

*INTERACTIVE*

---

*product family*



*SunSoft*  
A Sun Microsystems, Inc. Business

## **First printing (October 1991)**

No part of this manual may be reproduced in any form or by any means without written permission of:

INTERACTIVE Systems Corporation  
2401 Colorado Avenue  
Santa Monica, California 90404

© Copyright INTERACTIVE Systems Corporation 1985-1991

© Copyright AT&T Corporation 1987-1988

© Copyright X/Open Company Limited 1989

### **RESTRICTED RIGHTS:**

For non-U.S. Government use:

These programs are supplied under a license. They may be used, disclosed, and/or copied only as permitted under such license agreement. Any copy must contain the above copyright notice and this restricted rights notice. Use, copying, and/or disclosure of the programs is strictly prohibited unless otherwise provided in the license agreement.

For U.S. Government use:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR Section 52.227-14 (Alternate III) or subparagraph (c)(1)(ii) of the clause at DFARS 252.227-7013, Rights in Technical Data and Computer Software.

All rights reserved. Printed in the U.S.A.

The following trademarks shown as registered are registered in the United States and other countries:

TEN/PLUS is a registered trademark of INTERACTIVE Systems Corporation.

VP/ix is a trademark of INTERACTIVE Systems Corporation.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Adobe is a registered trademark of Adobe Systems Incorporated.

DEC and VT220 are trademarks of Digital Equipment Corporation.

386 and 486 are trademarks of Intel Corporation.

AT and IBM are registered trademarks of International Business Machines Corporation.

PC/XT is a trademark of International Business Machines Corporation.

MS-DOS is a registered trademark of Microsoft Corporation.

SunRiver is a registered trademark of SunRiver Corporation.

X/Open is a trademark of X/Open Company Limited.

# **International Supplement Guide**

## **CONTENTS**

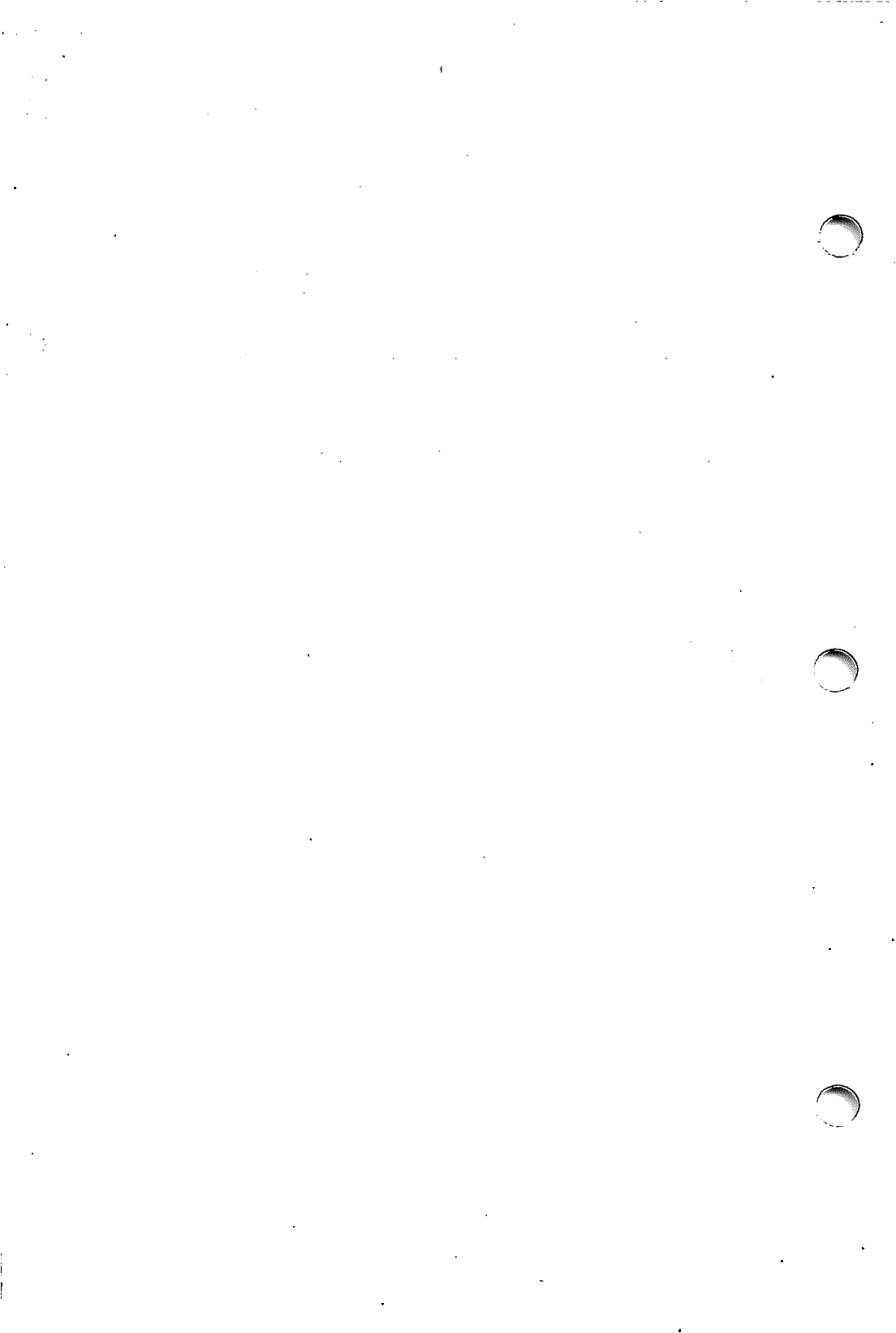
**International Supplement Overview and Installation Instructions**

**International Supplement User's Manual**

**International Supplement Manual for Advanced Users**

**X/Open Conformance Statement – Questionnaire**

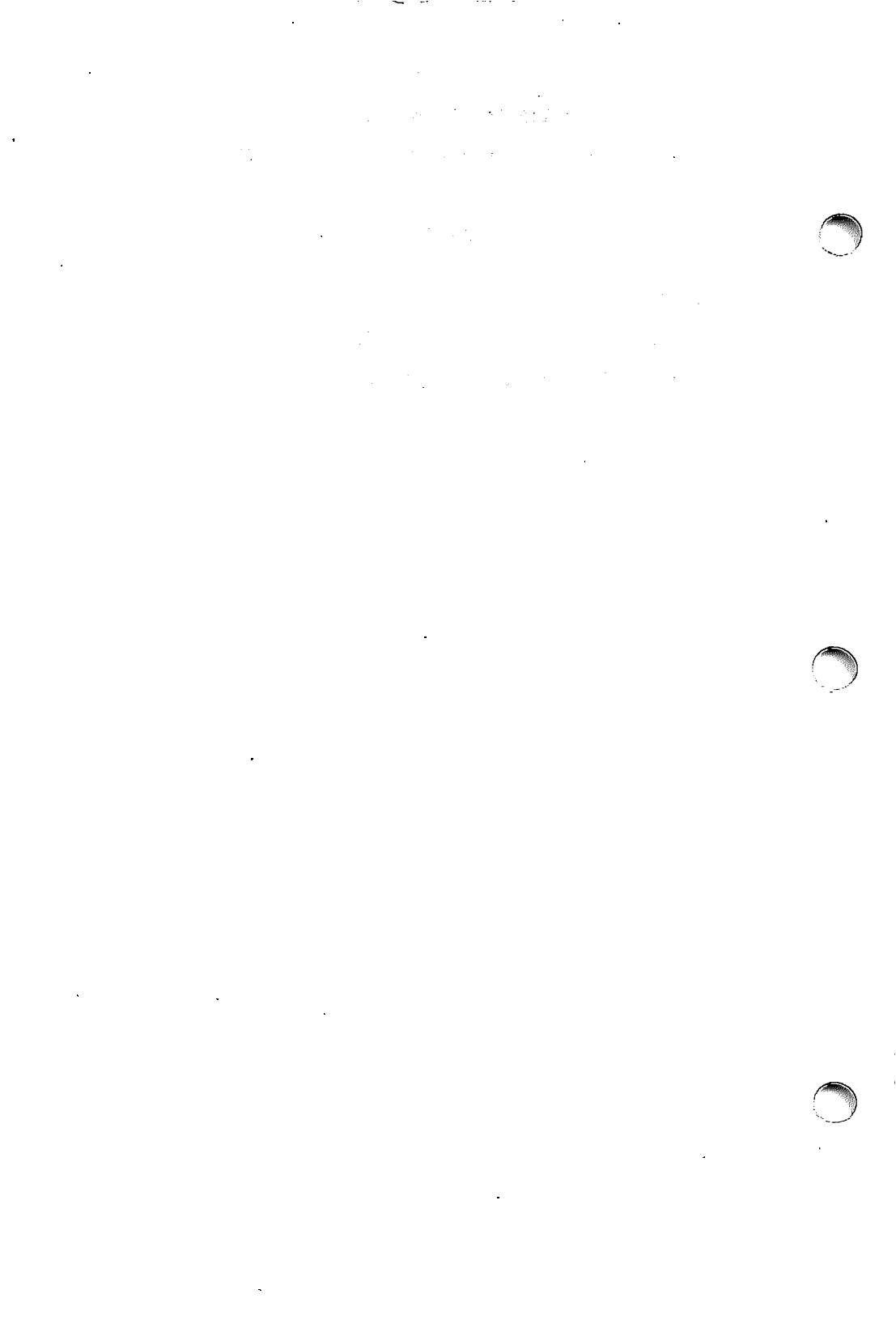
**International Supplement Reference Manual**



# **International Supplement Overview and Installation Instructions**

## **CONTENTS**

1. OVERVIEW . . . . .	1
2. INSTALLATION INSTRUCTIONS . . . . .	3
3. DOCUMENTATION REFERENCES . . . . .	4



# International Supplement

## Overview and Installation Instructions

### 1. OVERVIEW

INTERACTIVE's International Supplement extends the INTERACTIVE UNIX\* System V/386 Release 3.2 Operating System for use in an international environment. It allows software vendors to develop their applications in such a way that the text of one single application can be displayed in a different language, depending on the environment in which it is executed; a separate copy of the application for each language is *not* required.

