by the locale's full weekday name.
%b is replaced by the locale's abbreviated month name.
%B is replaced by the locale's full month name.
%c is replaced by the locale's appropriate date and time representation.
%C† is replaced by the century number (the year divided by 100 and truncated to an integer as a decimal number [00-99]).
%d is replaced by the day of the month as a decimal number [01,31]
%D† is replaced by the date (%m / %d / %y).
%e† is replaced by the day of the month as a decimal number [1,31] in a 2-digit field with leading space character fill.
%h† same as %b.

%H is replaced by the hour (24-hour clock) as a decimal number [00,23].
%I is replaced by the hour (12-hour clock) as a decimal number [01,12].
%j is replaced by the day of the year as a decimal number [001,366]
%m is replaced by the month as a decimal number [01,12].
%M is replaced by the minute as a decimal number [00,59].
%n† is replaced by a newline character.
%p is replaced by the locale's equivalent of either AM or PM.
%r† is replaced by the time in a.m. and p.m. notation; in the POSIX locale this is equivalent to %I:%M:%S %p.
%R† is replaced by the time in 24-hour notation (%H:%M).
%S is replaced by the second as a decimal number [00,61].
%t† is replaced by a tab character.
%T† is replaced by the time (%H:%M:%S).
%u is replaced by the weekday as a decimal number [1,7], with 1 representing Monday.
%U is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number [00,53].
%V is replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1; otherwise it is week 53 of the previous year, and the next week is week 1.
%w is replaced by the weekday (Sunday as the first day of the week) as a decimal number [0,6].
%W is replaced by the week number of the year (Monday as the first day of the week) as a decimal number [00,53].
%x is replaced by the locale's appropriate date representation.
%X is replaced by the locale's appropriate time representation.
%y is replaced by the year without century as a decimal number [00,99].
%Y is replaced by the year with century as a decimal number.
%Z is replaced by the timezone name, or by no characters if no timezone exists.
%% is replaced by %.

Items marked with a dagger (†) in the preceding list are X/Open extensions to the interface definition included primarily for compatibility with previous issues of the Portability Guide.

Some conversion specifiers can be modified by the E and O modifier characters (see the **XSH** specification) to indicate a different format or specification. If this information is not supported in the current locale, the unmodified conversion specifier is used.

If a conversion character is not one of the above, the behaviour of *strftime*( ) is undefined.

The range of values for %S is [00,61] rather than [00,59] to allow for the occasional leap second, and the even more occasional double leap second.

**Examples**

This example first initialises the curses package, clears the standard window, and moves the cursor to the bottom left-hand corner of the screen. It then calls *strftime*( ) to produce a date and time string of the form:

```
Wednesday, January 24, 1990 (09:50:59)
```

If this call fails (for example, because *buf* is not large enough), *asctime*( ) is used instead to produce a non-internationalised date and time string.

```
...
time_t tval;
struct tm tmptr;
char   buf [bufsize];

setlocale(LC_ALL, "");

initscr( );
clear( );
move(LINES - 1, 0);

tval = time((time_t*) NULL);
tmptr = localtime(tval);

if (strftime(buf, bufsize,
       "%A, %B %d, %Y (%T)", tmptr))
              printw("%s", buf);
else
       printw("%s", asctime(timptr));
...
```

The next example uses *strftime*() to produce output similar to that generated by the *date* command:

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

#define DEFAULT_TIME "%a %b %d %H:%M:%S %Z %Y"
#define            PROG_NAME    "xdate"
#define            ERROR_MSG    1
#define FMT_MSG      2

char tbuf [BUFSIZ];

main   (argc, argv)
int argc;
char   **argv;
{
   time_t  clock = time((time_t*) NULL);
   struct  tm *tmptr = localtime(&clock);
   nl_catd catd;

   setlocale(LC_ALL, "");
   catd = catopen(PROG_NAME, NL_CAT_LOCALE);

   if (argc == 1) {

      strftime(tbuf, BUFSIZ,
               catgets(catd, NL_SETD, FMT_MSG, DEFAULT_TIME, tmptr);
      puts(tbuf);

   } else {
```

```
        for (--argc, ++argv;
            argc;
            --argc, ++argv) {

            if (*argv[0] == '+') {
                strftime(tbuf, BUFSIZ,
                    argv[0]+1, tmptr);
                puts(tbuf);
            } else {
                printf( catgets(catd,
                    NL_SETD, ERROR_MSG,
                    "%s: unknown argument\n"),
                    argv[0]);
                exit(1);
            }
        }
    }

    exit(0);
}
```

### 4.6.2 The wcsftime( ) Function

This function is defined in the header **<wchar.h>**.

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *format,
       const struct tm *timptr);
```

The *wcsftime*( ) function places wide-character codes into the array pointed to by *wcs* as controlled by the string pointed to by *format*. No more than *maxsize* wide characters are placed into the array.

This function behaves as if the character string produced by the *strftime*( ) function had been converted to a wide character string by calling *mbstowcs*( ). Otherwise this interface is identical to *strftime*( ).

### 4.6.3 The strptime( ) Function

This function is defined in the header **<time.h>**.

```
char   *strptime(const char *buf, const char *format, struct tm *tm);
```

This function performs the reverse operation to *strftime*( ) and converts the character string pointed to by *buf* to a time value. This is stored in the **tm** structure pointed to by *tm*, using the format string pointed to by *format*. If successful, *strptime*( ) returns a pointer to the location character following the last character in *buf*. Otherwise it returns a null pointer.

The *format* is a character string containing zero or more fields as follows,

- one or more white-space characters (as defined by the *isspace*( ) function)
- an ordinary character (neither % nor a white-space character)
- a conversion specification.

Each conversion specification is introduced by a % character, followed by a conversion character which specifies the replacement required.

A directive composed of white-space characters or a series of directives composed of %n, %t, white-space characters or any combination, is executed by scanning up to the first character that is not white space, or until no more characters can be scanned. A directive that is an ordinary character is executed by scanning the next character from the buffer. Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. The scanned characters are compared to the local values associated with the conversion specifier. If a match is found, values for the appropriate **tm** structure members are set to values corresponding to the locale information. If no match is found, *strptime*( ) fails.

**Conversions**

The full list of conversion specifications is given in the **XSH** specification, but at least the following are supported on all conforming implementations:

%a the locale's full or abbreviated weekday name.
%A as %a.
%b the locale's full or abbreviated month name.
%B as %b.
%c the locale's appropriate date and time representation.
%C the locale's long format date and time representation.
%d day of the month as a decimal number [01,31]; leading zeros are permitted but are not required.
%D date as %m / %d / %y.
%e same as %d.
%h as %b.
%H hour (24-hour clock) as a decimal number [00,23]; leading zeros are permitted but are not required.
%I hour (12-hour clock) as a decimal number [01,12]; leading zeros are permitted but are not required.
%j day of the year as a decimal number [001,366].
%m month as a decimal number [01,12]; leading zeros are permitted but are not required.
%M minute as a decimal number [00,59]; leading zeros are permitted but are not required.
%n newline character.
%p the locale's equivalent of either AM or PM.
%r the time in AM / PM notation according to U.K. / U.S. conventions (%I:%M:%S %p).
%R the time in 24 hour notation (%H:%M).
%S seconds as a decimal number [00,61]; leading zeros are permitted but are not required.
%t tab character.
%T the time as %H:%M:%S.
%U week number of the year as a decimal number [00,53] (Sunday is the first day of the week); leading zeros are permitted but are not required.
%w weekday as a decimal number [0,6] (Sunday is the first day of the week); leading zeros are permitted but are not required.
%W as %w, except Monday is the first day of the week.
%x the locale's appropriate date representation.
%X the locale's appropriate time representation.
%y the year without century as a decimal number [00,99]; leading zeros are permitted but are not required.
%Y the year with century as a decimal number.
%% is replaced by %.

Case is ignored when matching items such as month or weekday names (see the **XSH** specification).

Some conversion specifiers can be modified by the E and O modifier characters (see the **XSH** specification) to indicate that an alternative format or specification should be used by the unmodified directive. If the alternative format or specification does not exist in the current locale, the behaviour is as if the unmodified directive is used.

**Examples**

The following example accepts a date and time string as its input, which it converts internally to a **tm** structure by calling *strptime*( ). The format of the input is fixed (locale-independent) as an aid to usability.

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

#define PROG_NAME  "settm"
#define USAGE      1
#define ERROR      2

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

   setlocale(LC_ALL, "");
   catd = catopen(PROG_NAME, NL_CAT_LOCALE);

   if(argc != 2) {
      puts(
         catgets(catd, NL_SETD, USAGE,
         "usage: settm date-and-time"));
      exit(1);
   }

   if(strptime(argv[1], "%A %B %d %T", &tm)
      == NULL) {
      puts(
         catgets(catd, NL_SETD, ERROR,
         "incorrect date format"));
      exit(2);
   }
   ...
}
```

where date and time is expected as a string of the form:

```
Sun Mar 31 14:23:00
```

## 4.7 Printing Functions

X/Open has extended the definition of the printing functions to make their operation more tenable in an international environment. These changes affect the following interfaces:

| Interface |
|---|
| *fprintf*( ) |
| *printf*( ) |
| *sprintf*( ) |
| *vfprintf*( ) |
| *vprintf*( ) |
| *vsprintf*( ) |

### 4.7.1 Numbered Argument Lists

Conversion specifications have been updated such that conversions can be applied to the *n*th argument in the argument list rather than to the next unused argument. This type of conversion specification is introduced by the sequence %*digit*$, where *digit* is a value in the range [1,{NL_ARGMAX}] (defined in **<limits.h>**, giving the position of the argument in the argument list.

Similarly, in format strings containing the above form of a conversion specification, a field width or precision can be indicated by the sequence **digit*$. Again *digit* is a number in the range [1,{NL_ARGMAX}], giving the position of the associated argument in the argument list.

As an example:

```
fmt = catgets(catd, NL_SETD, FMT,
     "%1$.*2$d:%3$.*2$d:%4$.*2$d\n");

printf(fmt, hour, width, min, sec);
```

where the value of *width* is used to specify the precision for each of the *d* conversion specifications.

Note that numbered and unnumbered conversion specifications should not be mixed in a single format string. The results of doing this are undefined and, even though it may be allowed by some implementations, there is no guarantee that mixed format strings are portable to other X/Open systems. Also specifying a numbered argument *n* requires that all leading arguments, from the first to the *n*th, are specified in the format string.

### 4.7.2 Decimal-point Characters

All the printing functions listed at the start of this section allow for the insertion of a language-dependent decimal-point (that is, radix) character in the output string. This affects f, e, E, g and G conversions. The decimal-point character is defined by the value of item RADIXCHAR in the program's locale. In the POSIX locale, or in a locale where a decimal-point character is not defined, the decimal-point character defaults to a period (**.**).

### 4.7.3 Thousands Grouping Characters

The list of flag characters has been extended to allow for the formatting of the integer portion of decimal conversions with the locale's thousands (non-monetary) grouping character. This flag is defined in a format string by specifying an apostrophe (for example, `"%'d"`). No thousands grouping character is defined in the POSIX locale.

### 4.7.4 Wide-character Conversions

The list of conversion characters and their meaning has been updated to include:

C The **wchar_t** argument is converted to an array of bytes representing the character, and the resulting character is written. Behaviour when field width or precision is specified is undefined. The conversion is the same as that expected from the *wctomb*( ) function.

S The argument is a pointer to an array of type **wchar_t**. Wide characters from the array are converted to a sequence of bytes representing characters and the resulting array is written up to (but not including) a terminating null byte. Field precision specifies the maximum number of bytes to be printed. The conversion is the same as that expected from the *wcstombs*( ) function.

Note that these conversion characters are not supported by the *printw*( ), *mvprintw*( ), *mvwprintw*( ) and *wprintw*( ) functions (see the **Curses** specification).

## 4.8 Scanning Functions

The formatted-input conversion functions have also been updated to provide operation more suited to an international environment. The following interfaces are affected by these changes:

| Interface |
|---|
| *fscanf*( ) |
| *scanf*( ) |
| *sscanf*( ) |

### 4.8.1 Numbered Argument Lists

Like *printf*( ), the definition of conversion specifications has been updated such that conversions can be applied to a numbered argument list. These are introduced by the sequence %*digit*$, where *digit* is a number in the range [1,{NL_ARGMAX}], giving the position of the associated argument in the argument list.

For example:

```
int i; float x; char name[50]; char *fmt;

      fmt = catgets(catd, NL_SETD, FMT,
        "%2$2d %1$f %*d %3$[0-9]");

      scanf(fmt, &x, &i, name);
```

assuming the default format string and with the input

```
56789 0123 56a72
```

assigns 56 to *i*, 789.0 to *x*, skip 0123, and places the string "56\0" in *name*. The next call to *fgetc*( ) returns **a**.

Note that the assignment suppressing character(*****) is not affected and works the same as it would with plain % conversion specifiers.

Also note that the behaviour of ranges within scansets ([0-9] in the above example) is implementation-defined. Whether they take notice of collating sequence information in a program's locale, or whether they use simple character code ordering, is not specified by X/Open. For maximum portability, it is recommended that expressions of this form should be avoided wherever possible.

Like *printf*( ), numbered and unnumbered conversion specifications should not be mixed in the same format string. Similar warnings also apply to using all arguments up to the highest value of *n* specified in a format string.

### 4.8.2 Decimal-point Characters

The *scanf*( ) function in all its forms allows for the detection of language-dependent radix characters. This affects *e*, *f* and *g* conversions. The radix character is defined by the value of item RADIXCHAR in the program's locale. In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (**.**).

### 4.8.3 Wide-character Conversions

The list of conversion characters has been updated to include:

C Matches a sequence of characters of the number specified by the field width (1 by default). The sequence is then converted to wide-character format and the resulting wide characters are stored in the corresponding argument, which is a pointer to the initial wide character of an array large enough to accept the resulting sequence. The conversion is the same as that expected from the *mbtowc*( ) function. No null byte is added. If field width is specified, the behaviour is unspecified.

S Matches a sequence of characters that are not white space. The sequence is then converted to wide-character format and the resulting wide characters are stored in the corresponding argument, which is a pointer to the initial wide character of an array large enough to accept the resulting sequence, including a terminating null wide character. The conversion is the same as that expected from the *mbstowcs*( ) function. If field width is specified, it denotes the maximum number of characters to accept.

## 4.9 Number Conversion Functions

Three new functions have been added that provide conversion from wide-character strings to various numeric formats:

| Interface |
|---|
| *wcstod*( ) |
| *wcstol*( ) |
| *wcstoul*( ) |

### 4.9.1 The wcstod( ) Function

This function is defined in the header **<wchar.h>**.

```
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

The *wcstod*( ) function converts the initial portion of the wide-character string pointed to by *nptr* to a type **double**. First it decomposes the input into three parts:

- an initial, possibly empty sequence of white-space characters (as defined by *iswspace*( ))
- a subject sequence interpreted as a floating-point constant
- a final sequence of one or more unrecognised wide-character codes.

It then attempts to convert the subject sequence to a floating-point number and returns the result.

The expected form of a subject sequence is:

- an optional + or – sign
- a sequence of digits optionally containing a radix character
- an optional exponent part consisting of e or E, followed by an optional sign, followed by one or more decimal digits.

The radix character is defined by the value of item RADIXCHAR in the program's locale. In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (**.**).

Note that implementation-defined forms of a subject sequence may be supported in other than the POSIX locale.

If the subject sequence is empty or does not have the expected form, no conversion is performed and a pointer to the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If successful, the *wcstod*( ) function returns the converted value as a type **double**. If the conversion could not be performed, zero is returned and *errno* may be set to indicate the error. If the correct value is outside the representable range of values, ±HUGEVAL is returned (according to the sign of the value) and *errno* is set to [ERANGE]. If the correct value would cause underflow, *wcstod*( ) returns zero and *errno* is set to [ERANGE].

**Note:** Be aware that zero is also a valid return value from this function. Applications needing to check for errors should set *errno* to zero before calling *wcstod*( ). If the value of *errno* is non-zero on return, a failure has occurred.

### 4.9.2 The wcstol( ) Function

This function is defined in the header **<wchar.h>**.

```
long int   wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

The *wcstol*( ) function converts the initial portion of the wide-character string pointed to by *nptr* to a type **long int**. First it decomposes the input into three parts:

- an initial, possibly empty sequence of white-space characters (as defined by *iswspace*( ))
- a subject sequence interpreted as an integer represented in some radix determined by the value of *base*
- a final sequence of one or more unrecognised characters

It then attempts to convert the subject sequence to a long integer and returns the result.

If the value of *base* is zero, the subject sequence is that of an integer constant, optionally preceded by a **+** (plus) or – (minus) sign. If the value of *base* is between 2 and 36, the expected form of a subject sequence is a sequence of letters or digits, where the letters a (or A) to z (or Z) are ascribed the values 10 to 35. Only letters whose ascribed values are less than the *base* are permitted. If the value of *base* is 16, the characters 0x or 0X can precede the sequence of letters and digits, following the sign (if present).

If the subject sequence is empty or does not have the expected form, no conversion is performed and a pointer to the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the POSIX locale, additional implementation-defined subject sequence forms may be accepted.

If successful, the *wcstol*( ) function returns the converted value as a type **long int**. If the conversion could not be performed, zero is returned and *errno* may be set to indicate the error. If the correct value is outside the representable range of values, {LONG_MAX} or {LONG_MIN} is returned (according to the sign of the value) and *errno* is set to [ERANGE].

### 4.9.3 The wcstoul( ) Function

This function is defined in the header **<wchar.h>**.

```
unsigned long int   wcstoul(const wchar_t *nptr, wchar_t **endptr,
                int base);
```

This function is similar to the *wcstol*( ) function except that the return vale is an **unsigned long int**. Also, if the correct value is outside the range of representable values, {ULONG_MAX} is returned and *errno* is set to [ERANGE].

## 4.10 ISO C Multi-byte Functions

As already explained in Section 1.5 on page 7, a multi-byte character is a sequence of one or more bytes representing a single graphic symbol or control code. A wide character code is an integral data type of type **wchar_t**, which is large enough to hold any member of the largest extended execution character set supported by an implementation.

Text recorded or generated externally to the program is normally encoded as a series of multi-byte characters. To process these internally, it is recommended that they are first converted to wide character representation.

The **XSH** specification defines various functions for converting between multi-byte character and wide character representations. These functions, and the type **wchar_t** and **size_t**, are all defined in the header **<stdlib.h>**. The behaviour of these functions may be affected by the setting of locale category LC_CTYPE.

### 4.10.1 The mblen() Function

```
int    mblen(const char *s, size_t n);
```

If *s* is not a null pointer, this function determines the number of bytes constituting the character pointed to by *s*. If an invalid character is encountered, –1 is returned. In no case is the return value greater than *n* or the value of MB_CUR_MAX. If *s* is a null pointer, the function returns non-zero or zero depending on whether character encodings have or do not have state-dependent encodings.

### 4.10.2 The mbstowcs() Function

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

This function converts a sequence of characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding wide-character codes. No more than *n* wide-character codes are stored in the array pointed to by *pwcs*. The function returns either the number of wide-character codes stored in *pwcs*, which is not greater than *n*, or (**size_t**)–1 if an invalid character is encountered in *s*. The array is terminated by a null wide character unless the return value is *n*.

If *pwcs* is a null wide character pointer, the *mbstowcs*( ) function returns the number of elements required for the wide character array.

### 4.10.3 The mbtowc() Function

```
int    mbtowc(wchar_t *pwc, const char *s, size_t n);
```

If *s* is not a null pointer, this function determines the number of bytes that constitute the character pointed to by *s*, converts this to the corresponding wide-character representation and stores the result in the wide character pointed to by *pwc*. In no case is the return value greater than *n* or the value of MB_CUR_MAX.

If *s* is a null pointer, the function returns non-zero or zero depending on whether character encodings have or do not have state-dependent encodings. Otherwise if *s* is not a null pointer, the function returns zero (if *s* points to a null character), or the number of bytes that constitute the converted character (if the next *n* or fewer bytes form a valid character), or –1 (if they do not form a valid character).

### 4.10.4 The wcstombs( ) Function

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

The *wcstombs*( ) function converts the sequence of wide-character codes pointed to by *pwcs* into a sequence of characters that begins in the initial shift state and stores the results in the array pointed to by *s*. No more than *n* bytes are stored in this array.

The function returns the number of bytes stored in the array, excluding the terminating null character, or *n* (whichever is smaller). If an invalid wide-character code is encountered, (**size_t**)–1 is returned.

If *s* is a null pointer, the *wcstombs*( ) function returns the number of bytes required for the character array.

### 4.10.5 The wctomb( ) Function

```
int    wctomb(char *s, wchar_t wchar);
```

This function converts the wide character *wchar* to its multi-byte character representation and stores the result in the array pointed to by *s*. No more than {MB_CUR_MAX} bytes are stored in this array.

If *s* is a null pointer, the function returns non-zero or zero depending on whether character encodings have or do not have state-dependent encodings. Otherwise if *s* is not a null pointer, the function returns the number of bytes that constitute the converted multi-byte character, or –1 if the value of *wchar* does not correspond to a valid character.

## 4.11 I/O Functions

The functions described in the previous section allow applications to convert between multi-byte character representation and wide character representation. This could be used, for example, to maintain data on file store in character representation, while manipulating it internally in its wide-character form.

Functions such as *printf* and *scanf* remove the need for programs to perform this conversion manually, by defining conversion characters that convert between wide character and multi-byte character representations automatically (%C and %S).

The **XSH** specification also defines a number of other I/O functions that perform this conversion, as follows:

| **Interface** |
|---|
| *getwc*( ) |
| *getwchar*( ) |
| *fgetwc*( ) |
| *fgetws*( ) |
| *fputwc*( ) |
| *fputws*( ) |
| *putwc*( ) |
| *putwchar*( ) |
| *ungetwc*( ) |

These functions, the data types **wint_t** and **wchar_t**, and the constant WEOF are defined in the header **<wchar.h>**.

### 4.11.1 The getwc( ) Function

```
wint_t    getwc(FILE *stream);
```

The *getwc*( ) function is equivalent to the *fgetwc*( ) function, except that if it is implemented as a macro, evaluation of the stream may occur more than once, so the argument should never be an expression with side effects.

### 4.11.2 The getwchar( ) Function