The International Supplement contains internationalised versions of the most popular UNIX System utilities, such as `date`, `sort`, and `ls`. When using these utilities, users see the date displayed in their own language and can sort text files using the dictionary order of any supported language they specify.

The International Supplement also adds to the INTERACTIVE UNIX Operating System the functionality needed to make it fully compliant with X/Open\* Company Limited's Issue 3 of the *X/Open Portability Guide* (XPG3) (available from Prentice Hall). This guide contains practical standards for application portability, as adopted by X/Open Company Limited. This international group of hardware manufacturers and software vendors has defined a Common Applications Environment (CAE) that is built on the interfaces to the UNIX Operating System. Compliance with this CAE is now a requirement when systems are offered to most governments and corporations.

The *International Supplement Guide* includes:

- **International Supplement Overview and Installation Instructions**  
Provides a general overview of this guide, information about installation requirements, and references and conventions used.
- **International Supplement User's Manual**  
Provides a comprehensive description of how the INTERACTIVE UNIX System can be used in non-U.S. environments. Among other things, it discusses how to use different keyboards and how to correctly use UNIX System utilities.

- **International Supplement Manual for Advanced Users**

This manual is intended for system administrators, programmers, and other advanced users. It describes how to set up a user's international environment to correctly enter data on the keyboard, use UNIX System utilities, and run internationalised applications. It describes the format of collation tables and character classification tables and tells how they should be installed. It also gives a brief overview of the facilities that need to be added to a C source program to give the resulting application internationalised capabilities.

- **X/Open Conformance Statement – Questionnaire**

Provides the information required to describe the conformance of the INTERACTIVE UNIX Operating System with X/Open Company Limited's Issue 3 of the *X/Open Portability Guide*.

- **International Supplement Reference Manual**

Includes most of the relevant utilities and new library routines referred to in this guide. Although many of these entries are also present in the documentation for the INTERACTIVE UNIX Operating System, users and system administrators can now generally find them in one centralised place. Manual entries for the internationalised versions of UNIX System commands can be found in Volume 1 of the *X/Open Portability Guide*, Issue 3.



## 2. INSTALLATION INSTRUCTIONS

The International Supplement is installed using `sysadm installpkg` in the same manner as other INTERACTIVE subsets or extensions.

- For information about installing optional subsets, refer to section 6.1 of the “INTERACTIVE UNIX Operating System Installation Instructions” in the *INTERACTIVE UNIX Operating System Guide*.
- For information about using `sysadm`, refer to sections 2 and 3 of the “INTERACTIVE UNIX Operating System Maintenance Procedures” in the *INTERACTIVE UNIX Operating System Guide*.

After you have installed the International Supplement, your INTERACTIVE UNIX System will contain internationalised versions of several UNIX System commands, such as `date` and `who`. These are installed in the standard UNIX System directories where they belong, for example, `/bin` and `/usr/bin`. Copies of the original binaries can be found in a subdirectory of the original directory called `.sysv`, for example, `/bin/.sysv` and `/usr/bin/.sysv`. Refer to section 10 of the “International Supplement User’s Manual” for a list of the internationalised commands and functionality.

In addition to the commands specified by XPG3, INTERACTIVE has added the `colldef` and `showcat` commands. Refer to `colldef(1P)` and `showcat(1P)` for more information. The supplement also contains sample files for locales, message catalogues, and charmap files (the latter are used by `iconv(1P)` and `colldef(1P)`). locales are installed in the directory `/lib/locale/ISC`. Where appropriate, source files for these locales are located in `/lib/locale/ISC/localename/src`. The default message catalogue location is `/lib/locale/ISC/msgcat`. The `libc.cat` message catalogue contains the English language version of the error messages displayed by the library routines `perror(3P)` and `strerror(3P)`. `/lib/locale/ISC/msgcat/src/libc.msg` is the source file; it can be translated into other languages, which can then be used to generate alternate message catalogues for use by those routines.

A subset of contributed data files containing additional `locales`, keyboard mapping files, and so on, is also supplied. Some of these files have been contributed by third parties. All of these files are supplied “as-is” and are not supported.

### 3. DOCUMENTATION REFERENCES

Throughout this guide, the following full documentation titles will be referenced in shortened versions as follows:

<i>Full Title</i>	<i>Shortened Version</i>
INTERACTIVE UNIX System V/386 Release 3.2 Operating System Guide	INTERACTIVE UNIX Operating System Guide
INTERACTIVE UNIX System V/386 Release 3.2 User's/System Administrator's Reference Manual	INTERACTIVE UNIX System User's/System Administrator's Reference Manual
INTERACTIVE Software Development System Guide and Programmer's Reference Manual	INTERACTIVE SDS Guide and Programmer's Reference Manual

References of the form *name(n)* refer to an entry called *name* in section *n* of the reference manual or manual entries associated with that product or as stated in the documentation. Manual entries referred to in this guide may be found in either the “International Supplement Reference Manual” in this guide, the *INTERACTIVE SDS Guide and Programmer's Reference Manual* that accompanied your INTERACTIVE Software Development System (make special note of *ctime(3P)*, *perror(3P)*, *printf(3P)*, *scanf(3P)*, *environ(5P)*, and *regexp5P*), or the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual* that accompanied your INTERACTIVE UNIX Operating System.

# International Supplement User's Manual

## CONTENTS

1. INTRODUCTION . . . . .	1
2. INTERNATIONALISATION . . . . .	2
3. THE X/OPEN PORTABILITY GUIDE . . . . .	3
3.1 Computer Applications and Portability . . . . .	3
3.2 Standardisation and the Portability Guide . . . . .	3
3.3 Common Applications Environment . . . . .	4
3.4 Standard Portable Operating System Interface (POSIX.1) . . . . .	4
3.5 POSIX.2 . . . . .	5
3.6 The INTERACTIVE UNIX Operating System . . . . .	5
4. ENTERING DATA . . . . .	7
4.1 U.S. Personal Computer Keyboard Layout . . . . .	8
4.2 Generating Characters Not Present on a U.S. Keyboard . . . . .	9
4.2.1 Deadkeys . . . . .	9
4.2.2 Composing Characters Using Compose Sequences . . . . .	10
4.2.3 Decimal Representation . . . . .	11
4.2.4 Smiling Faces . . . . .	11
4.3 European Personal Computer Keyboard Layouts . . . . .	11
4.4 Cyrillic or Greek Keyboards . . . . .	14
4.5 Keyboard Layouts on 7-bit Terminals . . . . .	17
4.6 Using the VP/ix Environment . . . . .	17
4.7 Entering Data and Using INTERACTIVE X11 . . . . .	18
5. STORING DATA IN THE COMPUTER . . . . .	19
5.1 ASCII . . . . .	19
5.2 8-bit Characters and Codesets . . . . .	20
5.3 IBM Codepages . . . . .	22
5.4 ISO Codesets . . . . .	23
5.5 7-bit Codesets . . . . .	24

5.6	Choosing and Configuring a Codeset . . . . .	24
5.6.1	Converting From One Codeset to Another . . . . .	25
6.	DISPLAYING DATA . . . . .	26
6.1	7-bit Terminals . . . . .	26
6.2	The Console . . . . .	27
6.3	Displaying Data and Using INTERACTIVE X11 . . . . .	27
7.	THE INTERNATIONAL ENVIRONMENT . . . . .	28
7.1	The International Environment . . . . .	28
7.2	Controlling the International Environment . . . . .	30
8.	INTERNATIONALISED BEHAVIOUR . . . . .	32
8.1	Date and Time Format . . . . .	32
8.2	Character Classification . . . . .	33
8.3	Collation . . . . .	34
8.3.1	An Example . . . . .	34
8.4	Numeric and Monetary Formatting . . . . .	35
8.5	Yes/No Responses . . . . .	35
8.6	Message Catalogues . . . . .	36
8.7	The X/Open Environment . . . . .	36
9.	THE SYSTEM V ENVIRONMENT . . . . .	37
9.1	Date and Time Formats . . . . .	37
9.2	Character Classification . . . . .	37
10.	INTERNATIONALISED INTERACTIVE UNIX SYSTEM UTILITIES . . . . .	39
	GLOSSARY . . . . .	43

# International Supplement User's Manual

## 1. INTRODUCTION

This document explains the *internationalisation* features of the INTERACTIVE UNIX\* Operating System and describes how to use it on computer systems outside the United States (U.S.), where there are differences in local language, customs, and standards. This document focuses on usability and is restricted to those areas where languages are spoken that use an alphabet that contains fewer than one hundred letters. Korean, Japanese, Chinese, and other languages with thousands of different letters are not supported by the standard INTERACTIVE UNIX Operating System. In certain countries, INTERACTIVE's distributors sell a special version of the product to accommodate these special markets. Contact your sales representative for more information.

To find out how to set up a user to use the system in an international environment, refer to the "International Supplement Manual for Advanced Users."

## 2. INTERNATIONALISATION

Computers and their method of operation have generally been associated with American English. Until recently, computer users and programmers accepted the fact that operating and programming a computer had to be in English.

Internationalisation is the art of making a computer, a computer system, or a computer program (often called an application) function in a non-U.S. environment. The word itself illustrates that the different behaviour a computer system must support not only depends on the use of a different language, but also on the country of origin, even if the language is the same. Spelling may be different, for example – in American English the word is spelled internationalization, while in England the spelling is internationalisation. To avoid the spelling problem, the acronym *I18N* is becoming common (whether in the U.S. or England, internationalisation begins with the letter I, ends with N, and has 18 letters in between).

When the word internationalisation is brought up in a conversation, people often react with comments such as: “Oh. You are going to have a French or German version of your product, as well.” But *I18N* does not refer to the *translation* of software, but rather to its usability and translatability. An internationalised application or computer system is one that can be adapted to different environments without needing modification. The term *localisation* (and its acronym *L10N*) is used to describe the adaptation of computer programs to a single language and/or country, which, if mismanaged, can be as costly as making a separate version for each language.

### 3. THE X/OPEN PORTABILITY GUIDE

The term X/Open\* is often associated with standards. X/Open is a trade name, as well as trademark, of X/Open Company Limited. This organisation started as a consortium of European computer manufacturers (Bull, ICL, Siemens, Olivetti, Nixdorf, and Philips) whose principal aim is to increase the volume of applications available on their computer systems. In parallel, they have attempted to maximize the return on investments in software development made by users and independent software vendors (*ISVs*). Today, almost all major computer manufacturers are members of the X/Open group.

#### 3.1 Computer Applications and Portability

In the sixties, most computer applications were developed on and for a single proprietary computer system. In order to make the same application run on a different computer system, it had to be completely rewritten, usually in a different computer language. In the late seventies and early eighties, with the advent of the UNIX Operating System, this situation changed dramatically. This very portable operating system became available on a variety of hardware and supported a common new language – C.

There was still room for improvement, however. Most implementations of the UNIX Operating System were actually different flavors with different features. The C programming language by itself was simply a definition of a language. Supplying libraries with functions like `printf`, which software developers could immediately use in their programs, was the responsibility of compiler vendors. As far as interfacing with terminals and databases, there was no standard at all. As a result, despite the UNIX Operating System, porting applications (modifying the source program of an application to make it work on a different computer system) required a lot of effort and experienced programmers. Porting became a separate skill.

#### 3.2 Standardisation and the Portability Guide

Many standards committees, as well as AT&T (the developer of the UNIX System), tried to achieve a higher level of standardisation and compatibility. AT&T published the first issue of their System V Interface Definition (*SVID*), describing all the features of the UNIX Operating System that would be maintained, new ones that would be introduced, and old ones that would disappear in the next release.

This was a step in the right direction, but it was incomplete because it described only the interfaces to the operating system.

In 1985, the X/Open Company Limited published the *X/Open Portability Guide (XPG)*. It basically listed the SVID as its first chapters, but also included a description of the C language, the COBOL language, how to interface with databases, and other information. It is important to note that the X/Open Company always *adopted* standards where they existed, as opposed to *creating* new ones. Where standards were missing (for example, for internationalisation), they recommended standards.

### 3.3 Common Applications Environment

Now, more than five years later, the third issue of the *X/Open Portability Guide (XPG3)* is accepted by most governments and major corporations as the “bible” of the computer industry. Published in 1989, it consists of seven volumes describing the Common Applications Environment (*CAE*) defined by the X/Open Company and built on top of the interfaces of the UNIX Operating System, covering other aspects required for a comprehensive applications interface. The portion that discusses the operating system and its utilities is referred to as the X/Open System Interface (*XSI*). The seven volumes are:

- *XSI Commands and Utilities*
- *XSI System Interfaces and Headers*
- *XSI Supplementary Definitions*
- *Programming Languages*
- *Data Management*
- *Window Management*
- *Networking Services*

### 3.4 Standard Portable Operating System Interface (POSIX.1)

Volume 2 of the *X/Open Portability Guide, XSI System Interfaces and Headers*, is a superset of the *POSIX.1 Standard* published by the Institute of Electrical and Electronics Engineers, Inc. (*IEEE*). *POSIX.1* stands for the Standard Portable Operating System Interface for Computer Environments. This standard defines a standard operating system interface and environment based on the UNIX Operating System documentation to support application portability



at the source level. This is the first of a group of proposed standards known colloquially and collectively as POSIX. It is a superset of the system interfaces of the UNIX Operating System. XSI also adds a number of interfaces, particularly in the area of internationalisation, which go beyond both the SVID and POSIX.1.

### 3.5 POSIX.2

Volume 1 of XPG3, *XSI Commands and Utilities*, is based on the SVID, which means that the utilities have the same names and features as the standard utilities supplied with the UNIX System (with some additional utilities). However, when used in an international environment, many of these utilities exhibit additional behaviour, based on the draft *POSIX.2 Standard*. The latter describes how the command interpreter and the utilities of the operating system should work and interface with the user; it is expected to become an official standard very soon.

Volume 3 of XPG3, *XSI Supplementary Definitions*, contains a section specifically about internationalisation, which defines the requirements and pieces together the I18N features in XPG3.

### 3.6 The INTERACTIVE UNIX Operating System

The INTERACTIVE UNIX Operating System is fully compliant with the *POSIX.1 Standard* and with XPG3. The International Supplement adds to the INTERACTIVE UNIX Operating System the items needed for full compliance with the X/Open standard, where appropriate for an operating system, its utilities, and its interface to the C language.

The supplement contains a set of UNIX System utilities that have been enhanced to function according to the description of volume 1 of XPG3. These utilities and their new features are described in section 10 of this document.

The combination of the following software provides customers with a system that is fully compliant with the X/Open standard and that will be branded with the X/Open BASE logo:

- INTERACTIVE UNIX Operating System
- INTERACTIVE Software Development System
- International Supplement

The full seven-volume *X/Open Portability Guide* is now published by Prentice Hall and is available in specialized bookstores. This set

is the only official and complete documentation for the X/Open standard. The documentation supplied with the International Supplement focuses on internationalisation issues only.

## 4. ENTERING DATA

The UNIX System is an interactive, multi-user, time-sharing operating system, which means that several computer users interact with the computer at the same time, usually by typing on a keyboard. This input, as well as the result of the computations done by the application used, is displayed on the computer screen as output.

The device used to interact with the computer is either a self-contained unit with a keyboard and a screen that is connected to a serial port of the computer (a *terminal*), or a directly connected keyboard and a monitor attached to the computer's video card, usually referred to as the *console*.

Input consists of keystrokes that typically represent letters and other symbols, which are pictured on the keys of the keyboard. A computer, however, speaks no particular language and has no notion of what a letter is. Instead, a letter is stored in a computer (either in its memory or in a file on the fixed disk) as a number. Unless every computer system uses the same number to store a certain letter, much confusion is created when attempting to transfer data from one type of machine to another. For that reason, conventions and standards for storing characters into a computer have been created. For more information about this, refer to section 5, "STORING DATA IN THE COMPUTER."

Most keyboards today have 101 or 102 keys. These keys can be divided into three groups:

- The central section of the keyboard
- The numeric keypad
- The function keys

The central section of the keyboard contains keys used to type regular letters and punctuation characters such as the period (.) and semicolon (;). The layout of this section of the keyboard differs from country to country.

The numeric keypad is a section of the keyboard that is designed for easy and fast access to all the numeric characters (0–9) and symbols indicating operators, such as plus (+) and the asterisk (\*). It is often compared to the keys on a calculator. This set of keys can be used in two modes. In the first, they generate the numerals and symbols pictured on the keycaps; in the second, they act as special function keys and cursor movement keys. The mode in effect is

indicated by the NUMLOCK light and can be changed by using the **NUMLOCK** key. When the NUMLOCK light is on, the keys generate the numerals and symbols on the keycaps.