```
wint_t    getwchar(void);
```

The function call *getwchar*( ) is equivalent to *getwc*(stdin).

### 4.11.3 The fgetwc( ) Function

```
wint_t    fgetwc(FILE *stream);
```

The *fgetwc*( ) function obtains the next character from the input stream, converts it to wide-character representation and advances the file position indicator for *stream*.

If successful, *fgetwc*( ) returns a wide character encoded as a value of type **wint_t**. If a read error occurs, or if the data obtained from the input stream does not form a valid character, or the stream is at end-of-file, the constant WEOF is returned and *errno* is set to indicate the error.

### 4.11.4 The fgetws( ) Function

```
wchar_t   *fgetws(wchar_t *ws, int n, FILE *stream);
```

The *fgetws*( ) function reads characters from *stream*, converts them to wide-character codes and stores the results in the array of type **wchar_t** pointed to by *ws*. The operation is terminated when *n*–1 characters have been read, or a newline character is read and transferred to *ws*, or an end-of-file condition is encountered. The wide-character string is then terminated with a null wide character value.

If successful, *fgetws*( ) returns *ws*. If a read error occurs, or if the data obtained from the stream does not form a valid character, or the stream is at end-of-file, a null pointer is returned and *errno* is set to indicate the error.

### 4.11.5 The fputwc( ) Function

```
wint_t    fputwc(wint_t wc, FILE *stream);
```

The *fputwc*( ) function writes the character corresponding to the wide-character code *wc* to the output stream pointed to by *stream*. If successful, *fputwc*( ) returns the value it has written. Otherwise it returns WEOF and sets *errno* to indicate the error.

### 4.11.6 The fputws( ) Function

```
int       fputws(const wchar_t *ws, FILE stream);
```

The *fputws*( ) function writes a character string corresponding to the (null-terminated) wide-character string pointed to by *ws* to the stream pointed to by *stream*. The null terminating wide character is not written.

If successful, *fputws*( ) returns a non-negative number. Otherwise it returns –1 and sets *errno* to indicate the error.

### 4.11.7 The putwc( ) Function

```
wint_t    putwc(wint_t wc, FILE *stream);
```

The *putwc*( ) function is equivalent to the *fputwc*( ) function, except that if it is implemented as a macro, evaluation of the stream may occur more than once, so the argument should never be an expression with side effects.

### 4.11.8 The putwchar( ) Function

```
wint_t    putwchar(wint_t wc);
```

The function call *putwchar*(*wc*) is equivalent to *putwc*(*wc*, stdout)

### 4.11.9 The ungetwc() Function

```
wint_t    ungetwc(wint_t wc, FILE *stream);
```

The *ungetwc*( ) function pushes the character corresponding to the wide-character code *wc* back onto the input stream pointed to by *stream*. Characters pushed back are returned by subsequent reads on that stream in the reverse order of their pushing. Any intervening call to a file-positioning function causes characters pushed back to be discarded. The external storage associated with *stream* is unchanged.

Note that only one pushed back character is guaranteed. Also if the value of *wc* is WEOF, the call fails and the input stream is unchanged.

If successful, *ungetwc*( ) returns the wide-character code corresponding to the pushed-back character. Otherwise it returns WEOF.

## 4.12 String Functions

In addition to the collation functions described in Section 4.4 on page 52, the following functions are defined in the **XSH** specification for manipulating wide-character strings:

| Interface | |
|---|---|
| *wcscat* | *wcsncpy* |
| *wcschr* | *wcspbrk* |
| *wcscmp* | *wcsrchr* |
| *wcscpy* | *wcsspn* |
| *wcscspn* | *wcstok* |
| *wcslen* | *wcswcs* |
| *wcsncat* | *wcswidth* |
| *wcsncmp* | *wcwidth* |

These functions and the types **wchar_t**, **size_t** and **wint_t** are defined in the header **<wchar.h>**.

### 4.12.1 The wcscat( ) Function

```
wchar_t   *wcscat(wchar_t *ws1, const wchar_t *ws2);
```

This function appends a copy of the wide-character string pointed to by *ws2*, including the terminating null wide-character code, to the end of the wide-character string pointed to by *ws1*. It returns *ws1* if successful. No return value is reserved to indicate an error.

### 4.12.2 The wcschr( ) Function

```
wchar_t   *wcschr(const wchar_t *ws, wint_t wc);
```

This function locates the first occurrence of *wc* in the wide-character string pointed to by *ws*. It returns a pointer to the wide-character code if successful, or a null pointer.

### 4.12.3 The wcscmp( ) Function

```
int       wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

This function compares the wide-character strings pointed to by *ws1* and *ws2*. The sign of a non-zero value returned by *wcscmp*( ) is determined by the sign of the difference between the values of the first pair of wide characters that differ in the objects being compared.

The *wcscmp*( ) function returns an integer greater than, equal to or less than zero, depending on whether the wide string pointed to by *ws1* is greater than, equal to or less than the wide string pointed to by *ws2*.

### 4.12.4 The wcscpy( ) Function

```
wchar_t   *wcscpy(wchar_t *ws1, const wchar_t *ws2);
```

This function copies the wide-character string pointed to by *ws2*, including the terminating null wide character, into the array pointed to by *ws1*. It returns *ws1* if successful. No return value is reserved to indicate an error.

### 4.12.5 The wcscspn( ) Function

```
size_t    wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

This function computes the number of initial elements in the wide-character string pointed to by *ws1* that consist entirely of wide-character codes *not* included in the wide-character string pointed to by *ws2*. It returns *ws1*. No return value is reserved to indicate an error.

### 4.12.6 The wcslen( ) Function

```
size_t    wcslen(const wchar_t *ws);
```

This function computes the number of wide-character codes in the wide-character string pointed to by *ws*, not including the terminating null wide-character code. It returns this number if successful. No return value is reserved to indicate an error.

### 4.12.7 The wcsncat( ) Function

```
wchar_t   *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

This function is similar to *wcscat*( ), except that at most *n* wide-character codes from *ws2* are appended to *ws1*. The initial character from *ws2* overwrites the null wide-character code at the end of *ws1*. A terminating null wide-character code is always appended to the result.

### 4.12.8 The wcsncmp( ) Function

```
int       wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

This function is similar to *wcscmp*( ), except that at most *n* wide-character codes are compared.

### 4.12.9 The wcsncpy( ) Function

```
wchar_t       *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

This function is similar to *wcscpy*( ), except that at most *n* wide-character codes are copied from the array pointed to by *ws2* to the array pointed to by *ws1*.

### 4.12.10 The wcspbrk( ) Function

```
wchar_t       *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

This function locates the first occurrence of any wide-character code from the wide-character string pointed to by *ws2* in the wide string pointed to by *ws1*. It returns a pointer to a wide-character code, or a null pointer.

### 4.12.11 The wcsrchr( ) Function

```
wchar_t   *wcsrchr(const wchar_t *ws, wint_t wc);
```

This function locates the last occurrence of *wc* in the wide-character string pointed to by *ws*. It returns a pointer to a wide-character code, or a null pointer.

### 4.12.12 The wcsspn( ) Function

```
size_t    wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

This function computes the number of initial elements in the wide-character string pointed to by *ws1* that consist entirely of wide-character codes included in the wide-character string pointed to by *ws2*. It returns the number of initial elements satisfying the above criteria. No return value is reserved to indicate an error.

### 4.12.13 The wcstok( ) Function

```
wchar_t   *wcstok(wchar_t *ws1, const wchar_t *ws2);
```

A sequence of calls to this function decomposes the wide-character string pointed to by *ws1* into a series of tokens, each of which is delimited by a wide-character code from the wide-character string pointed to by *ws2*. The first call in the sequence has *ws1* as its first argument; subsequent calls pass a null pointer as this argument. The wide-character string pointed to by *ws2* may change between calls.

The *wcstok*( ) function returns a pointer to the first character of the next token, or a null pointer if there are no more tokens.

### 4.12.14 The wcswcs( ) Function

```
wchar_t   *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
```

This function locates the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide-character string pointed to by *ws2*. It returns a pointer to the located wide-character string, or a null pointer. If *ws2* points to a zero length wide-character string, the function returns *ws1*.

### 4.12.15 The wcswidth( ) Function

```
int       wcswidth(const wchar_t *pwcs, size_t n);
```

This function determines the number of column positions required for *n* wide-character codes (or fewer if a null wide character is encountered in the input string before *n* has expired) in the string pointed to by *pwcs*. It returns zero if *pwcs* points to a null wide character, or the number of column positions required, or –1 if *pwcs* is not a printing wide-character code.

### 4.12.16 The wcwidth( ) Function

```
int       wcwidth(wint_t wc);
```

This function determines the number of column positions required for the wide-character code *wc*. The value of *wc* must be representable as a type **wchar_t**, and must be a wide-character code corresponding to a valid character in the current locale. It returns zero if *wc* is a null wide-character code, or the number of column positions required, or –1 if *wc* is not a printing wide-character code.

## 4.13 Error Handling

Two functions that produce error messages are defined for use by internationalised applications. The language of messages produced by these functions is locale-specific.

### 4.13.1 The perror( ) Function

This function is declared in the header **<stdio.h>**.

```
void      perror(const char *s)
```

The *perror*( ) function maps the error number in *errno* to a language-dependent error message, which is written to the standard output stream as follows:

- If *s* is not a null pointer or a pointer to an empty string, the string pointed to by *s* followed by a colon and space.
- An error message followed by a newline character.

Because it is not possible for a program to retrieve an error message string for internal use, possible applications of this interface are strictly limited.

### 4.13.2 The strerror( ) Function

This function is declared in the header **<string.h>**.

```
char      *strerror(int errnum)
```

The *strerror*( ) function maps the error number in *errnum* to a locale-dependent error message string and returns a pointer thereto. The string pointed to should not be modified by a program, but may be overwritten by subsequent calls to *strerror*( ). Upon successful completion *strerror*( ) returns a pointer to the generated message string. The contents of the error message strings should be determined by the setting of the LC_MESSAGES category. On error, *errno* may be set, but no return value is reserved to indicate an error.

This interface is generally more useful than *perror*( ), as it allows programs to build their own error message strings, for example:

```
...
FILE *fp = fopen(argv[1], "w");
int  open_error = errno;

if (fp == (FILE*) NULL) {
   printf(
      catgets(catd, NL_SETD, ERROR,
         "%s: fopen fails: %s\n"),
      argv[0],
      strerror(open_errno));
   exit(1);
}
...
```

where the constant ERROR is a message identifier assumed to be defined elsewhere in the program. This would produce a message of the form:

```
progname: fopen fails: Too many open files
```

The second and third arguments access localisation data and could be translated to the language of the caller.

## 4.14 Codeset Conversion

The **XSH** specification defines a set of functions that provide codeset conversion capabilities from the program level. These require that the user initialise a codeset conversion stream, identifying both the source and the target codesets. Codeset names are implementation-defined.

Once initialised, character sequences encoded in the source codeset can be passed to the *iconv*( ) function, which performs the conversions and returns equivalent sequences encoded in the target codeset.

These functions and the data type **iconv_t** are defined in the header **<iconv.h>**.

### 4.14.1 The iconv_open( ) Function

```
iconv_t   iconv_open(const char *tocode, const char *fromcode);
```

The *iconv_open*( ) function performs the necessary initialisations to convert character encodings from the source codeset identified by *fromcode* to the codeset identified by *tocode* and returns a conversion descriptor of type **iconv_t**.

If unsuccessful, *iconv_open*( ) returns (**iconv_t**)–1 and sets *errno* to indicate the error.

Settings of *fromcode* and *tocode*, and permitted combinations thereof, are implementation-dependent.

### 4.14.2 The iconv( ) Function

```
size_t    iconv(iconv_t cd, const char **inbuf, size_t *inbytesleft,
          char **outbuf, size_t *outbytesleft);
```

The *iconv*( ) function converts a sequence of characters from one codeset in the array specified by *inbuf*, into a sequence of corresponding characters encoded in another codeset, in the array specified by *outbuf*. The codesets are those specified in the *iconv_open*( ) call that returned the conversion descriptor *cd*.

On input, *inbytesleft* indicates the number of bytes to the end of the input buffer to be converted. Similarly, *outbytesleft* indicates the number of bytes available to the end of the out put buffer. These values are decremented while the conversions are being done, such that on return they indicate the state of their associated buffers.

If *iconv*( ) encounters a character in the input buffer that is legal but for which an identical character does not exist in the target encoding, *iconv*( ) performs an implementation-defined conversion on the character. If no error occurs, *iconv*( ) returns the number of non-identical conversions that were performed. Otherwise, *iconv*( ) updates the variables to reflect the extent of the conversion, returns (**size_t**)–1, and sets *errno* to indicate the error.

### 4.14.3 The iconv_close( ) Function

```
void      iconv_close(iconv_t cd);
```

The *iconv_close*( ) deallocates the conversion descriptor *cd* and all other resources allocated by the *iconv_open*( ) function. Upon successful completion, zero is returned. Otherwise –1 is returned and *errno* is set to indicate the error.

### 4.14.4 Examples

The following example shows how these functions might be used to implement a simple codeset conversion filter, which accepts settings of *tocode* and *fromcode* as input arguments.

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

#define  ICONV_DONE( )  (r >= 0)
#define  ICONV_INVAL( ) ((r == -1) && (errno == EILSEQ))
#define  ICONV_OVER( )  ((r == -1) && (errno == E2BIG))
#define  ICONV_TRUNC( ) ((r == -1) && (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], NL_CAT_LOCALE);

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

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

```
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"));
         iconv_close(cd);
      exit(2);
   }

   if (ICONV_TRUNC( ) || ICONV_OVER( ))

   /*
    * Data remaining in buffer - copy
    * it to the beginning
    */

   memcpy(ibuf, ip, ileft);

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

   /*
    * 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"));
       iconv_close(cd);
    exit(3);
    }

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

Chapter 5

# Using Internationalised Software

This chapter explains how environment variables are used to indicate a user's requirements for culture-dependent data.

## 5.1 Language Announcement

The facilities described so far in this Guide have referred primarily to the development of internationalised software. Another area that needs to be considered is how users indicate their requirements for specific language operation.

X/Open systems support a language announcement mechanism that provides for:

- single language working, where a System Administrator can define default language operation
- mixed language working, where different users of a system can work in different languages
- multi-locale working, where a single user can work in a mixture of locales.

Language announcement is implemented via a set of environment variables, which correspond to locale categories defined for the *setlocale*( ) function.

### 5.1.1 General Announcement

An environment variable *LANG* is defined to indicate a user's requirements for specific language, territory and codeset operation, where a unique setting of *LANG* is reserved to identify each supported locale. Each locale has associated with it instances of collating sequence, character conversion and character classification tables, language information, and message catalogues.

The value set in *LANG* is a locale-name of the form:

LANG=*language*[*_territory*][*.codeset*]

where the length of the entire string cannot exceed {NL_LANGMAX} bytes. The set of characters, excluding separators, is restricted to alphanumeric characters defined in the portable character set.

Note that permissible settings of *language*, *territory* and *codeset* are not standardised and may vary from system to system.

**Note:** All examples in this section assume a particular implementation and should not be taken to imply specific naming conventions.

On its own, *language* selects the required language and implies implementation-defined settings for *territory* and *codeset*. If other than these defaults are required, then *territory* or *codeset* can also be specified. For example:

```
LANG=fr
```

might select the French language, French cultural conventions and the ISO 8859-1: 1987 standard codeset. This could be modified by setting:

```
LANG=fr_CH
```

which selects Swiss cultural conventions instead of French, or

```
LANG=fr.6937
```

which selects the ISO 6937: 1983 standard codeset instead of ISO 8859-1: 1987 standard, or:

```
LANG=fr_CH.6937
```

which selects both Swiss cultural conventions and the ISO 6937: 1983 standard codeset.

An alternative general announcement capability is provided by the *LC_ALL* environment variable, which has the same syntax as described above. The difference between *LANG* and *LC_ALL* is that *LC_ALL* takes precedence over all other international environment variables, whereas *LANG* has the lowest precedence and can be used together with the category-specific environment variables (see next section) to provide multi-language working.

For a complete description of the hierarchy and precedence of international environment variables see Section 4.1.3 on page 44.

### 5.1.2 Announcement Categories

*LANG* and *LC_ALL* provide a general announcement mechanism by which users can identify their overall language requirements. This is sufficient when a single set of localisation data covers all aspects of a user's requirements for native language working. It does not cover the case where (for example) a user wishes to interface to the system in one language and process text files recorded in a different language.

This is catered for by defining additional environment variables, one for each of the following categories supported by the *setlocale*( ) function:

LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME

These identify a user's requirements for language, territory and codeset operation with respect to string collation, character classification and case conversion, message translations, currency symbol and monetary value formats, numeric data representation, and time formats respectively. If any of these are not defined in the current environment, *LANG* provides the necessary default settings.

As an example, if a user wanted to interface with an application in French, but required to sort German language text files, *LANG* and *LC_COLLATE* might be defined as follows:

```
LANG=fr
LC_COLLATE=de_DE
```

Note that *LC_ALL* cannot be used in this situation. Because *LC_ALL* takes precedence over all other international environment variables when the program locale is initialised from the environment, the assignment to *LC_COLLATE* in the above example would have no effect.

The category-specific environment variables are also defined to accept an additional modifier field. This allows the user to select a specific instance of localisation data within a single locale category. Again the permitted settings of this field are not defined by X/Open, but typically they would allow users to select between such things as character and dictionary ordering.

The right-hand side of category-specific environment variables is thus defined as:

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

The *modifier* part can be applied to any valid setting of the *language*, *territory* and *codeset* part, that is:

```
LC_COLLATE=de@dict
LC_COLLATE=de_CH@dict
LC_COLLATE=de.6937@dict
LC_COLLATE=de_CH.6937@dict
```

Permitted settings of the *modifier* field are not standardised and may or may not be specified by individual X/Open systems.

### 5.1.3 The POSIX Locale

In general locale-names are implementation-defined and cannot be relied on to have the same value on different systems. The POSIX locale, however, is guaranteed to exist on all X/Open systems and can be useful in situations where it is important to run an application in a well-known or standard environment. Non-internationalised applications always use this environment, and internationalised software can be forced to run in it by setting *LC_ALL* as follows:

```
LC_ALL=POSIX
```

*LC_ALL* is preferable to *LANG* in this case because it causes all locale categories to be set, irrespective of whether other category-specific environment variables are initialised or not.

Setting the POSIX locale explicitly is quite different from not setting the international environment variables, or setting them to the empty string, either of which causes *setlocale*( ) to behave in an implementation-defined manner.

### 5.1.4 Single Language Working

Assuming the international environment variables are not set, processing takes place in an implementation-defined default locale. This may vary from system to system and should not be relied on by either System Administrators or users.

System Administrators have the option of fixing the default language operation of a system by declaring the appropriate international environment variables in command interpreter start-up files. For example, if the *sh* command is the normal command interpreter, the file **/etc/profile** could contain the following assignment:

```
LANG=en_US.88591
```

which would ensure English language operation, using U.S. customs and the ISO 8859-1:1987 standard codeset, unless otherwise overridden by specific user action (see next section).

This situation is complicated by the ability to specify different entry programs for each user of an X/Open system. If any of these identify a program other than *sh*, the above file is not executed and hence language operation again reverts to an implementation-defined default.

Most programs that can be used as initial entry programs have some sort of start-up file, which can be modified with something equivalent to the above shell assignment. Failing this, another alternative is to use *sh* as the start-up program and modify the user's **$HOME/.profile** file to pass control to the required entry program. This ensures that the environment is set up correctly, as *sh* executes **/etc/profile** first, which may set the *LANG* environment variable. The following example shows how to pass control to the new entry program (altprog):

```
$ cat $HOME/.profile
#
#      Dummy script to cause sh to initialise
#      the environment and then pass control
#      to another command interpreter.
#

exec altprog
```

This achieves the desired results without affecting the user image, because the only actions performed by *sh* are to initialise the environment and then pass control to the alternative entry program.

This issue will be addressed more fully in the future.

### 5.1.5 Mixed Language Working

As the X/Open announcement mechanism works through the environment, of which there is one per process, it follows that each process can specify different and quite unrelated language requirements. Hence each user of the system can, if required, specify language requirements and have them associated with all subsequent processes. This applies to processes set off directly by the user, and to background processes initiated by commands such as *at* and *batch*.

Assuming *sh* is being used, there are two ways users can override the system default and specify their own language requirements:

- They can make appropriate assignments in their **$HOME/.profile** file. This file is executed (after **/etc/profile**) whenever a user logs in to the system, or when the shell is invoked by one of the *exec* functions and the first character of argument zero is –.
- Thy can assign a language to one of the international environment variables directly from the command stream.

The first usage identified above allows a user to set up the international environment such that specific language operation results at each log-in. The second usage allows language operation to be adjusted between individual commands.

Caveats about the use of other command interpreters apply as in the previous section.

The following is an interactive example:

```
$ LANG=en date       (to set English)
Thu Feb 01 13:36:48 GMT 1990
$ LANG=fr date       (to set French)
jeu fév 01 13:36:56 GMT 1990
$ LANG=de date       (to set German)
Don Feb 01 13:37:08 GMT 1990
```

At this point it is worth noting a number of special provisions made by *sh* for operation in an international environment:

- Changes to the international environment variables take immediate effect, both within the shell itself and with respect to the environment of subsequent processes.

- The international environment variables are exported implicitly. For example, in the statement:

  ```
  LANG=foo; export LANG
  ```

  specific binding of *LANG* to the external environment is not necessary; that is, ‘‘export LANG’’ could have been omitted.

- Assignments of the form:

  ```
  LANG=sp cmd
  (LANG=sp; cmd)
  ```

  are equivalent and only affect the environment of the process created to execute *cmd*.

The first provision indicated above has implications for operation of the shell itself, in that all subsequent, related actions (for example, filename expansion) are affected by any changes to the shell's locale. This also affects the language of internally generated shell messages.

### 5.1.6 Mixed-locale Working

This refers to the ability of a user to specify that different aspects of program operation are to be related to different locales. For example, an English user of the *sort* program might require to collate Italian language text files, or a user at a terminal supporting the ISO 8859-1: 1987 standard codeset might want to process files using ISO 6937: 1983 standard character encodings.

These requirements are satisfied by provision of the category-specific environment variables described in Section 5.1.2 on page 88, which allow individual locale categories to be initialised with different localisation data.

## 5.2 Commands and Utilities

X/Open systems support an internationalised utility environment in which standard commands and utilities described in the **XCU** specification operate according to the language requirements of each user. As with other applications, users state their requirements for language operation via the international environment variables described in Section 5.1 on page 87.

The effects of internationalisation on individual commands is described in the **XCU** specification. The following sections of this Guide describe generic effects on standard commands and utilities by locale category.

Note that standard commands and utilities initialise their locale from the environment; that is, by a call to *setlocale*( ) of the form:

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

### 5.2.1 LC_COLLATE

*LC_COLLATE* affects the behaviour of internationalised regular expressions (see Section 5.3 on page 97), and any utility that orders text. A prime example of this is the *sort* command, which sorts one or more files according to collating sequence information defined in the current locale.

As an example, the following data:

```
abcdef
abc
abcd
hot
cheque
cull
abcd
LLLLLL
ABCDE
lot
czech
cult
efgh
chess
llllll
abcde
ábcd
àbcd
```

when sorted in French and Spanish language locales with the following commands:

```
$ LANG=fr sort sortdata >french
$ LANG=sp sort sortdata >spanish
$ sdiff -w40 french spanish
```

results in the differences illustrated in the following list.

The French result is in the first column and the Spanish result is in the second column:

```
abc             abc
abcd            abcd
abcd            abcd
á́bcd            á́bcd
à̂bcd            à̂bcd
ABCDE           ABCDE
abcde           abcde
abcdef          abcdef
cheque    <
chess     <
cull      <
cult            cult
          >     cull
czech           czech
          >     cheque
          >     chess
efgh            efgh
hot             hot
          >     lot
LLLLLL          LLLLLL
llllll          llllll
lot       <
```

Notice that in the Spanish example, the character sequences **ch** and **ll** are treated as multi-character collating symbols that collate after **c** and **l** respectively. Also note the ordering of accented characters within the equivalence class of base character **a** (**a**, **á**, **à**).

Examples of ways in which *LC_COLLATE* affects other commands include:

- *expr*
  String values are compared using collating sequence information in the program's locale. This might yield surprising results in an expression of the form:

  ```
  expr $reply = "y"
  ```

  where $reply contains **ÿ** (y-umlaut) and *LC_COLLATE* only applies primary weighting. In this case **ÿ** and **y** would compare equally.

- *comm*, *join* and *uniq*
  For all these commands the value of *LC_COLLATE* should be equal to the value it had when the input files were sorted, otherwise they may produce incorrect results.

- *ls*
  *LC_COLLATE* determines the order in which output is sorted (ascending collation order by default).

### 5.2.2 LC_CTYPE

*LC_CTYPE* affects the behaviour of internationalised regular expressions (see Section 5.3 on page 97) and utilities that need to perform some sort of character classification or case conversion. Character *type* information in a program's locale is commonly used by parsing functions to detect such things as field delimiters (for example, white-space characters) and alphabetic or numeric character sequences.

As an example, the **wc** command counts lines, words and bytes in a file or files, where a word is defined as a maximal string of characters delimited by a white-space character (as defined by the *isspace*( ) function).

Other examples of commands and utilities affected by *LC_CTYPE* include:

- *ed*, for determining which characters are classified as non-printing for the *l* (list) command.
- *grep*, for determining which characters are defined as letters (character class *alpha*) and identifying shift information for the **–i** option.
- *sort*, for determining which characters are defined as alphanumeric, space and tab (**–d** option), for folding the case of letters (**–f** option), and for identifying non-printing characters (**–i** option).
- *yacc*, for determining which characters are defined as letters and digits (for *name* fields).

### 5.2.3 LC_MONETARY

None of the standard commands and utilities currently defined in the **XCU** specification are affected by *LC_MONETARY*.

### 5.2.4 LC_NUMERIC

*LC_NUMERIC* affects the behaviour of commands that handle decimal numbers. For example, the **–n** option of the *sort* command alters the default ordering rules such that lines in a file are sorted by arithmetic value. A number is defined to consist of:

- optional blanks
- an optional minus sign
- zero or more digits with an optional decimal-point character.

The decimal-point character is defined by the value of item RADIXCHAR in the program's locale (category LC_NUMERIC), and may vary from one language to another.

Users should take care, when typing in numeric data, to note what number formats are expected in the current locale. A misplaced dot or comma could have expensive consequences in a commercial environment.

### 5.2.5 LC_MESSAGES

*LC_MESSAGES* determines the language of program messages, which it does by directing the operation of the *catopen*( ) function when selecting message localisations (see Section 3.5 on page 33). It also affects the behaviour of commands that prompt the user for a yes/no (y/n) type of response. For example, the **–i** (interactive) option of the *rm* command causes the user to be prompted for an affirmative response before a file is deleted. In the POSIX locale, the affirmative response character is defined as **y**. In other locales, it is defined by the regular expression string found in YESEXPR (category LC_MESSAGES).

All the commands and utilities that use XPG4 message catalogues are affected by *LC_MESSAGES*. Some examples are:

- *ln* and *mv*
  If the target file exists and its mode forbids writing, the user is asked for an affirmative response before the link is created.
- *find*
  If the –**ok** expression is used, the user is prompted for an affirmative response before the associated *cmd* is executed.
- *tar*
  Under the **w** option, *tar* prints the action to be taken, followed by the name of the file, and waits for the user's confirmation before proceeding.

### 5.2.6 LC_TIME

*LC_TIME* affects the format of date and time information produced by various commands. For example, the *date* command prints date and time information, optionally under the control of a user-specified format string, as follows:

```
$ date
Thu Feb 01 15:08:25 GMT 1990
$ date "+%A, %B %d, %Y"
Thursday, February 01, 1990
```

In a French language locale this produces:

```
$ date
jeu fév 01 15:10:36 GMT 1990
$ date "+%A, %B %d, %Y"
jeudi, février 01, 1990
```

And in a German language locale it produces:

```
$ date
Don Feb 01 15:12:14 GMT 1990
$ date "+%A, %B %d, %Y"
Donnerstag, Februar 01, 1990
```

Some examples of commands that output date and time information and are affected by *LC_TIME* include:

- *ar*, *cpio* and *tar*, when listing the contents of an archive with the **v** modifier
- *lpstat*, when displaying line-printer status information with the –**a**, –**o**, –**p**, –**t** or –**u** options
- *ls*, when listing the contents of a directory with the –**g**, –**l**, –**n** or –**o** options
- *mail* and *mailx*, when displaying date information in mail headers
- *pr*, when including date and time information in page headers
- *ps*, when generating a full listing with the –**f** option
- *who*, when writing the *time* field of output lines

### 5.2.7 8-bit Transparency

X/Open defines that all commands and utilities presented in the **XCU** specification are transparent to 8-bit data. This includes all file or character data processed by a command, and file store object names.

One implication of this provision is that it should be possible to transfer 8-bit data between systems. However, for the following reasons, this may not always be the case:

- Inter-system mail may be restricted to 7-bit data by underlying mail or network protocols.
- 8-bit data and filenames may not be portable to non-X/Open systems.

Under these circumstances, it is recommended that only characters defined in the Portable Character Set be used for data transfer, and only characters defined in the Portable Filename Character Set be used for naming transfer files.

## 5.3 Regular Expressions

In the past, regular expressions have provided pattern matching facilities in terms of machine collating sequences, the English language and the ASCII coded character set. In an international environment, these facilities need to be extended to cover other languages and other codesets.

Limitations in the syntax of regular expressions also make it difficult to produce constructs that can be applied to different languages, and in particular to languages with accented characters or multi-character collating elements.

The syntax of regular expressions has beed updated to overcome these shortcomings and, hence, to provide a definition of REs that is tenable in an international environment. In doing this, the following objectives were set by X/Open:

- Existing programs should continue to work correctly, without requiring that they are modified to allow for new syntax.
- The syntax should provide for the specification of character classes that are independent of language and codeset, similar to the facilities offered by functions like *isalpha*( ) in the C programming language.
- The syntax should provide for the specification of a base character on its own, or a base character with diacritical marks (for example, **a**, **á**, **à**, and so on).
- The syntax should provide for the specification of two or more characters as a single collating element (for example, **ll** in Spanish).

Character expressions, including character ranges, should also be independent of machine collating sequences. These should work instead off collating sequence information defined in the caller's locale.

### 5.3.1 Character Class Expressions

Within square brackets, an expression of the form:

```
[:class:]
```

is defined as a character class expression, where *class* can be one of the following:

| | |
|---|---|
| alpha | a letter |
| upper | an upper-case letter |
| lower | a lower-case letter |
| digit | a decimal digit |
| xdigit | a hexadecimal digit |
| alnum | an alphanumeric character (letter or digit) |
| space | a character producing white space in displayed text |
| punct | a punctuation character |
| print | a printing character |
| graph | a character with a visible representation |
| cntrl | a control character |
| blank | a blank character. |

Character class expressions provide for the specification of character types in a way that is independent of language and codeset. For example, the command:

```
grep '^[[:upper:]].*[[:punct:]]$' file
```

searches *file* for lines beginning with an upper-case letter and ending with a punctuation character. This expression continues to work correctly irrespective of the language and codeset used by characters in *file*, provided a matching language and codeset is specified in the calling locale.

### 5.3.2 Equivalence Classes

Within square brackets, an expression of the form:

```
[=c=]
```

is defined as an equivalence class. This matches any collating element having the same relative order (primary weighting) in the current collating sequence as **c**.

As an example, if **e**, **é** and **è** belong to the same equivalence class, then the command:

```
expr $var : '[[=e=]]fgh'
```

matches the character sequences:

```
efgh
éfgh
èfgh
```

### 5.3.3 Collating Symbols

Within square brackets, an expression of the form:

```
[.cc.]
```

is defined as a collating symbol. Multi-character collating elements (that is, *n*-character sequences such as *ch* and *ll* in Spanish) must be represented as collating symbols to distinguish them from single character collating elements. For example, the expression:

```
[ch].*
```

is treated as a simple list that matches **c** or **h** followed by any sequence of characters, whereas:

```
[[.ch.]].*
```

defines a collating symbol that only matches words beginning with the character sequence **ch**. If these expressions are applied to the following:

```
camper
champer
hamper
```

the first expression matches with all three words, while the collating symbol expression only matches with the word *champer*.

Note that if the string **ch** is not a valid collating symbol in the current locale, the expression is treated as invalid. This normally results in programs displaying some sort of error message.

### 5.3.4 Ranges

In range expressions of the form:

```
[c-c]
```

*c* can be a single character collating element, a collating symbol or an equivalence class, but not a character class expression. Thus:

```
[[.ch.]-e]
```

matches any element that collates between **ch** and **e** inclusive, while:

```
[a-[=e=]]
```

matches any element that collates between **a** and **e**, including accented forms of **e** defined in the same equivalence class. An expression of the form **[a-e]** only matches up to the unadorned form of **e**.

```
[[:digit:]-[:alpha:]]
```

is an example of an invalid expression; that is, because character class expressions cannot be used as range terminators.

### 5.3.5 Affected Commands

At a minimum, the following commands defined in the **XCU** specification support Internationalised regular expressions:

| Interface |
|---|
| *awk* |
| *ed* |
| *egrep* |
| *expr* |
| *grep* |
| *pg* |
| *sed* |

### 5.3.6 Shell Meta Notation

Changes have also been made to the shell's meta notation for matching filenames. On X/Open-compliant systems, the syntax for filename matching patterns is now defined as:

* Matches any string, including the empty string.

? Matches any single character.

[ . . . ] Matches any one of the enclosed characters. Characters in the list may be:

- single character collating elements, [*c*].
- collating symbols, [[.*cc*.]].
- range expressions, [*c*-*c*].
- character classes, [[:*class*:]].
- equivalence classes, [[=*c*=]].

A pair of characters separated by a dash (**-**) matches any symbol that collates between the pair, as defined by collating sequence information in the shell's locale. If the first

character following the opening square bracket is an exclamation mark (!), any character not enclosed is matched.

### 5.3.7 Examples and Warnings

In an international environment, a command of the form:

```
rm [a-z]*
```

removes all files whose names begin with a lower-case letter in the range **a** to **z**. If dictionary ordering is defined in the shell's locale, the above expression may also match filenames beginning with the upper-case letters **A** to **Z**. Conversely, it would fail to match lower-case letters that collate after **z** in, for example, the Scandinavian languages.

In an international environment, an expression of the above form is dangerous and should not be used unless the user is certain what may happen. The correct way to match lower-case letters is to use a pattern of the form:

```
rm [[:lower:]]*
```

which works predictably in all locales.

Because of the dangers described above, it is also recommended that shell scripts imported from non-internationalised environments be executed in the POSIX locale. This supports the ASCII set of characters, the English language (usually), U.S. customs, and is the locale most closely matching that of earlier UNIX environments.

Further examples include:

```
cat [[=a=]]*
```

which matches any filename beginning with a character in the same equivalence class as **a**, and

```
cp [[.ch.][.ll.]]* dir
```

which matches any filenames beginning with the collating symbols **ch** or **ll**.

## 5.4 The iconv Utility

The **XCU** specification defines generic facilities for handling data recorded in different coded character sets, although it makes no recommendations about which codesets should be supported. X/Open systems may support different codesets for specific geographic areas, or to provide compatibility with an existing field population of systems, or to support a particular application (for example, ISO mail systems).

All X/Open systems provide the *iconv* utility for converting to and from supported codesets. Since the codesets themselves are not standardised, the set of possible codeset conversions is implementation-dependent.

### 5.4.1 Function and Interface

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

The *iconv* utility converts the encoding of characters in *file* from the codeset identified by *fromcode* to the codeset identified by *tocode*. The converted file is written to standard output. If no *file* argument is given on the command line, *iconv* reads from standard input.

Character encodings in either codeset may include single-byte values (for example, for West European and North American usage), or multi-byte values (for example, for Asian usage). The handling of invalid characters in the input stream is implementation-defined.

Settings of *fromcode* and *tocode* are implementation-defined. Users are advised to consult system documentation to determine the settings for a particular implementation.

As an example:

```
iconv -f local8 -t IS88591 guide > guide.out
```

converts the contents of **guide** from a local 8-bit codeset (**local8**) to ISO 8859-1: 1987 standard and writes the results to file **guide.out**.

Chapter 6

# Locale Utilities

This chapter describes the *localedef* and *locale* utilities.

## 6.1 The localedef Utility

Various facilities are provided for defining localisation data; that is, locale-specific instances of collating sequence, character classification and character conversion information, language information and message translations. Provisions for generating message localisations are described in Chapter 3. Facilities for defining all other types of locale-specific information are provided by the *localedef* utility.

### 6.1.1 Function and Interface

```
localedef [-c] [-f charmap] [-i sourcefile] name
```

The *localedef* utility converts source definitions for locale categories into a localised form suitable for use by functions and utilities defined to support internationalised behaviour. It is this information, or some adaptation thereof, that is loaded into a program's locale by the *setlocale*( ) function.

The *localedef*( ) utility reads source definitions from *sourcefile* (if specified) or standard input and writes its output to the target locale identified by *name*.

Source definitions are accepted for one or more locale categories belonging to the same locale. Each category source definition in the input is identified by the corresponding category name, and terminated by an END *category-name* statement, for example:

```
LC_CTYPE
...
END LC_CTYPE
```

It is also permitted for the input to contain source for implementation-defined categories, provided these are defined after the standard categories.

The following standard categories are supported:

LC_CTYPE
LC_COLLATE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME

The *charmap* argument identifies a character set description file (see next section), which contains a mapping of character symbols and collating element symbols to actual character encodings. This option must be specified if symbolic names are used in the locale definition file (*sourcefile*).

The **–c** option directs *localedef* to create permanent output even if warning messages are issued.

### 6.1.2 Character Set Description Files

A character set description file defines the symbolic names and values of character codes in a supported coded character set. Each character set description file must contain source statements for all characters defined in the Portable Character Set and the Control Code Character Set (see **XCU** specification), and may define encoding for additional characters supported by the implementation.

The format of a character set description file is as follows:

```
CHARMAP
<code_set_name>
<mb_cur_max>      <value>
<mb_cur_min>      <value>
<escape_char>     <value>
<comment_char>
<symbolic-name>   <value>
<symbolic-name>   <value>
...
END CHARMAP
```

where

<code_set_name> gives the name of the defined coded character set

<mb_cur_max> gives the maximum number of bytes in a character (1 by default)

<mb_cur_min> gives the minimum number of bytes in a character (default <mb_cur_max>)

<escape_char> gives the character used for expressing numeric values (backslash by default)

<comment_char> is the character that, when placed in column 1, indicates that the line is to be ignored (the number-sign (#) by default)

<symbolic-name> is a unique name for each collating element or character

<value> is a character, decimal, octal or hexadecimal constant giving the value of the associated symbol.

The fields of a source statement are separated by one or more blanks (space, tab, and so on).

Symbolic names are only allowed to contain characters from the portable character set and should be delimited by less-than and greater-than characters.

For settings of *value*, a decimal constant is defined as one or more decimal digits preceded by the escape character and lower case d (for example, \d42), an octal constant is defined as one or more octal digits preceded by the escape character (for example, \141), and a hexadecimal constant is defined as one or more hexadecimal digits preceded by the escape character and a lower-case x (for example, \x6a).

Multi-byte values can be presented by concatenating two or more decimal, octal or hexadecimal constants.

Symbolic names must be unique within one character set description file, but two or more symbolic names are allowed to have the same value. It is possible to specify symbolic names for which no encoding exists in the associated codeset by not specifying a value.

Empty lines or lines containing the comment character in the first column position are treated as comments and are ignored.

The following example is intended to illustrate the syntax of a character set description file and does not define a real codeset.

```
CHARMAP
<mb_cur_max>     2
<mb_cur_min>     1
...
<acute-accent>   \047
<grave-accent>   \140
...
<A>              \101
<B>              \102
<C>              \103
<D>              \104
...
<A-grave>        \x60\x41
<A-acute>        \x27\x41
<a-grave>        \x60\x61
<a-acute>        \x27\x61
...
END CHARMAP
```

It is also possible to define ranges of symbolic names using the syntax:

```
<symbolic-name>...<symbolic-name> <value>
```

In this form, symbolic names consist of zero or more non-numeric characters from the portable character set, followed by an integer formed from one or more decimal digits. This is interpreted as a series of symbolic names formed from the common non-numeric stem and each of the integers between the first and second integer inclusive, for example:

```
<j0101>...<j0104>    \x12\x34
```

is interpreted as:

```
<j0101>        \x12\x34
<j0102>        \x12\x35
<j0103>        \x12\x36
<j0104>        \x12\x37
```

### 6.1.3 General Syntax

Within a *localedef* input file, the following general syntax applies:

- Each category source definition starts with a category identifier line and ends with a category trailer line (for example, LC_TIME ... END LC_TIME).
- Within each category, the definition consists of identifiers and operands separated by one or more blanks. Identifiers can be either keywords, identifying particular locale elements, or collating elements. Operands can be characters, collating elements, strings of characters, or values.
- Lines preceding the first category header can be used to change the comment character (# by default) and the escape character (backslash by default). These have the format:

```
comment_char  <char>
escape_char   <char>
```

- Continuation lines are indicated by placing an escape character as the last character on the line. Any blanks preceding a keyword or the first non-blank character on a continuation line are ignored.
- Empty lines or lines containing a comment character character in the first column position of a line are treated as comments and are ignored.

The format of operands is defined in the following sections. When an operand consists of more than one value, successive values in the definition are separated by semicolons. Strings in operands are enclosed between quotation marks.

Characters can be entered in one of the following formats:

- Within a string, the double-quote character, the escape character and the left angle bracket character must be escaped to be interpreted as the character itself.

  **Note:** The **XBD** specification is incorrect in stating that the right angle bracket must be escaped to be interpreted as the character itself.

  Outside strings, the characters:

  ```
  , ; < > escape_char
  ```

  must be escaped to be interpreted as the character itself.
- As an octal constant, specified as the escape character followed by one or more octal digits.
- As a decimal constant, specified as the escape character, followed by lower-case d, followed by one or more decimal digits.
- As a hexadecimal constant, specified as the escape character, followed by lower-case x, followed by one or more hexadecimal digits.
- As a symbolic name enclosed between less-than and greater-than symbols. This notation is only supported if a *charmap* is specified and the symbolic name exists in the *charmap*.

### 6.1.4 LC_CTYPE

The LC_CTYPE category defines character classification, case conversion and various other character attributes. Each operand consists of one or more characters or digits, separated by semicolons. Characters can be entered in any of the ways defined in the previous section, or as a sequence separated by an ellipsis symbol, for example,

```
\x61;...;\x7a
```

This is interpreted as including in the list all characters with an encoded value equal to or higher than the character preceding the ellipsis, and equal to or lower than the value following the ellipsis.

Three types of keyword are recognised:

- **General**

  These include the keyword,

  **copy**

  which specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is used, no other keyword can be specified.

- **Character Classification**

  These include the keywords:

  **upper**
  **lower**
  **alpha**
  **digit**
  **space**
  **cntrl**
  **punct**
  **graph**
  **print**
  **xdigit**
  **blank**

  The operand field of each specifies a list of characters defined as belonging to the associated character class.

  Locale-specific character classes can be defined using the **charclass** keyword. The operand is a list of names of locale-specific character classes. After defining names, character classes can be defined using the same syntax as other character class keywords.

- **Character Conversion**

  These include the keywords:

  **toupper**
  **tolower**

  where the operands define the mapping of lower-case to upper-case and upper-case to lower-case letters respectively. Each mapping in the operand is specified as character pairs, separated by semicolons and enclosed in parentheses (for example, (<a>, <A>);(b,B); and so on).

As an example:

```
LC_CTYPE
#
lower  <a>;<b>;<c>;<c-cedilla>;<d>;...;<z>
upper  A;B;C;Ç;D;...;Z
space  \x20;\x09;\x0a;\x0b;\x0c;\x0d
blank  \040;\011
toupper      (<a>,<A>);(b,B);(c,C);(ç,Ç);(d,D);(z,Z)
END LC_CTYPE
#
```

### 6.1.5 LC_COLLATE

The LC_COLLATE category specifies the codeset collation sequence as a set of collating elements and the rules defining how strings containing these should be ordered. The following capabilities are provided:

- multi-character collating elements
- user-defined ordering of collating elements
- multiple weights and equivalence classes
- one-to-many mapping
- many-to-many substitution
- equivalence class definition
- ordering by weights.

The following keywords are recognised in this category:

**copy**
**collating-element**
**collating-symbol**
**order_start**
**order_end**

**copy**
If present, this keyword identifies the name of an existing locale to be used as the source for the definition of this category, in which case no other keyword can be specified.

**collating-element**
This keyword is used to define multi-character collating elements, in the form:

```
collating-element <symbol> from <string>
```

where <*symbol*> is a string of one or more characters that does not duplicate any symbolic name defined in the current *charmap* file (if any). For example:

```
collating-element <ch> from "<c><h>"
collating-element <e-acute> from "<acute><e>"
collating-symbol <ll> from "ll"
```

**collating-symbol**
This keyword allows symbols to be defined for use in collation sequence statements (that is, between **order_start** and **order_end**). For example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

**order_start**
The **order_start** keyword precedes collation order entries and also defines the number of weights and other collation rules for this collation sequence definition.

The syntax of **order_start** is:

```
order_start <sort-rules>;...;<sort-rules>
```

where the optional operands specify the rules to be applied when comparing strings. The first operand applies to operations performed using primary weight; subsequent operands apply to operations performed using their corresponding weight. Each *sort-rule* is a comma-separated list of collation directives, where the following directives are currently defined:

forward
specifies that string comparison shall proceed from the start to the end of the string.

backward
specifies that string comparison shall proceed from the end to the start of the string. These two directives are mutually exclusive within a single *sort-rule*.

position
specifies that compare operations shall consider the relative position of non-ignored elements in the string, such that if strings compare equal, the element with the shortest distance from the start of the string collates first.

The maximum number of permissible operands is indicated by the setting of {COLL_WEIGHTS_MAX}. If no operands are defined, one **forward** operand is assumed.

For example:

```
order_start forward;backward
```

The **order_start** directive is followed by a series of collating element entries, with the following syntax:

```
<collating-element> <weight>;...;<weight>
```

A collating element can be specified as a character, in any of the permitted forms listed earlier, a **collating-element**, a **collating-symbol**, an ellipsis, or the special symbol UNDEFINED. The order in which collating elements are specified determines the character collation sequence, such that each collating element compares less than the elements following it.

A **collating-element** is used to specify multi-character collating elements and indicates that the associated characters are to be collated as a single unit in the relative order indicated.

A **collating-symbol** is used to define a position in the relative order for use in weights.

An ellipsis symbol indicates that a sequence of characters collates according to their encoded character values, that is:

- An initial ellipsis is interpreted as expanding to all characters with an encoded value greater than NUL and lower than the value of the collating element specified on the next line.
- An ellipsis enclosed between other lines is interpreted as expanding to all characters with an encoded value higher than the value of the collating element on the preceding line, and lower than the value of the collating element on the following line.
- A trailing ellipsis is expanded to all characters with an encoded value higher than the value of the collating element on the preceding line and lower than the highest coded character value in the current codeset.

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

The optional operands are used to define primary, secondary or successive weights for the collating element. These can be expressed as collating elements or as the special symbol IGNORE, and are interpreted to indicate the relative order between elements, rather than their absolute value. The following should be noted:

- A one-to-many mapping is indicated by an operand containing more than one collating element, for example:

  ```
  <eszet> <s><s>
  ```

- The special symbol IGNORE indicates that the collating element should be ignored when comparing strings.
- An empty operand is interpreted as the collating element itself.
- An ellipsis may be used as an operand if the collating element is an ellipsis. This is interpreted as the value of each character defined by the ellipsis.

In the following example the English text has been added for clarity but would not appear in a source file:

```
order_start         forward;backward
UNDEFINED           IGNORE;IGNORE
<LOW>
<space>             <LOW>;<space>
```

indicates (a) string ordering based on primary and secondary weighting, and (b) that all characters not specified in the definition should be ignored for collation purposes (for regular expressions they are ordered first).

```
...                 <LOW>;...
<a>                 <a>;<a>
```

All characters between <space> and <a> have the same primary equivalence class and secondary weights based on their ordinal encoded values.

```
<a-acute>           <a>;<a-acute>
<a-grave>           <a>;<a-grave>
```

The characters <a>, <a-acute> and <a-grave> belong to the same equivalence class. They are differentiated by secondary weighting.

```
<ch>                <ch>;<ch>
<Ch>                <ch>;<Ch>
```

The multi-character collating elements <c><h> and <C><h> represented by the collating symbols <ch> and <Ch> respectively, belong to the same primary equivalence class.

```
<s>                 <s>;<s>
<eszet>             "<s><s>";"<eszet><eszet>"
```

The character <eszet> is given the string "<s><s>" as a weight. Comparisons are performed as if all occurrences of the character <eszet> are replaced by <s><s> (assuming that <s> has the collating weight <s>). If it is necessary to define <eszet> and <s><s> as an equivalence class, a collating element must be defined for the string **ss**.

```
...                 <HIGH>;...
<HIGH>
order_end
```

End of example file.

**order_end**

This keyword terminates collating order entries introduced by the **order_start** keyword.

### 6.1.6 LC_MESSAGES

This category defines the format and values used for affirmative and negative responses. The operands are strings or extended regular expressions assigned to the following keywords:

**copy**
**yesexpr**
**noexpr**
**yesstr**
**nostr**

For example:

```
LC_MESSAGES
#
yesexpr             "([yY][[:alpha:]]*)|(OK)"
noexpr              "[nN][[:alpha:]]*"
yesstr              "yes"
nostr               "no"
#
END LC_MESSAGES
```

The **yesexpr** string is defined as **y** or **Y** followed by any sequence of alphabetic characters, or the sequence OK. The **noexpr** string is defined as **n** or **N** followed by any sequence of alphabetic characters.

If **copy** is present, the operand specifies the name of an existing locale to be used as the source for the definition of this category. In this case, no other keywords can be specified.

### 6.1.7 LC_MONETARY

This category defines the rules and symbols used to format monetary information. The operands are strings or integers. Unspecified keywords, or integer keywords set to –1, are used to indicate that the value is unspecified.

The following keywords are supported:

**copy**
Specifies the name of an existing locale to be used as the source for the definition of this category. If present, no other keywords can be specified.

**int_curr_symbol**
The international currency symbol specified as a string of four characters, where the first three characters contain the currency symbol and the last character is used to separate the symbol from the monetary quantity (for example, "USD ").

**currency_symbol**
The string used as the local currency symbol (for example, "$").

**mon_decimal_point**
The character used as a decimal delimiter (for example, ".").

**mon_thousands_sep**
The character used as a separator for groups of digits to the left of the decimal delimiter (for example, ",").

**mon_grouping**
Defines the size of each group of digits in formatted monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, the size of the previous group (if any) is repeatedly used for the remainder of the digits.

**Note:** This means that the value of the last integer is used to determine the size of subsequent groups.

If the last integer is –1, no further grouping is performed.

**positive_sign**
The string used to indicate a non-negative-valued monetary quantity (for example, "+").

**negative_sign**
The string used to indicate a negative-valued monetary quantity (for example, "–").

**int_frac_digits**
An integer indicating the number of digits to be displayed to the right of a decimal delimiter in monetary values using **int_curr_symbol** (for example, 2)

**frac_digits**
An integer indicating the number of digits to be displayed to the right of a decimal delimiter in monetary values using **currency_symbol** (for example, 2)

**p_cs_precedes**
For non-negative monetary values, set to 1 if **currency_symbol** precedes the value, or 0 if the symbol succeeds it.

**p_sep_by_space**
For non-negative monetary values, set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value, set to 1 if space separates the symbol from the value, or set to 2 if a space separates the symbol and the sign string, if adjacent.

**n_cs_precedes**
For negative monetary values, set to 1 if **currency_symbol** precedes the value, or 0 if the symbol succeeds it.

**n_sep_by_space**
For negative monetary values, set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value, set to 1 if space separates the symbol from the value, or set to 2 if a space separates the symbol and the sign string, if adjacent.

**p_sign_posn**
An integer value indicating where the **positive_sign** should be placed for a non-negative monetary value. The values the operand can take are the same as those defined for the *localeconv*( ) function (see Section 4.5.3 on page 60).

**n_sign_posn**
An integer value indicating where the **positive_sign** should be placed for a negative monetary value.

As an example:

```
#
# Monetary information for Norway
#
LC_MONETARY
#
int_curr_symbol    "NOK "
currency_symbol    "kr"
mon_decimal_point  ","
mon_thousands_sep  "."
mon_grouping       3
positive_sign      ""
negative_sign      "-"
int_frac_digits    2
frac_digits        2
p_cs_precedes      1
p_sep_by_space     0
n_cs_precedes      1
n_sep_by_space     0
p_sign_posn        1
n_sign_posn        2
#
END LC_MONETARY
```

### 6.1.8 LC_NUMERIC

This category defines the rules and symbols for formatting non-monetary numeric values. The operands are strings or integers. For some keywords, the string can only contain integers. Undefined keywords, string values set to the empty string, or integer keywords set to –1, are used to indicate that the value is unspecified.

The following keywords are defined:

**copy**
Specifies the name of an existing locale to be used as the source for the definition of this category. If present, no other keywords can be specified.

**decimal_point**
Specifies the character to be used as the decimal delimiter (for example, ".").

**thousands_sep**
Specifies the character to be used as the separator for grouping digits to the left of the decimal delimiter (for example, ",").

**grouping**
Defines the size of each group of digits in formatted non-monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, the size of the previous group (if any) is repeatedly used for the remainder of the digits.

**Note:** This means that the value of the last integer is used to determine the size of subsequent groups.

If the last integer is –1, no further grouping is performed.

A source description might be defined as follows:

```
#
LC_NUMERIC
#
decimal_point \x2e
thousands_sep \054
grouping      3
#
END LC_NUMERIC
```

### 6.1.9 LC_TIME

This category defines values of field descriptors supported by the *date* command and the *strftime*( ), *wcsftime*( ) and *strptime*( ) functions. The following keywords are defined:

**copy**
Specifies the name of an existing locale to be used as the source for the definition of this category. If present, no other keywords can be specified.

**abday**
Abbreviated weekday names substituted by the %a conversion directive.

**day**
Full weekday names substituted by the %A conversion directive.

**abmon**
Abbreviated month names substituted by the %b conversion directive.

**mon**
Full month names substituted by the %B conversion directive.

**d_t_fmt**
The date and time representation substituted by the %c conversion directive, specified as an *strftime*( ) format string.

**d_fmt**
The date representation substituted by the %x conversion directive, specified as an *strftime*( ) format string.

**t_fmt**
The time representation substituted by the %X conversion directive, specified as an *strftime*( ) format string.

**am_pm**
The representation of ante meridiem and post meridiem strings substituted by the %p conversion directive. The two strings are separated by a semicolon.

**t_fmt_ampm**
The time representation in 12-hour clock format substituted by the %r conversion directive, specified as an *strftime*( ) format string.

**era**
Defines how years are counted and displayed for each era in a locale, where each era description segment has the format:

```
direction:offset:start_date:end_date:era_name:era_format
```

There can be as many era description segments as necessary to describe the different eras.

**era_d_fmt**
Defines the locale's appropriate alternative date format, corresponding to the %Ex field descriptor.

**era_t_fmt**
Defines the locale's appropriate alternative time format, corresponding to the %EX field descriptor.

**era_d_t_fmt**
Defines the locale's appropriate alternative date and time format, corresponding to the %Ec field descriptor.

**era_year**
Defines the format of the year in alternative Era format, corresponding to the %EY field descriptor. This is an optional keyword in the ISO POSIX-2 DIS.

**alt_digits**
Defines alternative symbols for digits as a semi-colon separated string, where the first string is the alternative symbol corresponding to 0 (zero), the second string the alternative symbol corresponding to 1 (one), and so on.

An example of a source specification for this category might be:

```
#
LC_TIME
#
abday          "Sun";"Mon";"Tue";"Wed";\
               "Thu";"Fri";"Sat"
#
day            "Sunday";"Monday";"Tuesday";\
               "Wednesday";"Thursday";"Friday";\
               "Saturday"
#
abmon          "Jan";"Feb";"Mar";"Apr";\
               "May";"Jun";"Jul";"Aug";\
               "Sep";"Oct";"Nov";"Dec"
#
mon            "January";"February";"March";\
               "April";"May";"June";"July";\
               "August";"September";"October";\
               "November"; "December"
#
d_t_fmt        "%a %b %d %H:%M:%S %Z %Y"
#
d_fmt          "%d/%m/%y"
#
t_fmt          "%H:%M:%S"
#
am_pm          "AM";"PM"
#
t_fmt_ampm     "%I:%M:%S %p"
#
END LC_TIME
```

## 6.2 The locale Utility

The *locale* utility is provided to allow users to display information about the current locale environment or all public locales. Command options define what information is produced and written to the standard output.

### 6.2.1 Function and Interface

```
locale [–a | –m ]
locale [–ck] name . . .
```

When *locale* is invoked without operands, it summarises the current locale environment for each locale category, as determined by the setting of the international environment variables. For example:

```
$ locale
LANG=en_GB.646
LC_CTYPE="en_GB.646"
LC_COLLATE="en_GB.646"
LC_TIME=en_US.646
LC_NUMERIC="en_GB.646"
LC_MONETARY="en_GB.646"
LC_MESSAGES="en_GB.646"
LC_ALL=
```

where a value enclosed by double quotes indicates that the setting is implied by the setting of *LANG* or *LC_ALL*.

When invoked with operands, *locale* writes the values assigned to the keywords and locale categories, as follows:

- Specifying a keyword name selects that keyword and the category containing the keyword.
- Specifying a category name selects the named category and all keywords in that category.

### 6.2.2 Options

The following options are defined for this utility:

–**a** Causes information about all public locales to be output. At least the POSIX or C locale exists on all implementations; other locales are implementation-defined. For example:

```
$ locale –a
C
POSIX
en_GB.88591
en_US.88591
fr_FR.88591
de_DE.6937
...
$
```

**–c** Writes the names of selected locale categories. If keywords are also selected for writing, the category name precedes the keyword output for that category. For example:

```
$ locale –c noexpr
LC_MESSAGES
[nN][[:alpha:]]*
$
```

**–k** Writes the names and values of selected keywords. Non-numeric values are written in the format:

```
keyword="value"
```

For example:

```
$ locale –ck LC_MESSAGES
LC_MESSAGES
yesexpr="[yY][[:alpha:]]*"
noexpr="[nN][[:alpha:]]*"
$
```

If the keyword is **charmap**, the name of the charmap (if any) that was specified via the *localedef* –**f** option when the locale was created is written.

If the value is numeric, output is in the form:

```
keyword=value
```

where value may be a decimal value, or an octal or hexadecimal value preceded by the current escape character, for example:

```
$ locale –ck frac_digits
LC_MONETARY
frac_digits=2
$
```

If the –**k** option is not specified, selected keyword values are written without the keyword name.

**–m** Writes the names of all available charmaps, where the output is in a form suitable for use as the argument to the *localedef* –**f** option, for example:

```
$ locale -m
IS646
IS88591
IS6937
$
```

*Appendix A*

# *Migration Issues*

This appendix provides guidelines for upgrading source code from one issue of the Portability Guide to the next. This only refers to functions and interfaces superseded or in some way changed in successive issues. It does not provide a general statement of capability for each issue, which is already described elsewhere in this Guide.

## A.1 XPG2 to XPG3

XPG2 described a set of Internationalisation facilities that had not had extensive market exposure at the time. This issue of the Portability Guide was also published ahead of the ANSI C standard and the POSIX.1-1988 standard, both of which described Internationalisation facilities that X/Open aligned with in XPG3. XPG2 Internationalisation was therefore something of a trial-use definition, and was identified as an optional component of XPG2 systems.

In terms of system interfaces and headers, the following were superseded in XPG3:

- *nl_init*( ), which was replaced by the ANSI and POSIX specified *setlocale*( ) function.
- *nl_cxtime*( ) and *nl_ascxtime*( ), which were replaced by the ANSI *strftime*( ) function.
- *gcvt*( ), which was marked WITHDRAWN in XPG3.
- *nl_printf*( ), *nl_fprintf*( ) and *nl_sprintf*( ), whose functions were merged with the standard printing functions.
- *nl_scanf*( ), *nl_fscanf*( ) and *nl_sscanf*( ), whose functions were merged with the standard scanning functions.
- *nl_strcmp*( ), which was replaced by the ANSI *strcoll*( ) function.
- *nl_strncmp*( ), which was marked WITHDRAWN in XPG3.
- *catgetmsg*( ), which was marked WITHDRAWN in XPG3.

With the exception of *gcvt*( ) and *nl_strncmp*( ), XPG3 defined equivalent facilities to those presented in XPG2, and in most cases it is a simple matter to define a function or macro to map from the XPG2 interface to an equivalent XPG3 interface. For example:

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

#define valid(ptr) (ptr != (char*) NULL)

nl_init (lang)
char   *lang;
{
    /*
     * ensure 'lang' is valid before calling
     * setlocale( ).
     */

    if (valid (lang) && *lang) {
       if (valid (setlocale (LC_ALL, lang)))
           return (0);
       else
           return (-1);
    } else {
       return (-1);
    }
}
```

which converts calls to *nl_init*( ) into equivalent calls of *setlocale*( ). Similarly,

```
#include <nl_types.h>
#include <string.h>
#include <memory.h>

char *
catgetmsg (catd, set_id, msg_id, buf, len)
nl_catd      catd;
int    set_id;
int    msg_id;
char   *buf;
int    len;
{
    strncpy (buf,
       catgets (catd, set_id, msg_id,
           memset (buf, ' ', len)),
       len -1);

    return (buf);
}
```

implements *catgetmsg*( ) via calls to *catgets*( ). The use of *memset*( ) in the above example causes a null string to be returned if the identified message cannot be read from the message catalogue.

Finally, the XPG2 *ctime* functions can be implemented as follows:

```
#include <nl_types.h>
#include <langinfo.h>
#include <time.h>
#include <string.h>
#include <sys/types.h>

#define TBUFSIZE    128

char   _tbuf [TBUFSIZE];
char   _bbuf [TBUFSIZE];

#define format(buf) \
    (strcat (strcpy (buf, \
       nl_langinfo (D_T_FMT)), "\n"))

#define setfmt(fmt) \
    (fmt == (char*) NULL ? format (_bbuf) : \
       (*fmt == '\0' ? format (_bbuf) : fmt))

char *
nl_ascxtime (tm, fmt)
struct tm *tm;
char   *fmt;
{
    int      result;

    result = strftime (_tbuf, TBUFSIZE,
       setfmt (fmt), tm);

    return (result ? _tbuf : asctime (tm));
}

char *
nl_cxtime (clock, fmt)
long   *clock;
char   *fmt;
{
    return (
       nl_ascxtime (localtime (clock), fmt));
}
```

The printing and scanning functions are more simply mapped by **#define** statements, for example,

```
#define      nl_printf    printf
#define      nl_scanf     scanf
       .
       .
       .
```

The *gcvt*( ) function was withdrawn in XPG3 because this function is typically used to implement certain *printf*( ) conversions and should not be used directly by user programs. The *strncmp*( ) function was withdrawn primarily because it could only ever be supported for single-byte codesets (that is, where there was no distinction between characters and bytes). This was deemed too restrictive and hence the function was removed from the interface definition.

All other XPG2 functions were carried forward into XPG3.

## A.2 XPG3 to XPG4

Refer to the **Migration Guide**.

*Appendix B*

# *Example Locales*

This appendix contains sample *localedef* source descriptions. Although these are working examples, it is not implied that such locales are supported by X/Open systems.

Refer to the **X/Open Locale Registry** for approved locales.