The layout of the function key section of the keyboard depends on the manufacturer, but today most computer keyboards are relatively standard. They usually contain 10 or 12 function keys on the top row of the keyboard, labeled **F1** to **F10** or **F12**. These keys generate sequences of characters, such as **ESC** (escape, the code generated by the escape key) **[o]** **[p]**, often called *escape sequences*.

Applications can take advantage of these keys by determining the actual escape sequence generated by a function key through the `termcap` or `terminfo` interface. These interfaces allow the development of terminal-independent applications.

The layout of both the numeric keypad section and the function key section of the keyboard is the same regardless of the country in which a specific keyboard is used.

#### 4.1 U.S. Personal Computer Keyboard Layout

The central section of a keyboard designed for use in the United States contains keys for all letters of the English alphabet, all digits, and the most commonly used punctuation characters and special symbols. Some of these symbols, the slash (/), for example, are especially important when using the INTERACTIVE UNIX Operating System. In addition, a few special modifier keys are present.

The **SHIFT** key, when pressed simultaneously with a letter key, generates an uppercase character instead of a lowercase character, or alternate symbols instead of the numbers and symbols on the top row.

The **CAPS LOCK** key exchanges uppercase and lowercase. In other words, when this key is pressed, it changes the state of the keyboard so that all characters subsequently typed are automatically uppercase and only appear in lowercase when pressed together with the **SHIFT** key. A CAPS LOCK light indicates the status of the keyboard.

The spacebar generates a space character to put one or more spaces between words. Other special keys are **TAB**, **ALT**, **ENTER**, and **BACKSPACE**. To learn more about the meaning of these keys, refer to the *INTERACTIVE UNIX Operating System Guide*, and for more technical details, refer to the manual entry *keyboard(7)*.

The layout of the keyboard is not randomly chosen but is basically the same as on most typewriters. The layout is often referred to as *QWERTY*, after the order of the first five letters on the top row of keys containing letters. By using the same layout on all typewriters and terminal keyboards, computer users can type in text at a very high speed, regardless of the equipment they are using.





Although one might expect that the layout was chosen to give the easiest access to the most frequently used characters, this is not the case. The QWERTY keyboard layout was originally designed to be slow enough so that mechanical typesetting machine operators would not be able to type fast enough to jam their machines. Another keyboard layout, called DVORAK, places the most common letters in the English language on the home row of keys, but this layout is not in common use.

## 4.2 Generating Characters Not Present on a U.S. Keyboard

Although non-English characters like the German ä or the French ê are not present on a keyboard designed for use in American English, most of these characters can be generated. This allows non-Americans to write French letters on American systems, for example. There are three ways to generate characters for which there are no keycaps (explicit symbols on the keyboard):

- Deadkeys
- Compose sequences
- The decimal representation of the character

### 4.2.1 Deadkeys

The *deadkey* was invented by typewriter manufacturers. For example, imagine you need the French character ê. A French typewriter does not have a key for this character, but it has keys for both e and ^ . When the key  is pressed, a circumflex is printed but the typewriter carriage does not move. When the  key is then pressed, the letter "e" is printed on the same spot as the circumflex and an ê is formed. This technique works very similarly on a terminal. The only difference is that when  is pressed, *nothing* happens until  is pressed, after which the character ê appears on the screen.

A utility developed by INTERACTIVE that can be used to assign deadkeys, `ttymap`, is supplied with the INTERACTIVE UNIX Operating System. This utility is used to do everything discussed in

this section. To define  $\square$  as a deadkey and try the other examples listed below, type the command:

```
ttymap /usr/lib/keyboard/usa.map
```

Now when you press  $\square$ , nothing appears on the screen. When an e is typed next, the letter ê appears. To use the ^ character alone, press  $\square$  first and then the spacebar. If a sequence of two characters is typed that does not make sense at all, no character is sent to the application that is currently being used, and the machine beeps to indicate that an erroneous combination was typed.

#### 4.2.2 Composing Characters Using Compose Sequences

Although assigning deadkeys supports more characters than the ones printed on the keyboard, it has its disadvantages. As illustrated above, it is annoying when one needs the specific character alone that has been assigned as a deadkey. Instead of one keystroke, two keystrokes are needed to access that character. If too many keys act as deadkeys, the system is difficult for everyone to use.

Fortunately, another method exists, often referred to as *compose sequences*. A special key or sequence of keys is used to put the keyboard into a special mode. We will call the key or key sequence the **COMPOSE** key and the special mode the COMPOSE mode. The default **COMPOSE** key sequence for the INTERACTIVE UNIX Operating System is **CTRL** **SHIFT** **F1**. (Many MS-DOS\* (DOS) users will be familiar with it.) When in COMPOSE mode, the system expects two more characters to be typed by the user before a character is generated. Press **CTRL** **SHIFT** **F1** followed by **n**  $\square$  to produce the Spanish ñ (the n in mañana) on the screen. If you press the **COMPOSE** key sequence followed by pressing **!** twice, an inverted exclamation sign appears on the screen.

Both the value of the **COMPOSE** key and the list of **COMPOSE** key sequences and the characters they generate can be specified in a file that is then processed by the `ttymap` command. Refer to the "International Supplement Manual for Advanced Users" or `ttymap(1)` for more details.

Some terminals, for example, the DEC\* VT220,\* have a dedicated **COMPOSE** key on the keyboard, and the characters are generated by the terminal hardware.

### 4.2.3 Decimal Representation

A third method of generating characters is using their decimal representation. As explained in section 5, "STORING DATA IN THE COMPUTER," every character corresponds to a unique number. Up to 256 different characters can be used (although some terminals only support 128). When the **COMPOSE** key is used, followed by three digits, the character that is internally represented by the three-digit number (in decimal) is generated. This feature is also derived from the DOS system. Press the **COMPOSE** key sequence, followed by 065, and an A appears on the screen. 65 is the decimal value used by computers to store the uppercase letter A. Press the **COMPOSE** key sequence followed by 136 and the letter ê appears. If you type:

```
ttymap -d
```

all deadkeys and compose sequences are disabled.

### 4.2.4 Smiling Faces

Those familiar with personal computers and certain DOS applications may have seen interesting images the size of a character, such as smiling faces or musical notes. When control characters are used (characters generated by pressing **CTRL** and a letter key simultaneously), normally nothing is displayed on the screen. However, when the **ESC** key is pressed before pressing **CTRL**, an image appears on the screen (note that this only works on the console). For example, **ESC CTRL a** produces a smiling face.

## 4.3 European Personal Computer Keyboard Layouts

In Europe, computers are sold with either U.S. keyboards (to be used with very technical, engineering-style applications, usually in English) or keyboards designed for the local country. These keyboards differ from U.S. keyboards in the following ways:

- Keyboard layout
- 102 rather than 101 keys

The extra key is usually located between the **SHIFT** key and the leftmost bottom row key (Z on a U.S. keyboard). In most countries, this key has the angle bracket characters, < and >, printed on it. In addition, the backslash key (\) on U.S. keyboards, typically the rightmost or second rightmost key in the top row of the central keyboard section, is usually moved to the left of the **ENTER** key in the

third row (see Figure 1). The layout usually is the same as the one found on typewriters used in these countries. They are often named after the order of the first five keys on the second row of keys; keyboards used in France are called *AZERTY* keyboards, and keyboards used in Germany are called *QWERTZ* keyboards.





Figure 1. French Personal Computer Keyboard Layout

Most Western European languages have an alphabet that contains only a few more letters than English (usually not more than 12). For example, French uses all the letters used in English, as well as a number of accented characters, such as é, è, and á. Some of the characters, such as the ê used in previous examples, are accessed using a deadkey; most of the others are printed on a keycap.

The keys that are used for symbols, such as the square bracket ([]) and curly brace ({}), on U.S. keyboards, have local language accented characters printed on them rather than the American characters (see Figure 1). Although not often used in text, these symbols are certainly important in the context of the UNIX Operating System, especially when the system is used for C programming. Having sacrificed these symbols to support the local language, there must be an alternative way of obtaining them. The solution provided by most keyboard manufacturers is to print three symbols on the top row keys. In addition to the digits and symbols, such as plus (+) and minus (-), the braces and brackets are printed either in the right bottom corner or on the front of the keycap. To generate these symbols, press the key simultaneously with the right **ALT** key. (When using the INTERACTIVE UNIX Operating System, no distinction is made between the left and the right **ALT** key, but in certain applications, such as those based on X11, a distinction is made.)

In the INTERACTIVE UNIX Operating System, `ttymap` input files are provided for all major European keyboards. When the system is properly configured by the system administrator, keyboards function correctly without user intervention, even before logging into the system (an INTERACTIVE feature).

Keyboards to be used in France and Switzerland require special attention. On French keyboards, the **SHIFT** key must be used to access the digits printed on the top row. A Swiss keyboard can be used in two modes. It has keys with four characters printed on it (the same two characters are printed twice, but in opposite order). In German Swiss mode, German characters like ö are accessed by pressing a key, French ones like á by using the **SHIFT** key as well. In French Swiss mode, it works the opposite way.

#### 4.4 Cyrillic or Greek Keyboards

Certain languages, such as Greek or Russian, use completely different alphabets, sometimes referred to as Cyrillic. Although they may look similar, the Russian and Greek alphabets do differ. What

they have in common is the fact that they consist of a reasonably small set of letters (31 for Russian) and that, although some of the letters also exist in English, all of these letters are considered separate from the English set. A personal computer keyboard that supports these languages is designed differently than the ones discussed in the previous section.

The remainder of this section discusses a keyboard designed to support both U.S. English and Russian (use with Greek is theoretically the same). A U.S. English/Russian keyboard (other variants, such as German/Russian keyboards, exist) is physically identical to U.S. English keyboards. The only difference is that in addition to the English letters, the Russian letters are also pictured on the keycaps, usually in a different color (see Figure 2). Using `ttymap`, the keyboard is mapped to generate Russian characters when a key is pressed. A special key, called a toggle key, can be used within an application to switch between Russian and English. The default sequence for toggling between languages is `[CTRL] [SHIFT] [F2]`.

This feature of the INTERACTIVE UNIX `tty` system and the `ttymap` utility has been especially designed to support languages such as Greek and Russian. The same toggle key can be used with European keyboards to temporarily cause deadkeys to no longer act like deadkeys, for example. A French programmer might decide to use the toggle key when he switches between a C source code file and a French text file, for example.



**Figure 2.** English/Russian Personal Computer Keyboard Layout

## 4.5 Keyboard Layouts on 7-bit Terminals

The keyboards described so far are keyboards that are attached to devices capable of supporting 256 different symbols. Certain terminals only support up to 128 different symbols. The national keyboards supplied with these terminals sacrifice some of the symbols (such as { and \, although these are very useful in the context of the UNIX Operating System) and replace them with local language characters. The terminal itself usually has a **SETUP** key that allows the user to specify the language of choice to make the keyboard function properly.

The substitution characters can still be generated, but not displayed (see section 6, "DISPLAYING DATA"). To accommodate programmers who use such terminals, a new feature called trigraphs has been introduced into the *ANSI C* language. *Trigraphs* are three-letter sequences used in an *ANSI C* source file that are interpreted as a single symbol (essential to the C language). This allows a programmer who uses an Italian 7-bit terminal, for example, to still get the job done. The one-to-one relationship between trigraphs and the symbols they represent is listed in the table below:

<i>Trigraph</i>	<i>Symbol Represented</i>
??=	#
??/	\
??^	^
??(	[
??)	]
??!	
??<	{
??>	}
??~	~

Note that this feature is not available with the traditional Kernighan and Ritchie C compiler.

## 4.6 Using the VP/ix Environment

The Virtual Personal computer Interactive eXecutive environment (VP/ix\*) is a product developed and sold by INTERACTIVE Systems Corporation. It is a UNIX System application that emulates an *IBM\** PC/XT\*-compatible computer, which allows users of the INTERACTIVE UNIX Operating System to run DOS and DOS

applications as if they were UNIX System utilities. A copy of DOS is furnished with the product and is used by default whenever `vpix` (the name of the actual command) is invoked.

When the VP/ix Environment is used, all previously installed keyboard mapping is automatically disabled until the user leaves the VP/ix Environment. If a non-U.S. keyboard is used, DOS must be informed. With the VP/ix Environment, the system administrator can choose to give each VP/ix user an individual `C:` drive (this is a virtual disk drive, in reality a UNIX System file, that contains DOS and is used to boot it) or to use a system-wide `C:` drive. When a non-U.S. keyboard is used, using individual `C:` drives is preferable because this drive contains the essential DOS system files, `CONFIG.SYS` and `AUTOEXEC.BAT`, that need to be edited to insert information about the keyboard and language used, as well as which country's conventions should be applied. Refer to the documentation that accompanied your DOS system for details.

#### 4.7 Entering Data and Using INTERACTIVE X11

When INTERACTIVE X11 is used with the system, a special program called a `display server` is invoked. This program switches the system from a character-based environment to an all graphical environment. From that point on, all mapping information specified through the `ttymap` interface is no longer used. The server program is responsible for performing the correct actions each time a key is pressed on the keyboard. By default, it treats any keyboard as a U.S. keyboard. A utility called `xttymap` is provided to change the default actions of the server. It can read and interpret the same input file that is used with `ttymap`.

Due to limitations in the MIT code of X11 Release 4, **COMPOSE** key sequences and deadkeys cannot be supported when X-based applications are run. The one exception to this, however, is when text-based applications are used in an `xpcterm` window. These applications have access to the tty system, so `ttymap` can then be used to define deadkeys or compose sequences.

## 5. STORING DATA IN THE COMPUTER

The previous section explained how keyboards are used to generate letters and other characters on a computer running the INTERACTIVE UNIX Operating System. Typically, these characters are processed by the application that is currently running (it could be the shell, which is the command interpreter, or an editor, or any other application). In most cases, the characters are echoed on the screen.

Applications such as editors, `vi` or `e` (the TEN/PLUS\* editor), for example, store these characters in a file. As mentioned earlier, a computer speaks no particular language and has no notion of what a letter is. It stores numbers in the file rather than letters. Unless every computer system uses the same number to store a certain letter, files created on one computer cannot be read on another.

Most computer manufacturers use the same convention to represent characters internally; however, some differences in standards do exist. For example, many IBM computers (not PCs) use a standard called EBCDIC. The UNIX Operating System was designed to use the American Standard Code for Information Interchange (*ASCII*) standard for internal storage.

### 5.1 ASCII

ASCII is a convention, or *codeset*, describing one-to-one relationships between symbols and numbers. It represents letters as numbers that can be stored in 7 bits of the computer's memory, which means a choice of 128 different symbols (0 to 127). The numbers 0 to 32 are reserved for characters that cannot be displayed on the screen but have a special meaning to the system (so-called nonprintable characters). As an example, 7 represents the sound a computer makes when you press **CTRL** **g**. These characters are often referred to as control characters because the **CTRL** key is needed to generate them. The smiling faces that can be produced on the console (as discussed in the previous section) are not part of the ASCII standard.

Only 7 bits of internal storage are needed to store 128 different numbers (0 – 127), so the ASCII codeset is called a 7-bit codeset (7-bit US ASCII).

The 96 printable ASCII characters are encoded as follows:

32	33	!	34	"	35	#	36	\$	37	%	38	&	39	'	
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	

There are a few interesting points about the ASCII codeset. Uppercase characters are represented using lower numbers than lowercase characters, and the difference between the value of an uppercase character and its corresponding lowercase character is constant (32). This has often been used (and misused) by programmers. The last character, 127, is not always printable. This does not cause any problems, as this character is used by the INTERACTIVE UNIX Operating System as the DELETE character to interrupt programs. The ASCII codeset contains all letters of the English alphabet and none of the additional letters used in French, German, and other languages.

## 5.2 8-bit Characters and Codesets

Inside the computer, 7-bit numbers are actually stored as 8-bit entities. In most computers, a byte (8 bits or a series of 8 possible zeroes and ones) is the smallest possible unit used to store information, which makes it possible to actually use 256 different characters and symbols. Today this is true if you use the console. If you have a compiler on your system, you can compile and run the following program:

```
#define XOPEN_SOURCE
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
    int c;
    c = 32;
    while (c <=255) {
        printf("%4d %c ",c,c);
        if ((c+1)%8 == 0)
            printf("\n");
        c++;
    }
    printf("\n");
}
```

to display all letters and symbols that you can use on the console and the number by which they are represented inside the computer.



If you are not familiar with the C language, follow these instructions to compile and run this program:

1. Use an editor to create a file with a name that ends in `.c`, for example, `show.c`, and insert the exact text of the program.

2. For example, to create `show.c`, type:

```
make show
```

3. Then to run the program, type:

```
./show
```

Historically, the eighth bit of the byte that is used to store characters was used by the UNIX Operating System and its utilities for a variety of purposes. It could be used in a sorting algorithm to see if a character was already processed or, when a program allocated bytes of memory, to indicate that the byte was already used. In communication software across telephone lines (which are not 100 percent reliable), the eighth bit was used to do additional checking by forcing the software to always use either even or odd values for the number represented by the byte to send across the wire. This bit was then called a parity bit.

Most utilities provided with the UNIX Operating System were careless enough to ignore the value of this last bit, preventing the use of characters with the 8-bit set (such as the ones displayed when running the program listed above), usually referred to as 8-bit characters. Utilities such as `vi` were basically useless for editing non-English texts.

Beginning with UNIX System V Release 3.1, most utilities became what is called "8-bit clean." The INTERACTIVE UNIX Operating System is based on UNIX System V Release 3.2 and therefore contains these 8-bit utilities.

As 8-bit characters are now supported, an 8-bit codeset can be used, and the convention is to map 256 unique symbols to 256 unique numbers. As might be expected, more than one such codeset exists in the industry. Fortunately, all have one important feature in common: the first 128 characters of these codesets are exactly the same as the characters in the ASCII codeset. In other words, they are all supersets of the ASCII codeset.

### 5.3 IBM Codepages

The codeset used in IBM-compatible personal computers is probably the single most popular codeset used today, primarily by people who are not even aware that it is designed to support non-English languages. Until recently, this codeset was referred to as IBM-extended ASCII (which is a very good description of what an 8-bit codeset is: it extends the 128 character ASCII codeset by another 128 characters).