## B.1 Example Locale 1

This example assumes a character set description file that defines a Portable Character Set and Control Character Set to ASCII mapping.

```
#
LC_CTYPE
#
upper                   <A>;...;<Z>
lower                   <a>;...;<z>
space                   <tab>;<newline>;<vertical-tab>;\
                        <form-feed>;<carriage-return>;<space>
cntrl                   <alert>;<backspace>;<tab>;<newline>;\
                        <vertical-tab>;<form-feed>;<carriage-return>
punct                   <exclamation-mark>;...;<slash>;\
                        <colon>;...;<commercial-at>;\
                        <left-bracket>;...;<grave-accent>;\
                        <left-brace>;...;<tilde>
digit                   <0>;...;<9>
xdigit                  <0>;...;<9>;<A>;...;<F>;<a>;...;<f>
blank                   <space>;<tab>
#
# Note:Characters classified as upper and lower are
#      automatically classified as alpha. Characters
#      classified as upper, lower, alpha, digit and
#      punct are automatically classified as graph.
#      Characters specified as graph and space are
#      automatically classified as print.

toupper                 (<a>,<A>);(<b>,<B>);(<c>,<C>);\
                        (<d>,<D>);(<e>,<E>);(<f>,<F>);\
                        (<g>,<G>);(<h>,<H>);(<i>,<I>);\
                        (<j>,<J>);(<k>,<K>);(<l>,<L>);\
                        (<m>,<M>);(<n>,<N>);(<o>,<O>);\
                        (<p>,<P>);(<q>,<Q>);(<r>,<R>);\
                        (<s>,<S>);(<t>,<T>);(<u>,<U>);\
                        (<v>,<V>);(<w>,<W>);(<x>,<X>);\
                        (<y>,<Y>);(<z>,<Z>)
#
tolower                 (<A>,<a>);(<B>,<b>);(<C>,<c>);\
                        (<D>,<d>);(<E>,<e>);(<F>,<f>);\
                        (<G>,<g>);(<H>,<h>);(<I>,<i>);\
                        (<J>,<j>);(<K>,<k>);(<L>,<l>);\
```

```
                    (<M>,<m>);(<N>,<n>);(<O>,<o>);\
                    (<P>,<p>);(<Q>,<q>);(<R>,<r>);\
                    (<S>,<s>);(<T>,<t>);(<U>,<u>);\
                    (<V>,<v>);(<W>,<w>);(<X>,<x>);\
                    (<Y>,<y>);(<Z>,<z>)
#
END LC_CTYPE
#
LC_COLLATE
#
# This locale uses string ordering, where:
# 1.                The case of letters is folded on primary
#                   weighting and distinguished on secondary weighting.
# 2.                Special and control characters are ignored.
#
order_start
...                 IGNORE;IGNORE
<0>                 <0>;<0>
...                 ...;...
<9>                 <9>;<9>
...                 IGNORE;IGNORE
<A>                 <A>;<A>
<B>                 <B>;<B>
<C>                 <C>;<C>
<D>                 <D>;<D>
<E>                 <E>;<E>
<F>                 <F>;<F>
<G>                 <G>;<G>
<H>                 <H>;<H>
<I>                 <I>;<I>
<J>                 <J>;<J>
<K>                 <K>;<K>
<L>                 <L>;<L>
<M>                 <M>;<M>
<N>                 <N>;<N>
<O>                 <O>;<O>
<P>                 <P>;<P>
<Q>                 <Q>;<Q>
<R>                 <R>;<R>
<S>                 <S>;<S>
<T>                 <T>;<T>
<U>                 <U>;<U>
<V>                 <V>;<V>
<W>                 <W>;<W>
<X>                 <X>;<X>
<Y>                 <Y>;<Y>
<Z>                 <Z>;<Z>
...                 IGNORE;IGNORE
<a>                 <A>;<a>
<b>                 <B>;<b>
<c>                 <C>;<c>
<d>                 <D>;<d>
```

```
<e>                     <E>;<e>
<f>                     <F>;<f>
<g>                     <G>;<g>
<h>                     <H>;<h>
<i>                     <I>;<i>
<j>                     <J>;<j>
<k>                     <K>;<k>
<l>                     <L>;<l>
<m>                     <M>;<m>
<n>                     <N>;<n>
<o>                     <O>;<o>
<p>                     <P>;<p>
<q>                     <Q>;<q>
<r>                     <R>;<r>
<s>                     <S>;<s>
<t>                     <T>;<t>
<u>                     <U>;<u>
<v>                     <V>;<v>
<w>                     <W>;<w>
<x>                     <X>;<x>
<y>                     <Y>;<y>
<z>                     <Z>;<z>
...                     IGNORE;IGNORE
order_end
#
END LC_COLLATE
#
LC_MESSAGES
#
yesexpr                 "[yY][[:alpha:]]*"
noexpr                  "[nN][[:alpha:]]*"
#
END LC_MESSAGES
#
LC_MONETARY
#
int_curr_symbol         "USD "
currency_symbol         "$"
mon_decimal_point       "."
mon_thousands_sep       ","
mon_grouping            3
positive_sign           ""
negative_sign           "-"
int_frac_digits         2
frac_digits             2
p_cs_precedes           1
p_sep_by_space          0
n_cs_precedes           1
n_sep_by_space          0
p_sign_posn             2
n_sign_posn             2
#
```

```
END LC_MONETARY
#
LC_NUMERIC
#
decimal_point         "."
thousands_sep         ","
grouping              3
#
END LC_NUMERIC
#
LC_TIME
#
abday                 "Sun";"Mon";"Tue";"Wed";\
                      "Thu";"Fri";"Sat"
#
day                   "Sunday";"Monday";"Tuesday";\
                      "Wednesday";"Thursday";"Friday";\
                      "Saturday"
#
abmon                 "Jan";"Feb";"Mar";"Apr";\
                      "May";"Jun";"Jul";"Aug";\
                      "Sep";"Oct";"Nov";"Dec"
#
mon                   "January";"February";"March";\
                      "April";"May";"June";"July";\
                      "August";"September";"October";\
                      "November"; "December"
#
d_t_fmt               "%a %b %d %H:%M:%S %Z %Y"
#
d_fmt                 "%m/%d/%y"
#
t_fmt                 "%H:%M:%S"
#
am_pm                 "AM";"PM"
#
t_fmt_ampm            "%I:%M:%S %p"
#
END LC_TIME
```

## B.2 Example Locale 2

This example assumes a character set description file that defines a Portable Character Set and Control Character set to ISO 8859-1 mapping.

```
#
LC_CTYPE
#
upper                   <A>;...;<Z>;\
                        <A-grave>;...;<O-diaeresis>;\
                        <O-oblique>;...;<eszet>
#
lower                   <a>;...;<z>;\
                        <eszet>;<a-grave>;...;<o-diaeresis>;\
                        <o-oblique>;...;<y-diaeresis>
#
space                   <tab>;<newline>;<vertical-tab>;\
                        <form-feed>;carriage-return>;<space>;\
                        <no-break-space>
#
cntrl                   <NUL>;...;<IS1>;\
                        <DEL>;\
                        <RES1>;...;<APC>
#
punct                   <exclamation-mark>;...;<slash>;\
                        <colon>;...;<commercial-at>;\
                        <left-bracket>;...;<grave-accent>;\
                        <left-brace>;...;<tilde>;\
                        <inverted-exclamation-mark>;...;\
                        <inverted-question-mark>;\
                        <multiplication-sign>;<division-sign>
#
digit                   <0>;...;<9>
#
xdigit                  <0>;...;<9>;<A>;...;<F>;<a>;...;<f>
#
blank                   <space>;<tab>;<no-break-space>
#
# Note:Characters classified as upper and lower are
#      automatically classified as alpha. Characters
#      classified as  upper, lower, alpha, digit and
#      punct are automatically classified as graph.
#
#      Note also that the eszet character is
#      considered in this example as being both an
#      upper-case and lower-case character.
#
toupper                 (<a>,<A>);(<b>,<B>);(<c>,<C>);\
                        (<d>,<D>);(<e>,<E>);(<f>,<F>);\
                        (<g>,<G>);(<h>,<H>);(<i>,<I>);\
                        (<j>,<J>);(<k>,<K>);(<l>,<L>);\
                        (<m>,<M>);(<n>,<N>);(<o>,<O>);\
                        (<p>,<P>);(<q>,<Q>);(<r>,<R>);\
```

```
                (<s>,<S>);(<t>,<T>);(<u>,<U>);\
                (<v>,<V>);(<w>,<W>);(<x>,<X>);\
                (<y>,<Y>);(<z>,<Z>);(<a-grave>,<A>);\
                (<a-acute>,<A>);(<a-circumflex>,<A>);\
                (<a-tilde>,<A>);(<a-diaeresis>,<A>);\
                (<a-ring>,<A>);\
                (<ae-ligature>,<AE-ligature>);\
                (<c-cedilla>,<C>);(<e-grave>,<E>);\
                (<e-acute>,<E>);(<e-circumflex>,<E>);\
                (<e-diaeresis>,<E>);(<i-grave>,<I>);\
                (<i-acute>,<I>);(<i-circumflex>,<I>);\
                (<i-diaeresis>,<I>);\
                (<lowercase-thorn>,<uppercase-thorn>);\
                (<n-tilde>,<N>);(<o-grave>,<O>);\
                (<o-acute>,<O>);(<o-circumflex>,<O>);\
                (<o-tilde>,<O>);(<o-diaeresis>,<O>);\
                (<o-oblique>,<O>);(<u-grave>,<U>);\
                (<u-acute>,<U>);(<u-circumflex>,<U>);\
                (<u-diaeresis>,<U>);(<y-acute>,<Y>);\
                (<y-diaeresis>,<Y>)
#
tolower         (<A>,<a>);(<B>,<b>);(<C>,<c>);\
                (<D>,<d>);(<E>,<e>);(<F>,<f>);\
                (<G>,<g>);(<H>,<h>);(<I>,<i>);\
                (<J>,<j>);(<K>,<k>);(<L>,<l>);\
                (<M>,<m>);(<N>,<n>);(<O>,<o>);\
                (<P>,<p>);(<Q>,<q>);(<R>,<r>);\
                (<S>,<s>);(<T>,<t>);(<U>,<u>);\
                (<V>,<v>);(<W>,<w>);(<X>,<x>);\
                (<Y>,<y>);(<Z>,<z>);\
                (<A-grave>,<a-grave>);\
                (<A-acute>,<a-acute>);\
                (<A-circumflex>,<a-circumflex>);\
                (<A-tilde>,<a-tilde>);\
                (<A-diaeresis>,<a-diaeresis>);\
                (<A-ring>,<a-ring>);\
                (<AE-ligature>,<ae-ligature>);\
                (<C-cedilla>,<c-cedilla>);\
                (<E-grave>,<e-grave>);\
                (<E-acute>,<e-acute>);\
                (<E-circumflex>,<e-circumflex>);\
                (<E-diaeresis>,<e-diaeresis>);\
                (<I-grave>,<i-grave>);\
                (<I-acute>,<i-acute>);\
                (<I-circumflex>,<i-circumflex>);\
                (<I-diaeresis>,<i-diaeresis>);\
                (<uppercase-thorn>,<lowercase-thorn>);\
                (<N-tilde>,<n-tilde>);\
                (<O-grave>,<o-grave>);\
                (<O-acute>,<o-acute>);\
                (<O-circumflex>,<o-circumflex>);\
                (<O-tilde>,<o-tilde>);\
```

```
                    (<O-diaeresis>,<o-diaeresis>);\
                    (<O-oblique>,<o-oblique>);\
                    (<U-grave>,<u-grave>);\
                    (<U-acute>,<u-acute>);\
                    (<U-circumflex>,<u-circumflex>);\
                    (<U-diaeresis>,<u-diaeresis>);\
                    (<Y-grave>,<y-grave>)
#
END LC_CTYPE
#
LC_COLLATE
#
# This locale uses string ordering and assumes that
# {COLL_WEIGHT_MAX} is superior or equal to 3.
#
# The primary weighting doesn't distinguish case
# nor accents.
#
# The second weighting doesn't distinguish case but
# takes accents into account using reverse sorting.
#
# The third weighting distinguishes case.
#
# Special and control characters are ignored.
#
# Characters not used in French are ignored.
#
order_start          forward;backward;forward
...                  IGNORE;IGNORE;IGNORE
<0>                  <0>;<0>;<0>
...                  ...;...;...
<9>                  <9>;<9>;<9>
...                  IGNORE;IGNORE;IGNORE
<A>                  <A>;<A>;<A>
<B>                  <B>;<B>;<B>
<C>                  <C>;<C>;<C>
<D>                  <D>;<D>;<D>
<E>                  <E>;<E>;<E>
<F>                  <F>;<F>;<F>
<G>                  <G>;<G>;<G>
<H>                  <H>;<H>;<H>
<I>                  <I>;<I>;<I>
<J>                  <J>;<J>;<J>
<K>                  <K>;<K>;<K>
<L>                  <L>;<L>;<L>
<M>                  <M>;<M>;<M>
<N>                  <N>;<N>;<N>
<O>                  <O>;<O>;<O>
<P>                  <P>;<P>;<P>
<Q>                  <Q>;<Q>;<Q>
<R>                  <R>;<R>;<R>
<S>                  <S>;<S>;<S>
```

```
<T>                 <T>;<T>;<T>
<U>                 <U>;<U>;<U>
<V>                 <V>;<V>;<V>
<W>                 <W>;<W>;<W>
<X>                 <X>;<X>;<X>
<Y>                 <Y>;<Y>;<Y>
<Z>                 <Z>;<Z>;<Z>
...                 IGNORE;IGNORE;IGNORE
<a>                 <A>;<A>;<a>
<b>                 <B>;<B>;<b>
<c>                 <C>;<C>;<c>
<d>                 <D>;<D>;<d>
<e>                 <E>;<E>;<e>
<f>                 <F>;<F>;<f>
<g>                 <G>;<G>;<g>
<h>                 <H>;<H>;<h>
<i>                 <I>;<I>;<i>
<j>                 <J>;<J>;<j>
<k>                 <K>;<K>;<k>
<l>                 <L>;<L>;<l>
<m>                 <M>;<M>;<m>
<n>                 <N>;<N>;<n>
<o>                 <O>;<O>;<o>
<p>                 <P>;<P>;<p>
<q>                 <Q>;<Q>;<q>
<r>                 <R>;<R>;<r>
<s>                 <S>;<S>;<s>
<t>                 <T>;<T>;<t>
<u>                 <U>;<U>;<u>
<v>                 <V>;<V>;<v>
<w>                 <W>;<W>;<w>
<x>                 <X>;<X>;<x>
<y>                 <Y>;<Y>;<y>
<z>                 <Z>;<Z>;<z>
...                 IGNORE;IGNORE;IGNORE
<a-grave>           <A>;<a-grave>;<a>
<a-acute>           IGNORE;IGNORE;IGNORE
<a-circumflex>      <A>;<a-circumflex>;<a>
<a-tilde>           IGNORE;IGNORE;IGNORE
<a-diaeresis>       <A>;<a-diaeresis>;<a>
...                 IGNORE;IGNORE;IGNORE
<c-cedilla>         <C>;<c-cedilla>;<c>
<e-grave>           <E>;<e-grave>;<e>
<e-acute>           <E>;<e-acute>;<e>
<e-circumflex>      <E>;<e-circumflex>;<e>
<e-diaeresis>       <E>;<e-diaeresis>;<e>
...                 IGNORE;IGNORE;IGNORE
<o-circumflex>      <O>;<o-circumflex>;<o>
<o-tilde>           IGNORE;IGNORE;IGNORE
<o-diaeresis>       <O>;<o-diaeresis>;<o>
...                 IGNORE;IGNORE;IGNORE
<u-circumflex>      <U>;<u-circumflex>;<u>
```

```
<u-diaeresis>        <U>;<u-diaeresis>;<u>
...                  IGNORE;IGNORE;IGNORE
order_end
#
END LC_COLLATE

#
LC_MESSAGES
#
yesexpr              "[oO][[:alpha:]]*"
noexpr               "[nN][[:alpha:]]*"
#
END LC_MESSAGES

#
LC_MONETARY
#
int_curr_symbol      "FFR "
currency_symbol      "FR"
mon_decimal_point    ","
mon_thousands_sep    " "
mon_grouping         3
positive_sign        ""
negative_sign        "-"
int_frac_digits      2
frac_digits          2
p_cs_precedes        0
p_sep_by_space       1
n_cs_precedes        0
n_sep_by_space       1
p_sign_posn          1
n_sign_posn          1
#
END LC_MONETARY

#
LC_NUMERIC
#
decimal_point        ","
thousands_sep        ""
grouping             3
#
END LC_NUMERIC

#
LC_TIME
#
abday                "dim";"lun";"mar";"mer";\
                     "jeu";"ven";"sam"
#
day                  "dimanche";"lundi";"mardi";\
                     "mercredi";"jeudi";"vendredi";\
```

```
                    "samedi"
#
abmon               "jan";"fév";"mar";"avr";\
                    "mai";"jun";"jui";"aoû";\
                    "sep";"oct";"nov";"déc"
#
mon                 "janvier";"février";"mars";\
                    "avril";"mai";"juin";\
                    "juillet";"août";"septembre";\
                    "octobre";"novembre";"décembre"
#
#
d_t_fmt             "%A %d %B %Y %H:%M:%S %Z\n"
#
d_fmt               "%d/%m/%y"
#
t_fmt               "%H:%M:%S"
#
am_pm               "AM";"PM"
#
t_fmt_ampm          "%I:%M:%S %p"
#
END LC_TIME
```

## B.3 Example Locale 3

This example assumes a character set description file that defines a Japanese character set. This is considered to consist of 128 characters in JIS X0201 (the Japanese version of ISO 646), and 8836 characters in JIS X0208-1990 (JIS Kanji).

Note that the character set includes undefined code points in JIS X0208-1990.

The names for JIS Kanji used here are sufficient to identify the above character set. Kanji are given names based on their code value written in hexadecimal. Users may require a more general notation to represent all supported characters in a portable manner.

```
#
#       LC_CTYPE
#
LC_CTYPE

#
# upper class:
#       Uppercase alphabets in portable character set,
#       Roman uppercase letters in JIS X 0208,
#       Greek uppercase letters in JIS X 0208,
#       Russian uppercase letters in JIS X 0208, and
#       Uppercase letters in JIS X 0212.
#       Uppercase letters in udc or vdc classes may be added.

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
        <j0333>;...;<j0358>;\
        <j0601>;...;<j0624>;\
        <j0701>;...;<j0733>;\
        <J0665>;...;<J0669>;\
        <J0671>;\
        <J0673>;\
        <J0674>;\
        <J0676>;\
        <J0734>;...;<J0746>;\
        <J0901>;\
        <J0902>;\
        <J0904>;\
        <J0906>;\
        <J0908>;\
        <J0909>;\
        <J0911>;...;<J0913>;\
        <J0915>;\
        <J0916>;\
        <J1001>;...;<J1024>;\
        <J1026>;...;<J1087>

#
# lower class:
#       Lowercase alphabets in portable character set,
#       Roman lowercase letters in JIS X 0208,
#       Greek lowercase letters in JIS X 0208,
```

```
#       Russian lowercase letters in JIS X 0208, and
#       Lowercase letters in JIS X 0212.
#       Lowercase letters in udc or vdc classes may be added

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
        <j0365>;...;<j0390>;\
        <j0633>;...;<j0656>;\
        <j0749>;...;<j0781>;\
        <J0681>;...;<J0692>;\
        <J0782>;...;<J0794>;\
        <J0933>;...;<J0948>;\
        <J1101>;...;<J1127>;\
        <J1129>;...;<J1135>;\
        <J1137>;...;<J1187>

#
# alpha class (default):
# It includes, by default, all the characters defined in the upper
# class and the lower class.
#

#
# digit class
#
digit   <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>

#
# space class:
#       Space characters defined in ISO DIS 9945-2 "POSIX" locale
#       Space in JIS X 0208
#

space   <tab>;<newline>;<vertical-tab>;<form-feed>;\
        <carriage-return>;<space>;\
        <j0101>

#
# cntrl class:
#       C0 and C1 control characters as per ISO 6429.
#       SS2 and SS3 may be excluded if an accompanying charmap uses
#       them as single shifts to invoke graphic characters (as in EUC).
#       Control characters in udc or vdc may be added.

cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
        <form-feed>;<carriage-return>;\
        <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;\
        <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
        <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
        <IS1>;<DEL>;\
        <BPH>;<NBH>;<NEL>;<SSA>;<ESA>;<HTS>;<HTJ>;\
```

```
        <VTS>;<PLD>;<PLU>;<RI>;<SS2>;<SS3>;\
        <DCS>;<PU1>;<PU2>;<STS>;<CCH>;<MW>;<SPA>;<EPA>;\
        <SOS>;<SCI>;<CSI>;<ST>;<OSC>;<PM>;<APC>

#
# punct class:
# Special characters in udc or vdc may be added as long as they
# do not belong to classes cntrl, alpha, digit, jkanji, jhira,
# jkata or jdigit.
#
punct   <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
        <left-parenthesis>;<right-parenthesis>;<asterisk>;\
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
        <greater-than-sign>;<question-mark>;\
        <commercial-at>;\
        <left-square-bracket>;<backslash>;<yen-sign>;\
        <right-square-bracket>;<circumflex>;<underscore>;\
        <grave-accent>;\
        <left-curly-bracket>;<vertical-line>;<right-curly-bracket>;\
        <tilde>;<overline>;\
        <kana-full-stop>;<kana-opening-bracket>;\
        <kana-closing-bracket>;<kana-comma>;<kana-conjunctive>;\
        <j0102>;...;<j0110>;\
        <j0113>;...;<j0118>;\
        <j0123>;\
        <j0126>;\
        <j0129>;...;<j0194>;\
        <j0201>;...;<j0214>;\
        <j0226>;...;<j0233>;\
        <j0242>;...;<j0248>;\
        <j0260>;...;<j0274>;\
        <j0282>;...;<j0289>;\
        <j0294>;\
        <j0801>;...;<j0832>;\
        <J0215>;...;<J0225>;\
        <J0234>;...;<J0236>;\
        <J0275>;...;<J0281>


#
# graph class:
#       upper, lower, alpha, digit, xdigit
#       JIS X 0201 graphic characters
#       JIS X 0208 graphic characters
#       JIS X 0212 graphic characters
#       Graphic characters in udc or vdc classes may be added.

graph   <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
        <left-parenthesis>;<right-parenthesis>;<asterisk>;\
```

```
<plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
<zero>;<one>;<two>;<three>;<four>;\
<five>;<six>;<seven>;<eight>;<nine>;\
<colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
<greater-than-sign>;<question-mark>;\
<commercial-at>;\
<A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
<N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
<left-square-bracket>;<backslash>;<yen-sign>;\
<right-square-bracket>;<circumflex>;<underscore>;\
<grave-accent>;\
<a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
<n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
<left-curly-bracket>;<vertical-line>;<right-curly-bracket>;\
<tilde>;<overline>;\
<kana-full-stop>;<kana-opening-bracket>;\
<kana-closing-bracket>;<kana-comma>;<kana-conjunctive>;\
<kana-WO>;...;<kana-tsu>;\
<kana-prolonged-sound>;\
<kana-A>;...;<kana-N>;\
<kana-voiced-sound>;<kana-semivoiced-sound>;\
<j0102>;...;<j0194>;\
<j0201>;...;<j0214>;\
<j0226>;...;<j0233>;\
<j0242>;...;<j0248>;\
<j0260>;...;<j0274>;\
<j0282>;...;<j0289>;\
<j0294>;\
<j0316>;...;<j0325>;\
<j0333>;...;<j0358>;\
<j0365>;...;<j0390>;\
<j0401>;...;<j0483>;\
<j0501>;...;<j0586>;\
<j0601>;...;<j0624>;\
<j0633>;...;<j0656>;\
<j0701>;...;<j0733>;\
<j0749>;...;<j0781>;\
<j0801>;...;<j0832>;\
<j1601>;...;<j1694>;\
<j1701>;...;<j1794>;\
<j1801>;...;<j1894>;\
<j1901>;...;<j1994>;\
<j2001>;...;<j2094>;\
<j2101>;...;<j2194>;\
<j2201>;...;<j2294>;\
<j2301>;...;<j2394>;\
<j2401>;...;<j2494>;\
<j2501>;...;<j2594>;\
<j2601>;...;<j2694>;\
<j2701>;...;<j2794>;\
<j2801>;...;<j2894>;\
<j2901>;...;<j2994>;\
```

```
<j3001>;...;<j3094>;\
<j3101>;...;<j3194>;\
<j3201>;...;<j3294>;\
<j3301>;...;<j3394>;\
<j3401>;...;<j3494>;\
<j3501>;...;<j3594>;\
<j3601>;...;<j3694>;\
<j3701>;...;<j3794>;\
<j3801>;...;<j3894>;\
<j3901>;...;<j3994>;\
<j4001>;...;<j4094>;\
<j4101>;...;<j4194>;\
<j4201>;...;<j4294>;\
<j4301>;...;<j4394>;\
<j4401>;...;<j4494>;\
<j4501>;...;<j4594>;\
<j4601>;...;<j4694>;\
<j4701>;...;<j4751>;\
<j4801>;...;<j4894>;\
<j4901>;...;<j4994>;\
<j5001>;...;<j5094>;\
<j5101>;...;<j5194>;\
<j5201>;...;<j5294>;\
<j5301>;...;<j5394>;\
<j5401>;...;<j5494>;\
<j5501>;...;<j5594>;\
<j5601>;...;<j5694>;\
<j5701>;...;<j5794>;\
<j5801>;...;<j5894>;\
<j5901>;...;<j5994>;\
<j6001>;...;<j6094>;\
<j6101>;...;<j6194>;\
<j6201>;...;<j6294>;\
<j6301>;...;<j6394>;\
<j6401>;...;<j6494>;\
<j6501>;...;<j6594>;\
<j6601>;...;<j6694>;\
<j6701>;...;<j6794>;\
<j6801>;...;<j6894>;\
<j6901>;...;<j6994>;\
<j7001>;...;<j7094>;\
<j7101>;...;<j7194>;\
<j7201>;...;<j7294>;\
<j7301>;...;<j7394>;\
<j7401>;...;<j7494>;\
<j7501>;...;<j7594>;\
<j7601>;...;<j7694>;\
<j7701>;...;<j7794>;\
<j7801>;...;<j7894>;\
<j7901>;...;<j7994>;\
<j8001>;...;<j8094>;\
<j8101>;...;<j8194>;\
```

```
<j8201>;...;<j8294>;\
<j8301>;...;<j8394>;\
<j8401>;...;<j8406>;\
<J0215>;...;<J0225>;\
<J0234>;...;<J0236>;\
<J0275>;...;<J0281>;\
<J0665>;...;<J0669>;\
<J0671>;\
<J0673>;\
<J0674>;\
<J0676>;\
<J0681>;...;<J0692>;\
<J0734>;...;<J0746>;\
<J0782>;...;<J0794>;\
<J0901>;\
<J0902>;\
<J0904>;\
<J0906>;\
<J0908>;\
<J0909>;\
<J0911>;...;<J0913>;\
<J0915>;\
<J0916>;\
<J0933>;...;<J0948>;\
<J1001>;...;<J1024>;\
<J1026>;...;<J1087>;\
<J1101>;...;<J1127>;\
<J1129>;...;<J1135>;\
<J1137>;...;<J1187>;\
<J1601>;...;<J1694>;\
<J1701>;...;<J1794>;\
<J1801>;...;<J1894>;\
<J1901>;...;<J1994>;\
<J2001>;...;<J2094>;\
<J2101>;...;<J2194>;\
<J2201>;...;<J2294>;\
<J2301>;...;<J2394>;\
<J2401>;...;<J2494>;\
<J2501>;...;<J2594>;\
<J2601>;...;<J2694>;\
<J2701>;...;<J2794>;\
<J2801>;...;<J2894>;\
<J2901>;...;<J2994>;\
<J3001>;...;<J3094>;\
<J3101>;...;<J3194>;\
<J3201>;...;<J3294>;\
<J3301>;...;<J3394>;\
<J3401>;...;<J3494>;\
<J3501>;...;<J3594>;\
<J3601>;...;<J3694>;\
<J3701>;...;<J3794>;\
<J3801>;...;<J3894>;\
```

```
        <J3901>;...;<J3994>;\
        <J4001>;...;<J4094>;\
        <J4101>;...;<J4194>;\
        <J4201>;...;<J4294>;\
        <J4301>;...;<J4394>;\
        <J4401>;...;<J4494>;\
        <J4501>;...;<J4594>;\
        <J4601>;...;<J4694>;\
        <J4701>;...;<J4794>;\
        <J4801>;...;<J4894>;\
        <J4901>;...;<J4994>;\
        <J5001>;...;<J5094>;\
        <J5101>;...;<J5194>;\
        <J5201>;...;<J5294>;\
        <J5301>;...;<J5394>;\
        <J5401>;...;<J5494>;\
        <J5501>;...;<J5594>;\
        <J5601>;...;<J5694>;\
        <J5701>;...;<J5794>;\
        <J5801>;...;<J5894>;\
        <J5901>;...;<J5994>;\
        <J6001>;...;<J6094>;\
        <J6101>;...;<J6194>;\
        <J6201>;...;<J6294>;\
        <J6301>;...;<J6394>;\
        <J6401>;...;<J6494>;\
        <J6501>;...;<J6594>;\
        <J6601>;...;<J6694>;\
        <J6701>;...;<J6794>;\
        <J6801>;...;<J6894>;\
        <J6901>;...;<J6994>;\
        <J7001>;...;<J7094>;\
        <J7101>;...;<J7194>;\
        <J7201>;...;<J7294>;\
        <J7301>;...;<J7394>;\
        <J7401>;...;<J7494>;\
        <J7501>;...;<J7594>;\
        <J7601>;...;<J7694>;\
        <J7701>;...;<J7767>

#
# print class:
#       <space>, <j0101>,
#       upper, lower, alpha, digit, xdigit
#       JIS X 0201 printable characters
#       JIS X 0208 printable characters
#       JIS X 0212 printable characters
#       Pritable characters in udc or vdc classes may be added.

print   <space>;\
        <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
```

```
<left-parenthesis>;<right-parenthesis>;<asterisk>;\
<plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
<zero>;<one>;<two>;<three>;<four>;\
<five>;<six>;<seven>;<eight>;<nine>;\
<colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
<greater-than-sign>;<question-mark>;\
<commercial-at>;\
<A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
<N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
<left-square-bracket>;<backslash>;<yen-sign>;\
<right-square-bracket>;<circumflex>;<underscore>;\
<grave-accent>;\
<a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
<n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
<left-curly-bracket>;<vertical-line>;<right-curly-bracket>;\
<tilde>;<overline>;\
<kana-full-stop>;<kana-opening-bracket>;\
<kana-closing-bracket>;<kana-comma>;<kana-conjunctive>;\
<kana-WO>;...;<kana-tsu>;\
<kana-prolonged-sound>;\
<kana-A>;...;<kana-N>;\
<kana-voiced-sound>;<kana-semivoiced-sound>;\
<j0101>;...;<j0194>;\
<j0201>;...;<j0214>;\
<j0226>;...;<j0233>;\
<j0242>;...;<j0248>;\
<j0260>;...;<j0274>;\
<j0282>;...;<j0289>;\
<j0294>;\
<j0316>;...;<j0325>;\
<j0333>;...;<j0358>;\
<j0365>;...;<j0390>;\
<j0401>;...;<j0483>;\
<j0501>;...;<j0586>;\
<j0601>;...;<j0624>;\
<j0633>;...;<j0656>;\
<j0701>;...;<j0733>;\
<j0749>;...;<j0781>;\
<j0801>;...;<j0832>;\
<j1601>;...;<j1694>;\
<j1701>;...;<j1794>;\
<j1801>;...;<j1894>;\
<j1901>;...;<j1994>;\
<j2001>;...;<j2094>;\
<j2101>;...;<j2194>;\
<j2201>;...;<j2294>;\
<j2301>;...;<j2394>;\
<j2401>;...;<j2494>;\
<j2501>;...;<j2594>;\
<j2601>;...;<j2694>;\
<j2701>;...;<j2794>;\
<j2801>;...;<j2894>;\
```

```
<j2901>;...;<j2994>;\
<j3001>;...;<j3094>;\
<j3101>;...;<j3194>;\
<j3201>;...;<j3294>;\
<j3301>;...;<j3394>;\
<j3401>;...;<j3494>;\
<j3501>;...;<j3594>;\
<j3601>;...;<j3694>;\
<j3701>;...;<j3794>;\
<j3801>;...;<j3894>;\
<j3901>;...;<j3994>;\
<j4001>;...;<j4094>;\
<j4101>;...;<j4194>;\
<j4201>;...;<j4294>;\
<j4301>;...;<j4394>;\
<j4401>;...;<j4494>;\
<j4501>;...;<j4594>;\
<j4601>;...;<j4694>;\
<j4701>;...;<j4751>;\
<j4801>;...;<j4894>;\
<j4901>;...;<j4994>;\
<j5001>;...;<j5094>;\
<j5101>;...;<j5194>;\
<j5201>;...;<j5294>;\
<j5301>;...;<j5394>;\
<j5401>;...;<j5494>;\
<j5501>;...;<j5594>;\
<j5601>;...;<j5694>;\
<j5701>;...;<j5794>;\
<j5801>;...;<j5894>;\
<j5901>;...;<j5994>;\
<j6001>;...;<j6094>;\
<j6101>;...;<j6194>;\
<j6201>;...;<j6294>;\
<j6301>;...;<j6394>;\
<j6401>;...;<j6494>;\
<j6501>;...;<j6594>;\
<j6601>;...;<j6694>;\
<j6701>;...;<j6794>;\
<j6801>;...;<j6894>;\
<j6901>;...;<j6994>;\
<j7001>;...;<j7094>;\
<j7101>;...;<j7194>;\
<j7201>;...;<j7294>;\
<j7301>;...;<j7394>;\
<j7401>;...;<j7494>;\
<j7501>;...;<j7594>;\
<j7601>;...;<j7694>;\
<j7701>;...;<j7794>;\
<j7801>;...;<j7894>;\
<j7901>;...;<j7994>;\
<j8001>;...;<j8094>;\
```

```
<j8101>;...;<j8194>;\
<j8201>;...;<j8294>;\
<j8301>;...;<j8394>;\
<j8401>;...;<j8406>;\
<J0215>;...;<J0225>;\
<J0234>;...;<J0236>;\
<J0275>;...;<J0281>;\
<J0665>;...;<J0669>;\
<J0671>;\
<J0673>;\
<J0674>;\
<J0676>;\
<J0681>;...;<J0692>;\
<J0734>;...;<J0746>;\
<J0782>;...;<J0794>;\
<J0901>;\
<J0902>;\
<J0904>;\
<J0906>;\
<J0908>;\
<J0909>;\
<J0911>;...;<J0913>;\
<J0915>;\
<J0916>;\
<J0933>;...;<J0948>;\
<J1001>;...;<J1024>;\
<J1026>;...;<J1087>;\
<J1101>;...;<J1127>;\
<J1129>;...;<J1135>;\
<J1137>;...;<J1187>;\
<J1601>;...;<J1694>;\
<J1701>;...;<J1794>;\
<J1801>;...;<J1894>;\
<J1901>;...;<J1994>;\
<J2001>;...;<J2094>;\
<J2101>;...;<J2194>;\
<J2201>;...;<J2294>;\
<J2301>;...;<J2394>;\
<J2401>;...;<J2494>;\
<J2501>;...;<J2594>;\
<J2601>;...;<J2694>;\
<J2701>;...;<J2794>;\
<J2801>;...;<J2894>;\
<J2901>;...;<J2994>;\
<J3001>;...;<J3094>;\
<J3101>;...;<J3194>;\
<J3201>;...;<J3294>;\
<J3301>;...;<J3394>;\
<J3401>;...;<J3494>;\
<J3501>;...;<J3594>;\
<J3601>;...;<J3694>;\
<J3701>;...;<J3794>;\
```

```
            <J3801>;...;<J3894>;\
            <J3901>;...;<J3994>;\
            <J4001>;...;<J4094>;\
            <J4101>;...;<J4194>;\
            <J4201>;...;<J4294>;\
            <J4301>;...;<J4394>;\
            <J4401>;...;<J4494>;\
            <J4501>;...;<J4594>;\
            <J4601>;...;<J4694>;\
            <J4701>;...;<J4794>;\
            <J4801>;...;<J4894>;\
            <J4901>;...;<J4994>;\
            <J5001>;...;<J5094>;\
            <J5101>;...;<J5194>;\
            <J5201>;...;<J5294>;\
            <J5301>;...;<J5394>;\
            <J5401>;...;<J5494>;\
            <J5501>;...;<J5594>;\
            <J5601>;...;<J5694>;\
            <J5701>;...;<J5794>;\
            <J5801>;...;<J5894>;\
            <J5901>;...;<J5994>;\
            <J6001>;...;<J6094>;\
            <J6101>;...;<J6194>;\
            <J6201>;...;<J6294>;\
            <J6301>;...;<J6394>;\
            <J6401>;...;<J6494>;\
            <J6501>;...;<J6594>;\
            <J6601>;...;<J6694>;\
            <J6701>;...;<J6794>;\
            <J6801>;...;<J6894>;\
            <J6901>;...;<J6994>;\
            <J7001>;...;<J7094>;\
            <J7101>;...;<J7194>;\
            <J7201>;...;<J7294>;\
            <J7301>;...;<J7394>;\
            <J7401>;...;<J7494>;\
            <J7501>;...;<J7594>;\
            <J7601>;...;<J7694>;\
            <J7701>;...;<J7767>

#
# xdigit class
#
xdigit  <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>;\
        <A>;<B>;<C>;<D>;<E>;<F>;\
        <a>;<b>;<c>;<d>;<e>;<f>

#
# blank class
#
```

```
blank   <space>;<tab>

#
# Non-standard character classes specific to the ja_JP locale
#

charclass       ascii;line;jdigit;paren;jparen;jisx0201;jisx0201r;\
                jisx0208;jisx0212;udc;vdc;gaiji;jhira;jkata;jhankana;\
                jkanji;jspace
#
# ascii class: characters for which isascii() returns true
#       C0 control characters
#       SPACE and DELETE characters in ASCII (or JIS X 0201 Roman)
#       Graphic characters in ASCII (or JIS X 0201 Roman)

ascii   <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;\
        <alert>;<BEL>;<backspace>;<tab>;<newline>;\
        <vertical-tab>;<form-feed>;<carriage-return>;\
        <SO>;<SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
        <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;<IS1>;\
        <space>;\
        <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
        <left-parenthesis>;<right-parenthesis>;<asterisk>;\
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
        <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>;\
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
        <greater-than-sign>;<question-mark>;\
        <commercial-at>;\
        <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
        <left-square-bracket>;<backslash>;<right-square-bracket>;\
        <circumflex>;<underscore>;\
        <grave-accent>;\
        <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
        <left-curly-bracket>;<vertical-line>;<right-curly-bracket>;\
        <tilde>;<DEL>

#
# line class:
# The line drawing characters in JIS X 0208
# Line drawing characters in udc or vdc classes may be added.
#

line    <j0801>;...;<j0832>

#
# jdigit class: The digit characters in JIS X 0208

jdigit  <j0316>;...;<j0325>
```

```
#
# paren class:
# Parentheses and paired symbols in JIS X 0201 and JIS X 0208.
# Parentheses or paired symbols in udc or vdc classes may be added.
#

paren   <left-parenthesis>;<right-parenthesis>;\
        <left-square-bracket>;<right-square-bracket>;\
        <left-curly-bracket>;<right-curly-bracket>;\
        <kana-opening-bracket>;<kana-closing-bracket>;\
        <j0138>;...;<j0159>

#
# jparen class:
# The kana bracket characters in JIS X 0201 and the parentheses in
# JIS X 0208.  Parentheses or paired symbols in udc or vdc classes
# may be added.
#

jparen  <kana-opening-bracket>;<kana-closing-bracket>;\
        <j0138>;...;<j0159>

#
# jisx0201 class:
# All the printable characters in JIS X 0201.
# Printable characters in udc or vdc classes with their code points
# in undefined area of JIS X 0201 may be added.
#
jisx0201 <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
        <form-feed>;<carriage-return>;\
        <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;\
        <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
        <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
        <IS1>;\
        <space>;\
        <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
        <left-parenthesis>;<right-parenthesis>;<asterisk>;\
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
        <zero>;<one>;<two>;<three>;<four>;\
        <five>;<six>;<seven>;<eight>;<nine>;\
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
        <greater-than-sign>;<question-mark>;\
        <commercial-at>;\
        <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>;\
        <left-square-bracket>;<backslash>;<right-square-bracket>;\
        <circumflex>;<underscore>;\
        <grave-accent>;\
        <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>;\
        <left-curly-bracket>;<vertical-line>;<right-curly-bracket>;\
```

```
        <tilde>;<DEL>;\
        <kana-full-stop>;<kana-opening-bracket>;\
        <kana-closing-bracket>;<kana-comma>;<kana-conjunctive>;\
        <kana-WO>;...;<kana-tsu>;\
        <kana-prolonged-sound>;\
        <kana-A>;...;<kana-N>;\
        <kana-voiced-sound>;<kana-semivoiced-sound>

#
# jisx0201r class:
# All the printable characters in the right hand side of JIS X 0201.
# Printable characters in udc or vdc classes with their code points
# in undefined area of JIS X 0201 right hand side may be added.
#
jisx0201r <kana-full-stop>;<kana-opening-bracket>;\
        <kana-closing-bracket>;<kana-comma>;<kana-conjunctive>;\
        <kana-WO>;...;<kana-tsu>;\
        <kana-prolonged-sound>;\
        <kana-A>;...;<kana-N>;\
        <kana-voiced-sound>;<kana-semivoiced-sound>

#
# jisx0208 class:
# All the printable characters in JIS X 0208.
# Printable characters in udc or vdc classes whose code points are in
# the undefined area of JIS X 0208 may be added.
#

jisx0208 <j0101>;...;<j0194>;\
        <j0201>;...;<j0214>;\
        <j0226>;...;<j0233>;\
        <j0242>;...;<j0248>;\
        <j0260>;...;<j0274>;\
        <j0282>;...;<j0289>;\
        <j0294>;\
        <j0316>;...;<j0325>;\
        <j0333>;...;<j0358>;\
        <j0365>;...;<j0390>;\
        <j0401>;...;<j0483>;\
        <j0501>;...;<j0586>;\
        <j0601>;...;<j0624>;\
        <j0633>;...;<j0656>;\
        <j0701>;...;<j0733>;\
        <j0749>;...;<j0781>;\
        <j0801>;...;<j0832>;\
        <j1601>;...;<j1694>;\
        <j1701>;...;<j1794>;\
        <j1801>;...;<j1894>;\
        <j1901>;...;<j1994>;\
        <j2001>;...;<j2094>;\
        <j2101>;...;<j2194>;\
        <j2201>;...;<j2294>;\
```

```
<j2301>;...;<j2394>;\
<j2401>;...;<j2494>;\
<j2501>;...;<j2594>;\
<j2601>;...;<j2694>;\
<j2701>;...;<j2794>;\
<j2801>;...;<j2894>;\
<j2901>;...;<j2994>;\
<j3001>;...;<j3094>;\
<j3101>;...;<j3194>;\
<j3201>;...;<j3294>;\
<j3301>;...;<j3394>;\
<j3401>;...;<j3494>;\
<j3501>;...;<j3594>;\
<j3601>;...;<j3694>;\
<j3701>;...;<j3794>;\
<j3801>;...;<j3894>;\
<j3901>;...;<j3994>;\
<j4001>;...;<j4094>;\
<j4101>;...;<j4194>;\
<j4201>;...;<j4294>;\
<j4301>;...;<j4394>;\
<j4401>;...;<j4494>;\
<j4501>;...;<j4594>;\
<j4601>;...;<j4694>;\
<j4701>;...;<j4751>;\
<j4801>;...;<j4894>;\
<j4901>;...;<j4994>;\
<j5001>;...;<j5094>;\
<j5101>;...;<j5194>;\
<j5201>;...;<j5294>;\
<j5301>;...;<j5394>;\
<j5401>;...;<j5494>;\
<j5501>;...;<j5594>;\
<j5601>;...;<j5694>;\
<j5701>;...;<j5794>;\
<j5801>;...;<j5894>;\
<j5901>;...;<j5994>;\
<j6001>;...;<j6094>;\
<j6101>;...;<j6194>;\
<j6201>;...;<j6294>;\
<j6301>;...;<j6394>;\
<j6401>;...;<j6494>;\
<j6501>;...;<j6594>;\
<j6601>;...;<j6694>;\
<j6701>;...;<j6794>;\
<j6801>;...;<j6894>;\
<j6901>;...;<j6994>;\
<j7001>;...;<j7094>;\
<j7101>;...;<j7194>;\
<j7201>;...;<j7294>;\
<j7301>;...;<j7394>;\
<j7401>;...;<j7494>;\
```

```
        <j7501>;...;<j7594>;\
        <j7601>;...;<j7694>;\
        <j7701>;...;<j7794>;\
        <j7801>;...;<j7894>;\
        <j7901>;...;<j7994>;\
        <j8001>;...;<j8094>;\
        <j8101>;...;<j8194>;\
        <j8201>;...;<j8294>;\
        <j8301>;...;<j8394>;\
        <j8401>;...;<j8406>

#
# jisx0212 class:
# All the printable characters in JIS X 0212.
# Printable characters in udc or vdc classes whose code points are in
# the undefined area of JIS X 0212 may be added.
#

jisx0212 <J0215>;...;<J0225>;\
        <J0234>;...;<J0236>;\
        <J0275>;...;<J0281>;\
        <J0665>;...;<J0669>;\
        <J0671>;\
        <J0673>;\
        <J0674>;\
        <J0676>;\
        <J0681>;...;<J0692>;\
        <J0734>;...;<J0746>;\
        <J0782>;...;<J0794>;\
        <J0901>;\
        <J0902>;\
        <J0904>;\
        <J0906>;\
        <J0908>;\
        <J0909>;\
        <J0911>;...;<J0913>;\
        <J0915>;\
        <J0916>;\
        <J0933>;...;<J0948>;\
        <J1001>;...;<J1024>;\
        <J1026>;...;<J1087>;\
        <J1101>;...;<J1127>;\
        <J1129>;...;<J1135>;\
        <J1137>;...;<J1187>;\
        <J1601>;...;<J1694>;\
        <J1701>;...;<J1794>;\
        <J1801>;...;<J1894>;\
        <J1901>;...;<J1994>;\
        <J2001>;...;<J2094>;\
        <J2101>;...;<J2194>;\
        <J2201>;...;<J2294>;\
        <J2301>;...;<J2394>;\
```

```
<J2401>;...;<J2494>;\
<J2501>;...;<J2594>;\
<J2601>;...;<J2694>;\
<J2701>;...;<J2794>;\
<J2801>;...;<J2894>;\
<J2901>;...;<J2994>;\
<J3001>;...;<J3094>;\
<J3101>;...;<J3194>;\
<J3201>;...;<J3294>;\
<J3301>;...;<J3394>;\
<J3401>;...;<J3494>;\
<J3501>;...;<J3594>;\
<J3601>;...;<J3694>;\
<J3701>;...;<J3794>;\
<J3801>;...;<J3894>;\
<J3901>;...;<J3994>;\
<J4001>;...;<J4094>;\
<J4101>;...;<J4194>;\
<J4201>;...;<J4294>;\
<J4301>;...;<J4394>;\
<J4401>;...;<J4494>;\
<J4501>;...;<J4594>;\
<J4601>;...;<J4694>;\
<J4701>;...;<J4794>;\
<J4801>;...;<J4894>;\
<J4901>;...;<J4994>;\
<J5001>;...;<J5094>;\
<J5101>;...;<J5194>;\
<J5201>;...;<J5294>;\
<J5301>;...;<J5394>;\
<J5401>;...;<J5494>;\
<J5501>;...;<J5594>;\
<J5601>;...;<J5694>;\
<J5701>;...;<J5794>;\
<J5801>;...;<J5894>;\
<J5901>;...;<J5994>;\
<J6001>;...;<J6094>;\
<J6101>;...;<J6194>;\