The characters used in this codeset and the way they are encoded are exactly those characters displayed by the sample program, `show.c`, used in section 5.2, "8-bit Characters and Codesets." If you run this program again and look at the output, you will note the following:

- There is a symbol for almost every code in the second half of this codeset.
- The symbols consist of accented letters, both uppercase and lowercase, special symbols, and graphics characters to draw lines and boxes.
- For some lowercase accented characters, there are no uppercase equivalents (for example, `ë`).

Many personal computer programmers and applications use the graphics characters to draw straight lines, draw boxes around text, and so on. This codeset clearly supports most characters used in the major Western European languages, such as French and German. In recent years, alternate codesets were developed for personal computers, and software was developed to change the codeset used by them when running DOS. (Software to support this was developed for the INTERACTIVE UNIX Operating System as well.) In the DOS world, the name *codepage* was used, and the popular IBM-extended ASCII codeset is now called IBM codepage 437.

The introduction of additional codesets supports more languages spoken in a particular territory. A list of some of the existing IBM codepages and the targeted area or language includes:

<i>Codepage</i>	<i>Territory or Language</i>
437	U.S. English and Western Europe
850	International codepage (supports more letters and fewer graphics characters than codepage 437)
863	Canada
865	Norway/Denmark
866	Supports Russian alphabet

This list is incomplete; there are codepages for Greek and for the Slavic languages as well. Try running the program from the previous section again, but showing codepage 850 instead. Type:

```
loadfont 850
```

The screen will flash and the shell prompt will reappear. Now the console is using a different codeset. Notice the differences between the output of the command and the previous output. To switch back, type:

```
loadfont 437
```

## 5.4 ISO Codesets

The organization that sets international standards, called *ISO*, has also defined 8-bit codesets to be used on computer systems in different territories. This standard is more widely adopted on larger computer systems running the UNIX Operating System. This family of codesets is referred to as the ISO 8859 standard. The codeset used in Western Europe is the 8859-1 codeset, which is the standard adopted by the X/Open Company for information interchange. Type:

```
loadfont 8859
```

and run the `show` program again. The following can be observed:

- There is no symbol for the first 32 values of the second 128 numbers.
- There are no graphics characters to draw boxes.
- The difference between the values of an uppercase character and a lowercase character is always constant (32).

- The values chosen for the accented characters are different from IBM codepage 437 (for example, ê is represented by 234 in ISO 8859-1 and by 134 in IBM codepage 437).

To switch back, type:

```
loadfont 437
```

There are 9 different 8859 codesets, each for a different territory. The most important ones are:

<i>ISO Codeset</i>	<i>Territory or Languages Intended</i>
ISO 8859-1	Western Europe
ISO 8859-2	Eastern Europe (English, Czech, Polish and so on)
ISO 8859-5	English and Russian alphabet
ISO 8859-7	English and Greek alphabet

## 5.5 7-bit Codesets

Earlier in this document, we described terminals that support only 128 different characters and use a **SETUP** key to select a language or country. The 7-bit characters generated by most of these terminals follow an ISO standard convention, ISO 646, which is the ISO code name for the ASCII standard. For use with languages other than English, the local language letters are substituted for symbols such as {.

## 5.6 Choosing and Configuring a Codeset

It is the system administrator's responsibility to deal with codesets. The INTERACTIVE UNIX System utility that configures the system to correctly store characters that are generated by the keyboard is the same utility that is used to configure the keyboard, `ttymap`. The system administrator has to verify that data storage happens consistently, regardless of the type of terminal used. Otherwise what was edited as a ë on the console yesterday may appear as a { on a regular terminal today.

The system administrator must choose between one of the IBM codepages and one of the ISO 8859 conventions. The first issue that determines that decision is obvious – which language(s) will be used on the system. The other criteria that should be considered in this decision are as follows:

- If many files developed on a DOS system need to be processed or many applications will be used in the VP/ix Environment, an IBM codepage should be used.
- If the system needs to communicate with a heterogenous network of computers, an ISO 8859 codeset is the better choice.

All the files supporting international keyboards that are supplied with the INTERACTIVE UNIX Operating System (which are located in `/usr/lib/keyboard`) configure the console to use the IBM codepage 437 (850 for Norway). Additional mapping files are provided as-is with the International Supplement, located in sub-directories of `/usr/lib/keyboard`. They are named after the codeset, 437 or 8859-1, for example, and their names follow the X/Open convention for `locale` names, for example:

```
/usr/lib/keyboard/8859-1/fr_FR
```

which represents the mapfile for French in France, using the ISO 8859-1 codeset.

### 5.6.1 Converting From One Codeset to Another

The International Supplement contains a utility, `iconv`, which can be used to convert the encoding of characters in a file from one codeset to another. The following example shows the command needed to convert the encoding in `filename` from the IBM codepage 437 to ISO 8859-1:

```
iconv -f 437 -t 8859 filename > file.new
```

Refer to `iconv(1P)` for more details.

## 6. DISPLAYING DATA

When characters are displayed on the screen of your terminal or console, these characters physically consist of a set of white dots that make up the picture of the character. Typically, a rectangle of 8 by 16 dots is reserved for every character. The one-to-one relationship between a character (actually the numeric representation of a character) and its picture is called a font. Depending on how the INTERACTIVE UNIX System is used, fonts may or may not be modified.

After typing a character and possibly storing that character in file, a code (usually the same as the input code) is sent to the terminal to indicate that it should display something. If necessary, the code sent by the system or the application can be modified before it is sent to the screen. This practice is called *output mapping*. Again, `ttymap` is the utility responsible for this function. Proper output mapping and possible modification of the font guarantees the display of the proper character (or, when the actual character cannot be displayed, at least something that makes sense). Here are a number of suggestions for making the INTERACTIVE UNIX System work correctly.

### 6.1 7-bit Terminals

When 7-bit character terminals are used, a 128-character font that is hardcoded inside the terminal hardware is used. This font cannot be modified, but more sophisticated terminals allow access to several different fonts, one for each language supported. These terminals support the ISO 646 ASCII variants described in the previous section. To ensure consistency throughout the system (assuming a French 7-bit terminal is used):

- On input, map the 7-bit code generated for the French characters into their actual 8-bit value.
- On output, map the 8-bit code back to the 7-bit code to display the correct French character.
- Use trigraphs for ANSI C programming.
- To generate curly braces and other such characters, use the decimal representation. On output, map to a space character.

This ensures the proper display of the file used, especially when the same file is later edited on devices such as the console.

If the inability to display curly braces and other typical UNIX System characters, such as `\`, is too annoying, use this alternative approach:

- Use the **SETUP** key of the terminal to switch it to U.S. English. You now have access to a U.S. ASCII font but still have a French keyboard layout.
- When a French character key is pressed, it is mapped and stored using its correct 8-bit value.
- On output, it is mapped to the corresponding character without the accent, or the closest-looking English letter (for example, a `c` instead of a `ç`)
- Use decimal representation for the UNIX System characters, which are automatically stored as 7-bit characters and displayed correctly.

Your system administrator should develop the correct `ttymap` description file for your machine.

## 6.2 The Console

On the console, a font of 256 different symbols can be used. That font information is stored in Random Access Memory (*RAM*) on the video card inside the computer, to which the monitor is attached. The information can be changed (on old or inexpensive systems, the information is stored in Read Only Memory (*ROM*) and can only be changed by replacing the ROM with a different ROM).

INTERACTIVE has developed a utility called `loadfont` to change the font information in the video card. This utility has predefined, built-in fonts. However, anyone can use it to develop a personalized font. Refer to `loadfont(1)` for more information.

## 6.3 Displaying Data and Using INTERACTIVE X11

INTERACTIVE X11 and X11-based applications always use fonts when text is displayed. Most applications have a command line option, `-fn`, to indicate which font to use. Fonts for both the 8859-1 (most of the supplied fonts) and IBM 437 codesets are supplied with INTERACTIVE X11. The font files supplied with the International Supplement can also be used with INTERACTIVE X11 after converting them with the `bdf to snf` utility.

## 7. THE INTERNATIONAL ENVIRONMENT

The internationalisation features discussed thus far have all involved compliance with international standards and the ability to correctly enter, store, and display the letters used by the local language. Some of the other features an internationalised system should have are discussed here.

The *X/Open Portability Guide* dedicates 7 chapters to internationalisation (see Volume 3, *XSI Supplementary Definitions*, chapters 2-8) describing these features. The INTERACTIVE UNIX Operating System supports all the features described there. The abilities described allow developers to create internationalised applications and users to take advantage of the fact that these applications are indeed internationalised.

An internationalised application is a program that makes no hard-coded assumptions about the language, the local customs, or the coded character set. When the proper environment is set up for the user of that application, a program that displays the date displays it according to the local custom, a program that sorts takes into account the “natural” order of letters, and so on.

The international environment is used to define *user preferences*, and internationalised utilities and features adapt their behaviour to those preferences, even when they change. A default environment is often established, but the user is always free to change the environment as required.

The remainder of this section describes the international environment, how it is set up, and how it interacts with internationalised utilities and applications.

### 7.1 The International Environment

Running applications in an internationalised environment is based on the concept of a local environment or `locale`, which is defined as the subset of the user's environment that depends on language and cultural conventions.

A `locale` consists of a number of categories, with each category controlling a specific aspect of the international environment. Each category is usually referred to by the variable used to set or modify it. The International Supplement recognizes the following categories:



- *Date and Time Format*

This category, `LC_TIME`, affects how date and time are displayed.

- *Character Classification*

This category, `LC_CTYPE`, defines codeset characteristics and character classification.

- *Collation*

This category, `LC_COLLATE`, affects the collation (“sorting”) order.

- *Numeric and Monetary Formatting*

These categories, `LC_NUMERIC` and `LC_MONETARY`, affect the format of nonmonetary and monetary numeric information, such as the decimal delimiter.

- *Yes/No Responses*

This category, `LC_MESSAGES`, affects the strings used to indicate yes/no answers to utility and application queries. (Note that while the internationalised yes/no response is required by XPG3 for certain commands, the `LC_MESSAGES` category is not part of the `locale` as defined by XPG3.)

- *Message Catalogues*

Message catalogues are not yet covered by the `locale` categories, but use similar mechanisms.

The `locale` and the various categories only affect the behaviour of an application if the application is set up to do so. This ensures that old applications do not suddenly start behaving strangely. In addition, a particular `locale` instance that describes the desired behaviour must also have been created. Such instances are referred to by their name. X/Open has adopted a format for constructing `locale` names that makes them easy to identify. The format is:

```
language[_territory][.codeset]
```

where *language* is a two-letter abbreviation, for example, `fr` for French; *territory* is a two-letter abbreviation, `FR` for France or `CA` for Canada, for example; and *codeset* is the codeset designation, such as `437`. One `locale` category is always present – the “C” or POSIX `locale`, which defines the traditional UNIX System behaviour.

The creation of `locale` instances is described in the “International Supplement Manual for Advanced Users.”

## 7.2 Controlling the International Environment

A programmer can set and change the `locale` explicitly inside a program. This can be done to ensure a particular environment, for example, so that a particular program always behaves the same way. In most cases, however, the programmer leaves the choice to the end user by specifying that the `locale` be set to what the end user specified via environment variables. The environment variables are:

- LC\_ALL** If this environment variable is set, the environment is set to that locale for all categories, regardless of whether any of the other variables are set. Example: `LC_ALL=fr_FR.437`.
- LC\_COLLATE** This environment variable defines the desired environment for the `LC_COLLATE` category. `LC_COLLATE=fr_CA.863`, for example.
- LC\_CTYPE** This environment variable defines the desired environment for the `LC_CTYPE` category. Example: `LC_CTYPE=C`.
- LC\_MESSAGES** This environment variable defines the desired environment for the `LC_MESSAGES` category. `LC_MESSAGES=de_DE.850`, for example.
- LC\_MONETARY** This environment variable defines the desired environment for the `LC_MONETARY` category. Example: `LC_MONETARY=es_ES.8859-1`.
- LC\_NUMERIC** This environment variable defines the desired environment for the `LC_NUMERIC` category. `LC_NUMERIC=da_DK.865`, for example.
- LC\_TIME** This environment variable defines the desired environment for the `LC_TIME` category. Example: `LC_TIME=en_UK.437`.
- LANG** If this environment variable is set, the specified value is used for all categories not explicitly set; in other words, it is the

“fallback” (unless LC\_ALL is also set).  
The LANG variable is also used to locate a  
specific message catalogue. Example:  
LANG=en\_US.

## 8. INTERNATIONALISED BEHAVIOUR

This section explains how the international environment affects the behaviour of system utilities and applications.

### 8.1 Date and Time Format

The default conventions for the date and time format, as well as the names of the days of the week and months, follow U.S. conventions and are rarely applicable in other countries. By defining and using the date and time environment, the dates and times displayed by the system, utilities, and applications follow the local conventions and use the names of the days and months in the correct language.

The following aspects of formatting are supported by the INTERACTIVE UNIX Operating System:

- Format of time display.
- Format of date display.
- Format of combined date and time display.
- Format of 12-hour time display.
- Names of days of the week.
- Abbreviated names of days of the week.
- Names of the months.
- Abbreviated names of the months.
- Format of the ante meridiem and post meridiem strings used in 12-hour clock time displays.

For example: In a French environment, the output of `date` could be:

```
Mardi 30 juillet 1991 11:07:35 PDT
```

and the output of `ls -l`:

```
total 636
-rw-r--r--  1 paul   other    27399 janv. 24 18:36:02 ch01
-rw-r--r--  1 paul   other    13842 juil.  9 18:36:03 ch02
-rw-r--r--  1 paul   other     9057 mai  12 18:36:03 ch03
-rw-r--r--  1 paul   other      263 mai  12 15:44:45 document
-rw-r--r--  1 paul   other      398 sept. 24 12:37:34 Makefile
-rwxr-xr-x  1 paul   other    24202 avril 10      1991 show
```

## 8.2 Character Classification

Regardless of how it is encoded, a character has certain features. For example, it is either printable or nonprintable. If a different codeset is used, different numbers represent the characters. To keep track of this, the system uses a classification table, which contains information about all 256 characters in the codeset. Things that can be specified are:

- Lowercase letters
- Uppercase letters
- Digits
- White-space characters
- Punctuation characters
- Control characters
- Uppercase to lowercase conversion
- Lowercase to uppercase conversion
- Printable characters or nonprintable characters

Programs that are written to use functions like `isupper` and `isdigit` (refer to `ctype(3C)`) access this table and behave accordingly. The default table used by the system is the ASCII table that considers every 8-bit character nonprintable. This explains why programs such as `vi` do not display 8-bit characters correctly, but their octal representations instead, unless the proper environment is set up.

Using more than just the ASCII characters changes the meaning of many things, including the meaning of regular expressions. The string `[a-z]` no longer represents *all* lowercase characters. In some languages, there are alphabetic characters after `z` in the dictionary, and as discussed earlier, most codesets contain lowercase characters that are stored as 8-bit characters, which would be ignored if the above expression were evaluated numerically.

The *X/Open Portability Guide* specifies internationalised regular expressions. It introduces keywords that can be used to specify classes of characters, for example, `[:lower:]` is a regular expression that means “any lowercase letter.”

The INTERACTIVE UNIX Operating System fully supports internationalised regular expressions. Where appropriate, UNIX System utilities have been enhanced to support these capabilities. These utilities are supplied with the International Supplement (see section 10, "INTERNATIONALISED INTERACTIVE UNIX SYSTEM UTILITIES"). For a detailed description of internationalised regular expressions, refer to *regex*(5P).

### 8.3 Collation

*Collation*, according to a dictionary, is the "act of putting things in their proper order." Thus, collation rules define how the data are put in the proper order, or *sorted*. Traditionally, the collating order in the UNIX System has been ASCII order, that is, the order in which the characters appear in the ASCII codeset. This is the natural collating order for the English language.

For most languages in the world, however, this is not enough. Most European languages contain more letters than the 26 in the English language, with the additional letters typically collating between the letters in the ASCII set. For instance, an accented *á* sorts between a and b. The average European user expects sorted lists (for instance, the output from the *ls* command) to appear in the collation order of his or her language.

Languages with non-Latin-based alphabets, such as Russian or Greek, use a completely different set of characters. For these languages, collation takes on additional complexities.

The INTERACTIVE UNIX Operating System allows users to define their own collation order. This capability is a superset of the X/Open requirement for an internationalised system and is expected to satisfy the requirements for dictionary ordering for most European languages and non-European alphabetic languages. The standard utilities that depend on collation, such as *sort* and *ls*, have been modified to understand this user-specified collation order and are supplied with the International Supplement.

#### 8.3.1 An Example

Consider the following four lines (the four seasons in French):

```
printemps
été
automne
hiver
```

The regular UNIX System *sort* utility sorts them as follows:

```
automne
hiver
printemps
été
```

It uses the numeric representation of characters, and because *é* is represented by an 8-bit character, it is listed last. The UNIX System `sort` used to strip the eighth bit, sorting the above sequence as:

```
été
automne
hiver
printemps
```

which is, of course, wrong as well. (Making utilities 8-bit clean is not always sufficient.)

The internationalised `sort` gives the following (correct) result:

```
automne
été
hiver
printemps
```

## 8.4 Numeric and Monetary Formatting

The default conventions for decimal delimiter and other numeric formatting rules are seldom correct in an international environment. For example, the default decimal delimiter in the U.S. is a period, but in most European countries the comma is used instead, which, in turn, is used in the U.S. as the thousands separator character. So \$1,000, which is one thousand dollars in the U.S., could be interpreted as a single dollar in Europe. Misinterpreting things the other way around could be quite an expensive mistake! By defining numeric and monetary formatting with the correct values, programs display fractions using the appropriate decimal delimiter.