<J6201>;...;<J6294>;\
<J6301>;...;<J6394>;\
<J6401>;...;<J6494>;\
<J6501>;...;<J6594>;\
<J6601>;...;<J6694>;\
<J6701>;...;<J6794>;\
<J6801>;...;<J6894>;\
<J6901>;...;<J6994>;\
<J7001>;...;<J7094>;\
<J7101>;...;<J7194>;\
<J7201>;...;<J7294>;\
<J7301>;...;<J7394>;\
<J7401>;...;<J7494>;\
<J7501>;...;<J7594>;\
```

```
        <J7601>;...;<J7694>;\
        <J7701>;...;<J7767>

#
# udc class: user defined characters
#

udc

#
# vdc class: vender defined characters
#

vdc

#
# gaiji class: udc or vdc
#

gaiji

#
# jhira class:
# The Hiragana characters in JIS X 0208.
# Hiragana characters in udc or vdc classes may be added.
#

jhira  <j0401>;...;<j0483>;\
       <j0111>;<j0112>;\
       <j0121>;<j0122>;<j0128>

#
# jkata class:
# The Katakana characters in JIS X 0208 JIS X 0201.
# The voiced, semivoiced and prolonged sound marks in JIS X 0208
# and JIS X 0201.
# The Katakana iteration marks in JIS X 0208.
# Katakana characters in udc or vdc classes may be added.
#

jkata  <kana-WO>;<kana-a>;<kana-i>;<kana-u>;<kana-e>;<kana-o>;\
       <kana-ya>;<kana-yu>;<kana-yo>;<kana-tsu>;\
       <kana-prolonged-sound>;\
       <kana-A>;<kana-I>;<kana-U>;<kana-E>;<kana-O>;\
       <kana-KA>;<kana-KI>;<kana-KU>;<kana-KE>;<kana-KO>;\
       <kana-SA>;<kana-SHI>;<kana-SU>;<kana-SE>;<kana-SO>;\
       <kana-TA>;<kana-CHI>;<kana-TSU>;<kana-TE>;<kana-TO>;\
       <kana-NA>;<kana-NI>;<kana-NU>;<kana-NE>;<kana-NO>;\
       <kana-HA>;<kana-HI>;<kana-FU>;<kana-HE>;<kana-HO>;\
       <kana-MA>;<kana-MI>;<kana-MU>;<kana-ME>;<kana-MO>;\
       <kana-YA>;<kana-YU>;<kana-YO>;<kana-RA>;<kana-RI>;\
       <kana-RU>;<kana-RE>;<kana-RO>;<kana-WA>;<kana-N>;\
```

```
        <kana-voiced-sound>;<kana-semivoiced-sound>;\
        <j0501>;...;<j0586>;\
        <j0111>;<j0112>;\
        <j0119>;<j0120>;<j0128>

#
# jhankana class:
# The Katakana characters in JIS X 0201.
# The voiced, semivoiced and prolonged sound marks in JIS X 0201.
# Katakana characters, Katakana symbols in JIS X 0201, or udc/vdc
# in undefined area of JIS X 0201 may be added.
#

jhankana  <kana-WO>;<kana-a>;<kana-i>;<kana-u>;<kana-e>;<kana-o>;\
          <kana-ya>;<kana-yu>;<kana-yo>;<kana-tsu>;\
          <kana-prolonged-sound>;\
          <kana-A>;<kana-I>;<kana-U>;<kana-E>;<kana-O>;\
          <kana-KA>;<kana-KI>;<kana-KU>;<kana-KE>;<kana-KO>;\
          <kana-SA>;<kana-SHI>;<kana-SU>;<kana-SE>;<kana-SO>;\
          <kana-TA>;<kana-CHI>;<kana-TSU>;<kana-TE>;<kana-TO>;\
          <kana-NA>;<kana-NI>;<kana-NU>;<kana-NE>;<kana-NO>;\
          <kana-HA>;<kana-HI>;<kana-FU>;<kana-HE>;<kana-HO>;\
          <kana-MA>;<kana-MI>;<kana-MU>;<kana-ME>;<kana-MO>;\
          <kana-YA>;<kana-YU>;<kana-YO>;<kana-RA>;<kana-RI>;\
          <kana-RU>;<kana-RE>;<kana-RO>;<kana-WA>;<kana-N>;\
          <kana-voiced-sound>;<kana-semivoiced-sound>

#
# jkanji class: Kanji (Ideograms)
# Kanji in JIS X 0208 and JIS X 0212.
# Kanji Iteration mark in JIS X 0208.
# Han-numeral zero in JIS X 0208.
# Kanji in udc or vdc classes may be added.
#

jkanji  <j1601>;...;<j1694>;\
        <j1701>;...;<j1794>;\
        <j1801>;...;<j1894>;\
        <j1901>;...;<j1994>;\
        <j2001>;...;<j2094>;\
        <j2101>;...;<j2194>;\
        <j2201>;...;<j2294>;\
        <j2301>;...;<j2394>;\
        <j2401>;...;<j2494>;\
        <j2501>;...;<j2594>;\
        <j2601>;...;<j2694>;\
        <j2701>;...;<j2794>;\
        <j2801>;...;<j2894>;\
        <j2901>;...;<j2994>;\
        <j3001>;...;<j3094>;\
        <j3101>;...;<j3194>;\
        <j3201>;...;<j3294>;\
```

```
<j3301>;...;<j3394>;\
<j3401>;...;<j3494>;\
<j3501>;...;<j3594>;\
<j3601>;...;<j3694>;\
<j3701>;...;<j3794>;\
<j3801>;...;<j3894>;\
<j3901>;...;<j3994>;\
<j4001>;...;<j4094>;\
<j4101>;...;<j4194>;\
<j4201>;...;<j4294>;\
<j4301>;...;<j4394>;\
<j4401>;...;<j4494>;\
<j4501>;...;<j4594>;\
<j4601>;...;<j4694>;\
<j4701>;...;<j4751>;\
<j4801>;...;<j4894>;\
<j4901>;...;<j4994>;\
<j5001>;...;<j5094>;\
<j5101>;...;<j5194>;\
<j5201>;...;<j5294>;\
<j5301>;...;<j5394>;\
<j5401>;...;<j5494>;\
<j5501>;...;<j5594>;\
<j5601>;...;<j5694>;\
<j5701>;...;<j5794>;\
<j5801>;...;<j5894>;\
<j5901>;...;<j5994>;\
<j6001>;...;<j6094>;\
<j6101>;...;<j6194>;\
<j6201>;...;<j6294>;\
<j6301>;...;<j6394>;\
<j6401>;...;<j6494>;\
<j6501>;...;<j6594>;\
<j6601>;...;<j6694>;\
<j6701>;...;<j6794>;\
<j6801>;...;<j6894>;\
<j6901>;...;<j6994>;\
<j7001>;...;<j7094>;\
<j7101>;...;<j7194>;\
<j7201>;...;<j7294>;\
<j7301>;...;<j7394>;\
<j7401>;...;<j7494>;\
<j7501>;...;<j7594>;\
<j7601>;...;<j7694>;\
<j7701>;...;<j7794>;\
<j7801>;...;<j7894>;\
<j7901>;...;<j7994>;\
<j8001>;...;<j8094>;\
<j8101>;...;<j8194>;\
<j8201>;...;<j8294>;\
<j8301>;...;<j8394>;\
<j8401>;...;<j8406>;\
```

```
<J1601>;...;<J1694>;\
<J1701>;...;<J1794>;\
<J1801>;...;<J1894>;\
<J1901>;...;<J1994>;\
<J2001>;...;<J2094>;\
<J2101>;...;<J2194>;\
<J2201>;...;<J2294>;\
<J2301>;...;<J2394>;\
<J2401>;...;<J2494>;\
<J2501>;...;<J2594>;\
<J2601>;...;<J2694>;\
<J2701>;...;<J2794>;\
<J2801>;...;<J2894>;\
<J2901>;...;<J2994>;\
<J3001>;...;<J3094>;\
<J3101>;...;<J3194>;\
<J3201>;...;<J3294>;\
<J3301>;...;<J3394>;\
<J3401>;...;<J3494>;\
<J3501>;...;<J3594>;\
<J3601>;...;<J3694>;\
<J3701>;...;<J3794>;\
<J3801>;...;<J3894>;\
<J3901>;...;<J3994>;\
<J4001>;...;<J4094>;\
<J4101>;...;<J4194>;\
<J4201>;...;<J4294>;\
<J4301>;...;<J4394>;\
<J4401>;...;<J4494>;\
<J4501>;...;<J4594>;\
<J4601>;...;<J4694>;\
<J4701>;...;<J4794>;\
<J4801>;...;<J4894>;\
<J4901>;...;<J4994>;\
<J5001>;...;<J5094>;\
<J5101>;...;<J5194>;\
<J5201>;...;<J5294>;\
<J5301>;...;<J5394>;\
<J5401>;...;<J5494>;\
<J5501>;...;<J5594>;\
<J5601>;...;<J5694>;\
<J5701>;...;<J5794>;\
<J5801>;...;<J5894>;\
<J5901>;...;<J5994>;\
<J6001>;...;<J6094>;\
<J6101>;...;<J6194>;\
<J6201>;...;<J6294>;\
<J6301>;...;<J6394>;\
<J6401>;...;<J6494>;\
<J6501>;...;<J6594>;\
<J6601>;...;<J6694>;\
<J6701>;...;<J6794>;\
```

```
        <J6801>;...;<J6894>;\
        <J6901>;...;<J6994>;\
        <J7001>;...;<J7094>;\
        <J7101>;...;<J7194>;\
        <J7201>;...;<J7294>;\
        <J7301>;...;<J7394>;\
        <J7401>;...;<J7494>;\
        <J7501>;...;<J7594>;\
        <J7601>;...;<J7694>;\
        <J7701>;...;<J7767>;\
        <j0124>;<j0125>;<j0127>

#
# jspace class: The space character in JIS X 0208
#

jspace  <j0101>

#
# toupper and tolower:  also handle Roman, Greek and Russian
#                       characters in JIS X 0208 and JIS X 0212

toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);\
        (<z>,<Z>);\
        (<j0365>,<j0333>);(<j0366>,<j0334>);(<j0367>,<j0335>);\
        (<j0368>,<j0336>);(<j0369>,<j0337>);(<j0370>,<j0338>);\
        (<j0371>,<j0339>);(<j0372>,<j0340>);(<j0373>,<j0341>);\
        (<j0374>,<j0342>);(<j0375>,<j0343>);(<j0376>,<j0344>);\
        (<j0377>,<j0345>);(<j0378>,<j0346>);(<j0379>,<j0347>);\
        (<j0380>,<j0348>);(<j0381>,<j0349>);(<j0382>,<j0350>);\
        (<j0383>,<j0351>);(<j0384>,<j0352>);(<j0385>,<j0353>);\
        (<j0386>,<j0354>);(<j0387>,<j0355>);(<j0388>,<j0356>);\
        (<j0389>,<j0357>);(<j0390>,<j0358>);\
        (<j0633>,<j0601>);(<j0634>,<j0602>);(<j0635>,<j0603>);\
        (<j0636>,<j0604>);(<j0637>,<j0605>);(<j0638>,<j0606>);\
        (<j0639>,<j0607>);(<j0640>,<j0608>);(<j0641>,<j0609>);\
        (<j0642>,<j0610>);(<j0643>,<j0611>);(<j0644>,<j0612>);\
        (<j0645>,<j0613>);(<j0646>,<j0614>);(<j0647>,<j0615>);\
        (<j0648>,<j0616>);(<j0649>,<j0617>);(<j0650>,<j0618>);\
        (<j0651>,<j0619>);(<j0652>,<j0620>);(<j0653>,<j0621>);\
        (<j0654>,<j0622>);(<j0655>,<j0623>);(<j0656>,<j0624>);\
        (<j0749>,<j0701>);(<j0750>,<j0702>);(<j0751>,<j0703>);\
        (<j0752>,<j0704>);(<j0753>,<j0705>);(<j0754>,<j0706>);\
        (<j0755>,<j0707>);(<j0756>,<j0708>);(<j0757>,<j0709>);\
        (<j0758>,<j0710>);(<j0759>,<j0711>);(<j0760>,<j0712>);\
        (<j0761>,<j0713>);(<j0762>,<j0714>);(<j0763>,<j0715>);\
        (<j0764>,<j0716>);(<j0765>,<j0717>);(<j0766>,<j0718>);\
        (<j0767>,<j0719>);(<j0768>,<j0720>);(<j0769>,<j0721>);\
```

```
        (<j0770>,<j0722>);(<j0771>,<j0723>);(<j0772>,<j0724>);\
        (<j0773>,<j0725>);(<j0774>,<j0726>);(<j0775>,<j0727>);\
        (<j0776>,<j0728>);(<j0777>,<j0729>);(<j0778>,<j0730>);\
        (<j0779>,<j0731>);(<j0780>,<j0732>);(<j0781>,<j0733>);\
        (<J0681>,<J0665>);(<J0682>,<J0666>);(<J0683>,<J0667>);\
        (<J0684>,<J0668>);(<J0685>,<J0669>);(<J0687>,<J0671>);\
        (<J0689>,<J0673>);(<J0690>,<J0674>);(<J0692>,<J0676>);\
        (<J0782>,<J0734>);(<J0783>,<J0735>);(<J0784>,<J0736>);\
        (<J0785>,<J0737>);(<J0786>,<J0738>);(<J0787>,<J0739>);\
        (<J0788>,<J0740>);(<J0789>,<J0741>);(<J0790>,<J0742>);\
        (<J0791>,<J0743>);(<J0792>,<J0744>);(<J0793>,<J0745>);\
        (<J0794>,<J0746>);(<J0933>,<J0901>);(<J0934>,<J0902>);\
        (<J0936>,<J0904>);(<J0938>,<J0906>);(<J0940>,<J0908>);\
        (<J0941>,<J0909>);(<J0943>,<J0911>);(<J0944>,<J0912>);\
        (<J0945>,<J0913>);(<J0947>,<J0915>);(<J0948>,<J0916>);\
        (<J1101>,<J1001>);(<J1102>,<J1002>);(<J1103>,<J1003>);\
        (<J1104>,<J1004>);(<J1105>,<J1005>);(<J1106>,<J1006>);\
        (<J1107>,<J1007>);(<J1108>,<J1008>);(<J1109>,<J1009>);\
        (<J1110>,<J1010>);(<J1111>,<J1011>);(<J1112>,<J1012>);\
        (<J1113>,<J1013>);(<J1114>,<J1014>);(<J1115>,<J1015>);\
        (<J1116>,<J1016>);(<J1117>,<J1017>);(<J1118>,<J1018>);\
        (<J1119>,<J1019>);(<J1120>,<J1020>);(<J1121>,<J1021>);\
        (<J1122>,<J1022>);(<J1123>,<J1023>);(<J1124>,<J1024>);\
        (<J1126>,<J1026>);(<J1127>,<J1027>);(<J1129>,<J1029>);\
        (<J1130>,<J1030>);(<J1131>,<J1031>);(<J1132>,<J1032>);\
        (<J1133>,<J1033>);(<J1134>,<J1034>);(<J1135>,<J1035>);\
        (<J1137>,<J1037>);(<J1138>,<J1038>);(<J1139>,<J1039>);\
        (<J1140>,<J1040>);(<J1141>,<J1041>);(<J1142>,<J1042>);\
        (<J1143>,<J1043>);(<J1144>,<J1044>);(<J1145>,<J1045>);\
        (<J1146>,<J1046>);(<J1147>,<J1047>);(<J1148>,<J1048>);\
        (<J1149>,<J1049>);(<J1150>,<J1050>);(<J1151>,<J1051>);\
        (<J1152>,<J1052>);(<J1153>,<J1053>);(<J1154>,<J1054>);\
        (<J1155>,<J1055>);(<J1156>,<J1056>);(<J1157>,<J1057>);\
        (<J1158>,<J1058>);(<J1159>,<J1059>);(<J1160>,<J1060>);\
        (<J1161>,<J1061>);(<J1162>,<J1062>);(<J1163>,<J1063>);\
        (<J1164>,<J1064>);(<J1165>,<J1065>);(<J1166>,<J1066>);\
        (<J1167>,<J1067>);(<J1168>,<J1068>);(<J1169>,<J1069>);\
        (<J1170>,<J1070>);(<J1171>,<J1071>);(<J1172>,<J1072>);\
        (<J1173>,<J1073>);(<J1174>,<J1074>);(<J1175>,<J1075>);\
        (<J1176>,<J1076>);(<J1177>,<J1077>);(<J1178>,<J1078>);\
        (<J1179>,<J1079>);(<J1180>,<J1080>);(<J1181>,<J1081>);\
        (<J1182>,<J1082>);(<J1183>,<J1083>);(<J1184>,<J1084>);\
        (<J1185>,<J1085>);(<J1186>,<J1086>);(<J1187>,<J1087>)

tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);\
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);\
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);\
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);\
        (<Z>,<z>);\
        (<j0333>,<j0365>);(<j0334>,<j0366>);(<j0335>,<j0367>);\
        (<j0336>,<j0368>);(<j0337>,<j0369>);(<j0338>,<j0370>);\
```

```
(<j0339>,<j0371>);(<j0340>,<j0372>);(<j0341>,<j0373>);\
(<j0342>,<j0374>);(<j0343>,<j0375>);(<j0344>,<j0376>);\
(<j0345>,<j0377>);(<j0346>,<j0378>);(<j0347>,<j0379>);\
(<j0348>,<j0380>);(<j0349>,<j0381>);(<j0350>,<j0382>);\
(<j0351>,<j0383>);(<j0352>,<j0384>);(<j0353>,<j0385>);\
(<j0354>,<j0386>);(<j0355>,<j0387>);(<j0356>,<j0388>);\
(<j0357>,<j0389>);(<j0358>,<j0390>);\
(<j0601>,<j0633>);(<j0602>,<j0634>);(<j0603>,<j0635>);\
(<j0604>,<j0636>);(<j0605>,<j0637>);(<j0606>,<j0638>);\
(<j0607>,<j0639>);(<j0608>,<j0640>);(<j0609>,<j0641>);\
(<j0610>,<j0642>);(<j0611>,<j0643>);(<j0612>,<j0644>);\
(<j0613>,<j0645>);(<j0614>,<j0646>);(<j0615>,<j0647>);\
(<j0616>,<j0648>);(<j0617>,<j0649>);(<j0618>,<j0650>);\
(<j0619>,<j0651>);(<j0620>,<j0652>);(<j0621>,<j0653>);\
(<j0622>,<j0654>);(<j0623>,<j0655>);(<j0624>,<j0656>);\
(<j0701>,<j0749>);(<j0702>,<j0750>);(<j0703>,<j0751>);\
(<j0704>,<j0752>);(<j0705>,<j0753>);(<j0706>,<j0754>);\
(<j0707>,<j0755>);(<j0708>,<j0756>);(<j0709>,<j0757>);\
(<j0710>,<j0758>);(<j0711>,<j0759>);(<j0712>,<j0760>);\
(<j0713>,<j0761>);(<j0714>,<j0762>);(<j0715>,<j0763>);\
(<j0716>,<j0764>);(<j0717>,<j0765>);(<j0718>,<j0766>);\
(<j0719>,<j0767>);(<j0720>,<j0768>);(<j0721>,<j0769>);\
(<j0722>,<j0770>);(<j0723>,<j0771>);(<j0724>,<j0772>);\
(<j0725>,<j0773>);(<j0726>,<j0774>);(<j0727>,<j0775>);\
(<j0728>,<j0776>);(<j0729>,<j0777>);(<j0730>,<j0778>);\
(<j0731>,<j0779>);(<j0732>,<j0780>);(<j0733>,<j0781>);\
(<J0665>,<J0681>);(<J0666>,<J0682>);(<J0667>,<J0683>);\
(<J0668>,<J0684>);(<J0669>,<J0685>);(<J0671>,<J0687>);\
(<J0673>,<J0689>);(<J0674>,<J0690>);(<J0676>,<J0692>);\
(<J0734>,<J0782>);(<J0735>,<J0783>);(<J0736>,<J0784>);\
(<J0737>,<J0785>);(<J0738>,<J0786>);(<J0739>,<J0787>);\
(<J0740>,<J0788>);(<J0741>,<J0789>);(<J0742>,<J0790>);\
(<J0743>,<J0791>);(<J0744>,<J0792>);(<J0745>,<J0793>);\
(<J0746>,<J0794>);(<J0901>,<J0933>);(<J0902>,<J0934>);\
(<J0904>,<J0936>);(<J0906>,<J0938>);(<J0908>,<J0940>);\
(<J0909>,<J0941>);(<J0911>,<J0943>);(<J0912>,<J0944>);\
(<J0913>,<J0945>);(<J0915>,<J0947>);(<J0916>,<J0948>);\
(<J1001>,<J1101>);(<J1002>,<J1102>);(<J1003>,<J1103>);\
(<J1004>,<J1104>);(<J1005>,<J1105>);(<J1006>,<J1106>);\
(<J1007>,<J1107>);(<J1008>,<J1108>);(<J1009>,<J1109>);\
(<J1010>,<J1110>);(<J1011>,<J1111>);(<J1012>,<J1112>);\
(<J1013>,<J1113>);(<J1014>,<J1114>);(<J1015>,<J1115>);\
(<J1016>,<J1116>);(<J1017>,<J1117>);(<J1018>,<J1118>);\
(<J1019>,<J1119>);(<J1020>,<J1120>);(<J1021>,<J1121>);\
(<J1022>,<J1122>);(<J1023>,<J1123>);(<J1024>,<J1124>);\
(<J1026>,<J1126>);(<J1027>,<J1127>);(<J1029>,<J1129>);\
(<J1030>,<J1130>);(<J1031>,<J1131>);(<J1032>,<J1132>);\
(<J1033>,<J1133>);(<J1034>,<J1134>);(<J1035>,<J1135>);\
(<J1037>,<J1137>);(<J1038>,<J1138>);(<J1039>,<J1139>);\
(<J1040>,<J1140>);(<J1041>,<J1141>);(<J1042>,<J1142>);\
(<J1043>,<J1143>);(<J1044>,<J1144>);(<J1045>,<J1145>);\
(<J1046>,<J1146>);(<J1047>,<J1147>);(<J1048>,<J1148>);\
```

```
        (<J1049>,<J1149>);(<J1050>,<J1150>);(<J1051>,<J1151>);\
        (<J1052>,<J1152>);(<J1053>,<J1153>);(<J1054>,<J1154>);\
        (<J1055>,<J1155>);(<J1056>,<J1156>);(<J1057>,<J1157>);\
        (<J1058>,<J1158>);(<J1059>,<J1159>);(<J1060>,<J1160>);\
        (<J1061>,<J1161>);(<J1062>,<J1162>);(<J1063>,<J1163>);\
        (<J1064>,<J1164>);(<J1065>,<J1165>);(<J1066>,<J1166>);\
        (<J1067>,<J1167>);(<J1068>,<J1168>);(<J1069>,<J1169>);\
        (<J1070>,<J1170>);(<J1071>,<J1171>);(<J1072>,<J1172>);\
        (<J1073>,<J1173>);(<J1074>,<J1174>);(<J1075>,<J1175>);\
        (<J1076>,<J1176>);(<J1077>,<J1177>);(<J1078>,<J1178>);\
        (<J1079>,<J1179>);(<J1080>,<J1180>);(<J1081>,<J1181>);\
        (<J1082>,<J1182>);(<J1083>,<J1183>);(<J1084>,<J1184>);\
        (<J1085>,<J1185>);(<J1086>,<J1186>);(<J1087>,<J1187>)

END LC_CTYPE


#
#       LC_COLLATE
#
LC_COLLATE

order_start     forward

#
#       ISO 646 IRV  or  JIS X 0201 Roman
#
<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
<CAN>
```

```
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen>
<period>
<slash>
<zero>
<one>
<two>
<three>
<four>
<five>
<six>
<seven>
<eight>
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<A>
<B>
<C>
<D>
<E>
<F>
<G>
<H>
<I>
<J>
<K>
<L>
```

```
<M>
<N>
<O>
<P>
<Q>
<R>
<S>
<T>
<U>
<V>
<W>
<X>
<Y>
<Z>
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a>
<b>
<c>
<d>
<e>
<f>
<g>
<h>
<i>
<j>
<k>
<l>
<m>
<n>
<o>
<p>
<q>
<r>
<s>
<t>
<u>
<v>
<w>
<x>
<y>
<z>
<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
<DEL>
```

```
#
#       C1 control
#
<BPH>
<NBH>
<NEL>
<SSA>
<ESA>
<HTS>
<HTJ>
<VTS>
<PLD>
<PLU>
<RI>
<SS2>
<SS3>
<DCS>
<PU1>
<PU2>
<STS>
<CCH>
<MW>
<SPA>
<EPA>
<SOS>
<SCI>
<CSI>
<ST>
<OSC>
<PM>
<APC>

#
#       JIS X 0201 Katakana
#
<kana-full-stop>
<kana-opening-bracket>
<kana-closing-bracket>
<kana-comma>
<kana-conjunctive>
<kana-WO>
<kana-a>
<kana-i>
<kana-u>
<kana-e>
<kana-o>
<kana-ya>
<kana-yu>
<kana-yo>
<kana-tsu>
<kana-prolonged-sound>
<kana-A>
```

```
<kana-I>
<kana-U>
<kana-E>
<kana-O>
<kana-KA>
<kana-KI>
<kana-KU>
<kana-KE>
<kana-KO>
<kana-SA>
<kana-SHI>
<kana-SU>
<kana-SE>
<kana-SO>
<kana-TA>
<kana-CHI>
<kana-TSU>
<kana-TE>
<kana-TO>
<kana-NA>
<kana-NI>
<kana-NU>
<kana-NE>
<kana-NO>
<kana-HA>
<kana-HI>
<kana-FU>
<kana-HE>
<kana-HO>
<kana-MA>
<kana-MI>
<kana-MU>
<kana-ME>
<kana-MO>
<kana-YA>
<kana-YU>
<kana-YO>
<kana-RA>
<kana-RI>
<kana-RU>
<kana-RE>
<kana-RO>
<kana-WA>
<kana-N>
<kana-voiced-sound>
<kana-semivoiced-sound>

#
#       JIS X 0208
#
<j0101>
<j0194>
```

```
<j0201>
<j0214>
<j0226>
<J0233>
<J0242>
<J0248>
<j0260>
<j0274>
<J0282>
<J0289>
<j0294>
<j0316>
<j0325>
<j0333>
<j0358>
<j0365>
<j0390>
<j0401>
<j0483>
<j0501>
<j0586>
<j0601>
<j0624>
<j0633>
<j0656>
<j0701>
<j0733>
<j0749>
<j0781>
<j0801>
<j0832>
<j1601>
<j1694>
<j1701>
<j1794>
<j1801>
<j1894>
<j1901>
<j1994>
<j2001>
<j2094>
<j2101>
<j2194>
<j2201>
<j2294>
<j2301>
<j2394>
<j2401>
<j2494>
<j2501>
<j2594>
<j2601>
```

```
<j2694>
<j2701>
<j2794>
<j2801>
<j2894>
<j2901>
<j2994>
<j3001>
<j3094>
<j3101>
<j3194>
<j3201>
<j3294>
<j3301>
<j3394>
<j3401>
<j3494>
<j3501>
<j3594>
<j3601>
<j3694>
<j3701>
<j3794>
<j3801>
<j3894>
<j3901>
<j3994>
<j4001>
<j4094>
<j4101>
<j4194>
<j4201>
<j4294>
<j4301>
<j4394>
<j4401>
<j4494>
<j4501>
<j4594>
<j4601>
<j4694>
<j4701>
<j4751>
<j4801>
<j4894>
<j4901>
<j4994>
<j5001>
<j5094>
<j5101>
<j5194>
<j5201>
```

```
<j5294>
<j5301>
<j5394>
<j5401>
<j5494>
<j5501>
<j5594>
<j5601>
<j5694>
<j5701>
<j5794>
<j5801>
<j5894>
<j5901>
<j5994>
<j6001>
<j6094>
<j6101>
<j6194>
<j6201>
<j6294>
<j6301>
<j6394>
<j6401>
<j6494>
<j6501>
<j6594>
<j6601>
<j6694>
<j6701>
<j6794>
<j6801>
<j6894>
<j6901>
<j6994>
<j7001>
<j7094>
<j7101>
<j7194>
<j7201>
<j7294>
<j7301>
<j7394>
<j7401>
<j7494>
<j7501>
<j7594>
<j7601>
<j7694>
<j7701>
<j7794>
<j7801>
```

```
<j7894>
<j7901>
<j7994>
<j8001>
<j8094>
<j8101>
<j8194>
<j8201>
<j8294>
<j8301>
<j8394>
<j8401>
<j8406>

#
#       JIS X 0212
#
<J0215>
<J0225>
<J0234>
<J0236>
<J0275>
<J0281>
<J0665>
<J0669>
<J0671>
<J0673>
<J0674>
<J0676>
<J0681>
<J0692>
<J0734>
<J0746>
<J0782>
<J0794>
<J0901>
<J0902>
<J0904>
<J0906>
<J0908>
<J0909>
<J0911>
<J0913>
<J0915>
<J0916>
<J0933>
<J0948>
<J1001>
<J1024>
<J1026>
<J1087>
<J1101>
```

```
<J1127>
<J1129>
<J1135>
<J1137>
<J1187>
<J1601>
<J1694>
<J1701>
<J1794>
<J1801>
<J1894>
<J1901>
<J1994>
<J2001>
<J2094>
<J2101>
<J2194>
<J2201>
<J2294>
<J2301>
<J2394>
<J2401>
<J2494>
<J2501>
<J2594>
<J2601>
<J2694>
<J2701>
<J2794>
<J2801>
<J2894>
<J2901>
<J2994>
<J3001>
<J3094>
<J3101>
<J3194>
<J3201>
<J3294>
<J3301>
<J3394>
<J3401>
<J3494>
<J3501>
<J3594>
<J3601>
<J3694>
<J3701>
<J3794>
<J3801>
<J3894>
<J3901>
```

```
<J3994>
<J4001>
<J4094>
<J4101>
<J4194>
<J4201>
<J4294>
<J4301>
<J4394>
<J4401>
<J4494>
<J4501>
<J4594>
<J4601>
<J4694>
<J4701>
<J4794>
<J4801>
<J4894>
<J4901>
<J4994>
<J5001>
<J5094>
<J5101>
<J5194>
<J5201>
<J5294>
<J5301>
<J5394>
<J5401>
<J5494>
<J5501>
<J5594>
<J5601>
<J5694>
<J5701>
<J5794>
<J5801>
<J5894>
<J5901>
<J5994>
<J6001>
<J6094>
<J6101>
<J6194>
<J6201>
<J6294>
<J6301>
<J6394>
<J6401>
<J6494>
<J6501>
```

```
<J6594>
<J6601>
<J6694>
<J6701>
<J6794>
<J6801>
<J6894>
<J6901>
<J6994>
<J7001>
<J7094>
<J7101>
<J7194>
<J7201>
<J7294>
<J7301>
<J7394>
<J7401>
<J7494>
<J7501>
<J7594>
<J7601>
<J7694>
<J7701>
<J7767>

#
#       Undefined
#
UNDEFINED

order_end

#
END LC_COLLATE


#
#       LC_MESSAGES
#
LC_MESSAGES

yesexpr "^[<y><Y><j0389><j0357>]"
noexpr  "^[<n><N><j0378><j0346>]"

#
END LC_MESSAGES


#
#       LC_MONETARY
#
```

```
LC_MONETARY

int_curr_symbol         "<J><P><Y><space>"
currency_symbol         "<yen-sign>"
mon_decimal_point       ""
mon_thousands_sep       "<comma>"
mon_grouping            3
positive_sign           ""
negative_sign           "<hyphen>"
int_frac_digits         0
frac_digits             0
p_cs_precedes           1
p_sep_by_space          0
n_cs_precedes           1
n_sep_by_space          0
p_sign_posn             1
n_sign_posn             4

#
END LC_MONETARY


#
#       LC_NUMERIC
#
LC_NUMERIC

decimal_point   "<period>"
thousands_sep   "<comma>"
grouping        3

#
END LC_NUMERIC


#
#       LC_TIME
#
LC_TIME

# abday: abbreviated weekday names
# abday is defined as the first letters of Japanese weekday names
# in Kanji, such as Nichi, Getsu, and Ka.
abday           "<j3892>";"<j2378>";"<j1848>";"<j3169>";\
                "<j4458>";"<j2266>";"<j3758>"

# day: full weekday names
# day is defined as full names of Japanese weekday names in Kanji,
# such as Nichiyoubi, Getsuyoubi, and Kayoubi.
day        "<j3892><j4543><j3892>";"<j2378><j4543><j3892>";\
           "<j1848><j4543><j3892>";"<j3169><j4543><j3892>";\
           "<j4458><j4543><j3892>";"<j2266><j4543><j3892>";\
```

```
            "<j3758><j4543><j3892>"

# abmon: abbreviated month names
# abmon is defined as two columns digit of month number, and Gatsu
# in Kanji.  If month number is less than ten, leading space is padded.
# The names are " 1Gatsu", " 2Gatsu", ..., "12Gatsu".
abmon       "<space><one>OP;<j2378>";"<space><two><j2378>";\
            "<space><three><j2378>";"<space><four><j2378>";\
            "<space><five><j2378>";"<space><six><j2378>";\
            "<space><seven><j2378>";"<space><eight><j2378>";\
            "<space><nine><j2378>";\
            "<one><zero><j2378>";"<one><one><j2378>";\
            "<one><two><j2378>"

# mon: full month names
# mon is defined as digit of month number, and Gatsu in Kanji.  No
# space is padded.  The names are "1Gatsu", "2Gatsu", ..., "12Gatsu".
mon         "<one><j2378>";"<two><j2378>";"<three><j2378>";\
            "<four><j2378>";"<five><j2378>";"<six><j2378>";\
            "<seven><j2378>";"<eight><j2378>";"<nine><j2378>";\
            "<one><zero><j2378>";"<one><one><j2378>";\
            "<one><two><j2378>"

# d_t_fmt: date and time format
# The format is defined as:
#       "%YNen%mGatsu%dNichi %HJi%MFun%SByou"
# which will be formatted as, for example,
#       "1993Nen02Gatsu06Nichi 08Ji59Fun07Byou"
d_t_fmt "%Y<j3915>%m<j2378>%d<j3892><space>%H<j2794>%M<j4212>%S<j4135>"

# d_fmt: date format
# The format is defined as:
#       "%YNen%mGatsu%dNichi"
# which will be formatted as, for example,
#       "1993Nen02Gatsu06Nichi"
d_fmt           "%Y<j3915>%m<j2378>%d<j3892>"

# t_fmt: time format
# The format is defined as:
#       "%HJi%MFun%SByou"
# which will be formatted as, for example,
#       "08Ji59Fun07Byou"
t_fmt           "%H<j2794>%M<j4212>%S<j4135>"

# am_pm: ante meridiem (AM) and post meridiem (PM) strings
# "Gozen" for AM, "Gogo" for PM
am_pm           "<j2465><j3316>";"<j2465><j2469>"

# t_fmt_ampm: time format using am_pm
# The format is defined as:
#       "%p%IJi%MFun%SByou"
# which will be formatted as, for example,
```

```
#       "Gozen08Ji59Fun07Byou"
t_fmt_ampm      "%p%I<j2794>%M<j4212>%S<j4135>"

# era: year count and format for era
# era (Gengou) is defined as follows:
#   from 1990-01-01:             "Heisei%EyNen" (1990 is Heisei 2 Nen.)
#   from 1989-01-08 to 1989-12-31: "HeiseiGannen"
#   from 1927-01-01 to 1989-01-07: "Showa%EyNen" (1927 is Showa 2 Nen.)
era     "+:2:1990/01/01:+*:<j4231><j3214>:%EC%Ey<j3915>";\
        "+:1:1989/01/08:1989/12/31:<j4231><j3214>:%EC<j2421><j3915>";\
        "+:2:1927/01/01:1989/01/07:<j3028><j4734>:%EC%Ey<j3915>"

# era_d_fmt: date format using era
# The format is defined as:
#       "%EY%mGatsu%dNichi"
# which will be formatted as, for example,
#       "Heisei5Nen02Gatsu06Nichi"
era_d_fmt       "%EY%m<j2378>%d<j3892>"

# era_d_t_fmt: date and time format using era
# The format is defined as:
#       "%EY%mGatsu%dNichi %HJi%MFun%SByou"
# which will be formatted as, for example,
#       "Heisei5Nen02Gatsu06Nichi 08Ji59Fun07Byou"
era_d_t_fmt  "%EY%m<j2378>%d<j3892><space>%H<j2794>%M<j4212>%S<j4135>"

# era_t_fmt and alt_digits are not defined here.

#
END LC_TIME
```

# Glossary

**base character**
One of the set of characters defined in the Latin alphabet. In Western European languages other than English, these characters are commonly used with diacritical marks (accents, cedilla, and so forth) to extend the range of characters in an alphabet.

**byte**
An individually addressable unit of data storage that is equal to or larger than an octet, used to store a character or a portion of a character; see **character**. A byte is composed of a contiguous sequence of bits, the number of which is implementation-dependent. The least significant bit is called the *low-order* bit; the most significant is called the *high-order* bit. Note that this definition of *byte* deviates intentionally from the usage of *byte* in some international standards, where it is used as a synonym for *octet* (always eight bits). On a system based on the ISO POSIX-2 DIS, a byte may be larger than eight bits so that it can be an integral portion of larger data objects that are not evenly divisible by eight bits (such as a 36-bit word that contains four 9-bit bytes).

**character**
A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of a multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed.

**character array**
An array of type **char**.

**character class**
A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale.

**character set**
A finite set of different characters used for the representation, organisation or control of data.

**character string**
A contiguous sequence of characters terminated by and including the first null byte.

**coded character set**
A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation.

**collating element**
The smallest entity used to determine the logical ordering of character or wide character strings. See **collation sequence** on page 174. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements.

**collation**
The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements.

**collation sequence**
The relative order of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

Multi-level sorting is accomplished by assigning elements one or more collation weights, up to the limit {COLL_WEIGHTS_MAX}; see **<limits.h>** in the **XSH** specification. On each level, elements may be given the same weight (at the primary level, called an equivalence class; see **equivalence class**) or be omitted from the sequence. Strings that collate equal using the first assigned weight (primary ordering) are then compared using the next assigned weight (secondary ordering), and so on.

**composite graphic symbol**
A graphic symbol consisting of a combination of two or more other graphic symbols in a single character position, such as a diacritical mark and a basic letter.

**control character**
A character, other than a graphic character, that affects the recording, processing, transmission or interpretation of text.

**conversion descriptor**
A per-process unique value used to identify an open codeset conversion.

**downshifting**
The conversion of an upper-case character to its lower-case representation.

**eight-bit transparency**
The ability of a software component to process 8-bit characters without modifying or utilising any part of the character in a way that is inconsistent with the rules of the current coded character set.

**empty string**
A string whose first byte is a null byte.

**empty wide character string**
A wide-character string whose first element is a null wide-character code.

**equivalence class**
A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight.

**era**
An alternative method for counting and displaying years.

**graphic character**
A character, other than a control character, that has a visual representation when handwritten, printed or displayed.

**internationalisation**
The provision within a computer program of the capability of making itself adaptable to the requirements of different native languages, local customs and coded character sets.

**local customs**
The conventions of a geographical area or territory for such things as date, time and currency formats.

**locale**
The definition of the subset of a user's environment that depends on language and cultural conventions.

**localisation**
The process of establishing information within a computer system specific to the operation of particular languages, local customs and coded character sets.

**message catalogue**
A file or storage area containing program messages, command prompts and responses to prompts for a particular native language, territory and codeset.

**message catalogue descriptor**
A per-process unique value used to identify an open message catalogue. A message catalogue descriptor may be implemented using a file descriptor.

**multi-character collating element**
A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements **ch** and **ll**.

**native language**
A computer user's spoken or written language, such as American English, British English, Danish, Dutch, French, German, Italian, Japanese, Norwegian or Swedish.

**non-spacing characters**
A character, such as a character representing a diacritical mark in the ISO 6937:1983 standard coded character set, which is used in combination with other characters to form composite graphic symbols.

**null byte**
A byte with all bits set to zero.

**null pointer**
The value that is obtained by converting the number 0 into a pointer; for example, (**void** *) 0. The C language guarantees that this value does not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

**null string**
See **empty string** on page 174.

**null wide-character code**
A wide-character code with all bits set to zero.

**portable character set**
The collection of characters that are required to be present in all locales supported by X/Open-compliant systems:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ! # % ^ & * ( ) _ + - = { } [ ]
: " ~ ; ' ` < > ? , . | \ / @ $
```

Also included are <alert>, <backspace>, <tab>, <newline>, <vertical-tab>, <form-feed>, <carriage-return>, <space> and the null character, NUL.

This term is contrasted with the smaller *portable filename character set*.

**portable filename character set**
The set of characters from which portable filenames are constructed. For a filename to be portable across implementations conforming to this document set and the ISO POSIX-1 standard, it must consist only of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore and hyphen characters, respectively. The hyphen must not be used as the first character of a portable filename. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable pathname, the slash character may also be used.

**printable character**
One of the characters included in the **print** character classification of the LC_CTYPE category in the current locale.

**string**
A contiguous sequence of bytes terminated by and including the first null byte.

**upshifting**
The conversion of a lower-case character to its upper-case representation.

**white space**
A sequence of one or more characters that belong to the **space** character class as defined by the LC_CTYPE category in the current locale.

In the POSIX locale, white space consists of one or more blank characters (space and tab characters), newline characters, carriage-return characters, form-feed characters and vertical-tab characters.

**wide-character code**
An integer value corresponding to a single graphic symbol or control code.

**wide-character string**
A contiguous sequence of wide-character codes terminated by and including the first null wide-character code.

# Index