Applications such as accounting programs often have to be modified to display the correct monetary symbol. The manner in which numbers representing amounts of money are formatted is also subject to local conventions.

## 8.5 Yes/No Responses

Some utilities, such as `rm`, require the user to acknowledge whether a specific action should be taken. The usual response is either “yes” or “no.” Before internationalisation, such utilities required the user to respond using the English `y` or `n`. Such a response is not natural to French-speaking people in the world, where, of course, `oui` would be more natural instead of `yes`. INTERACTIVE has added

the capability to define the correct yes and no responses for a particular `locale`.

## 8.6 Message Catalogues

The message catalogue system specified by XPG3 allows program messages to be stored separately from the logic of the program, to be translated into different languages, and to be retrieved at run time, according to the language requirements of the user. This means that a single application (a single UNIX System executable) can support many languages. The program can be translated without requiring access to the C source code of the application — all that is needed is a message catalogue source file in one language, which can be used to translate it in to other languages.

For performance reasons, two different message catalogue formats are used:

- A message text source file.
- A message catalogue (used by the application and produced from the message text source using a new utility called `genocat` (refer to `genocat(1P)`). INTERACTIVE has also added a utility, `showcat`, that can be used to translate the contents of a message catalogue into its message text source (that is, the opposite of the `genocat` utility), unless an option to prevent this translation was used when `genocat` was used to create the message catalogue. Refer to `showcat(1P)` for more information.

## 8.7 The X/Open Environment

The set of internationalisation features described previously functions according to the *X/Open Portability Guide* and far exceeds those supported by UNIX System V.

Every application developed using the INTERACTIVE Software Development System and compiled with the `-Xp` option has access to this functionality. The International Supplement provides the ability to create and use `locales` other than the default (U.S. English, the “C” `locale`). It also provides the enhanced UNIX System utilities that understand the X/Open announcement mechanism (discussed below).

Refer to the “International Supplement Manual for Advanced Users,” the “International Supplement Reference Manual,” and Volumes 1, 2, and 3 of the *X/Open Portability Guide* for more details.



## 9. THE SYSTEM V ENVIRONMENT

Beginning with UNIX System V Release 3.1, serious attempts were made to make the UNIX Operating System function better in an international environment. Most UNIX System utilities that stripped the eighth bit of a byte were made 8-bit clean. In addition, some of the functionality described in the previous section was made available (in particular, date and time formats and character classification). In order to access the local language information, a utility or application needs to know its location. The mechanism used to communicate its location is called an *announcement mechanism*. Unfortunately, the System V and X/Open announcement mechanisms are different. The System V mechanism is described in this section because certain UNIX System utilities, such as `vi`, support it.

### 9.1 Date and Time Formats

Most UNIX System utilities that display the time or the date (`date` and `ls`, for example) and all applications developed on UNIX System V that use the `cftime` function (see `cftime(3C)`) can be given access to a different method of displaying the date (typically, in a different language, but the feature can also be used if you want to call Saturday “Partyday” instead, for instance). The date and time information needs to be stored in a text file. The following information is required:

- Abbreviated month names (in order)
- Month names (in order)
- Abbreviated weekday names (in order)
- Weekday names (in order)
- Default strings that specify formats for local time and date
- Strings used to replace AM and PM

This file must be stored in the directory `/lib/cftime`. When the shell variable `LANGUAGE` is set to the name of the file, the date and time are displayed accordingly. (Note that the X/Open mechanism uses `LC_TIME` instead.)

### 9.2 Character Classification

UNIX System V Releases 3.1 and later also supports character classification. A utility, `chrtbl`, converts a text file that contains a

description of the codeset into a binary file. When that file is installed in `/lib/chrclass`, and the shell variable `CHRCLASS` is set to the name of that file, the correct character classification is used. The format of that file is described in *chrtbl(1M)*. (Note that the X/Open mechanism uses `LC_CTYPE` instead.)

Although the use of the X/Open announcement mechanism is recommended, the System V method should be used for System V utilities and applications, such as `vi`, which were not internationalised for XPG3. All programs that are written using functions such as `isupper` have access to this mechanism.

## 10. INTERNATIONALISED INTERACTIVE UNIX SYSTEM UTILITIES

To use the internationalisation features described in the previous sections, a number of UNIX System utilities needed to be modified. Their enhanced behaviour complies with the specifications listed in volume 1 of the *X/Open Portability Guide* (Issue 3). Most of the differences in behaviour are transparent to the user. In most cases, when no local environment (`locale`) is set up, the behaviour defaults to the standard System V behaviour. The manual entries in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual* have not been modified to reflect the internationalised behaviour. Refer to volume 1 of the *X/Open Portability Guide* for more details.

These utilities are supplied with the International Supplement and are installed in the directories where the original UNIX System V utilities are located. A list of these utilities and the `locale` categories they understand follows. One category, described in the "International Supplement Manual for Advanced Users," which deals with regular expressions, is referred to as Internationalised Regular Expressions (Int. RE).

The following utilities are supplied:

Utility	Categories					
	Int. RE	LC_- CTYPE	LC_- COLL ATE	LC_- TIME	LC_- NUME- RIC	LC_- MESS- AGES
ar				Y		
awk	Y	Y	Y		Y	
comm			Y			
cp						Y
cpio		Y	Y	Y		
csplit	Y					
date				Y		
ed	Y	Y	Y			
egrep	Y	Y	Y			
expr	Y	Y	Y			

Utility	Categories					
	Int. RE	LC_ CTYPE	LC_ COLL ATE	LC_ TIME	LC_ NUME- RIC	LC_ MESS- AGES
fgrep		Y				
find		Y	Y			Y
grep	Y	Y	Y			
join			Y			
ln						Y
lpstat				Y		
ls		Y	Y	Y		
mail				Y		
mv						Y
pg	Y	Y	Y			
pr		Y		Y		
ps				Y		
red	Y	Y	Y			
rm						Y
rsh		Y	Y			
sed	Y	Y	Y			
sh		Y	Y			
sort		Y	Y		Y	
tar				Y		Y
tr		Y	Y			
uniq			Y			
uucp		Y	Y	Y		
uustat				Y		
uux		Y	Y	Y		
wc		Y				
who				Y		
yacc		Y				

For `awk`, the period (.) is used as the decimal delimiter in scripts (to provide portability), but in data to be processed, as well as output, the decimal delimiter of the current `locale` is honored.

`ar` and `yacc` are supplied with the INTERACTIVE Software Development System rather than the International Supplement.

In addition to the functionality specified by XPG3, other `uucp`-related commands have been changed so that they are affected by the category `LC_TIME` in the `locale`. One of these commands,

`uux`, is included in XPG3; the remainder are not. (They may be found in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.) The following is a summary of the additional functionality:

`uucico`, `uusched`, `uux`, `uuxqt`

`LC_TIME` determines the format of date and time strings output by these commands.

`uucleanup`

`LC_TIME` affects the format of date strings included in messages composed by `uucleanup`.



## GLOSSARY

### *announcement mechanism*

The mechanism used to communicate the location of local language information.

### *ANSI*

American National Standards Institute.

### *ASCII*

American Standard Code for Information Interchange.

### *AZERTY*

Name used to reference French keyboard layouts.

### *CAE*

Common Applications Environment

### *codepage*

A codeset. This term is used in the DOS world, particularly by IBM.

### *codeset*

A convention describing one-to-one relationships between symbols and numbers. It represents letters as numbers that can be stored in a computer's memory.

### *collation*

The act of putting things in their proper order (sorting).

### *compose sequence*

A special key or sequence of keys used to put the keyboard into a special mode where the system expects two more characters to be typed by the user before a character is generated. The default **COMPOSE** key sequence for the INTERACTIVE UNIX Operating System is **CTRL** **SHIFT** **F1**.

### *console*

A directly connected keyboard and a monitor attached to a computer's video card.

### *deadkey*

A procedure for overprinting invented by typewriter manufacturers, where when one key is pressed, a character is printed but the typewriter carriage does not move until the second key is pressed, so that characters consisting of two separate characters, such as ê, can be formed. The INTERACTIVE `ttymap` utility can be used to assign deadkeys. The only difference is that when the first key is pressed, nothing happens until the second key is pressed, after which the entire character appears on the screen.

*escape sequence*

Sequences of characters, such as **ESC** (escape, the code generated by the escape key) **o** **p**.

**IBM** International Business Machines.

**IEEE** Institute of Electrical and Electronics Engineers, Inc.

*internationalisation*

Making a computer, a computer system, or a computer program function appropriately in a non-U.S. environment.

**ISO** The international standards organisation. (Note that ISO is not an acronym.)

**ISV** Independent Software Vendor.

**I18N** Internationalisation.

**L10N** Localisation.

**locale** An abbreviation for the X/Open concept **local** environment, that subset of the user's environment that depends on language and cultural conventions. It consists of the following categories: Date and Time Format, Character Classification, Collation, Numeric and Monetary Formatting, Yes/No Responses, and Message Catalogues.

*localisation*

The adaptation of computer programs to a single language and/or country.

*output mapping*

Modification of the code sent by the system or the application to the screen before a character is displayed.

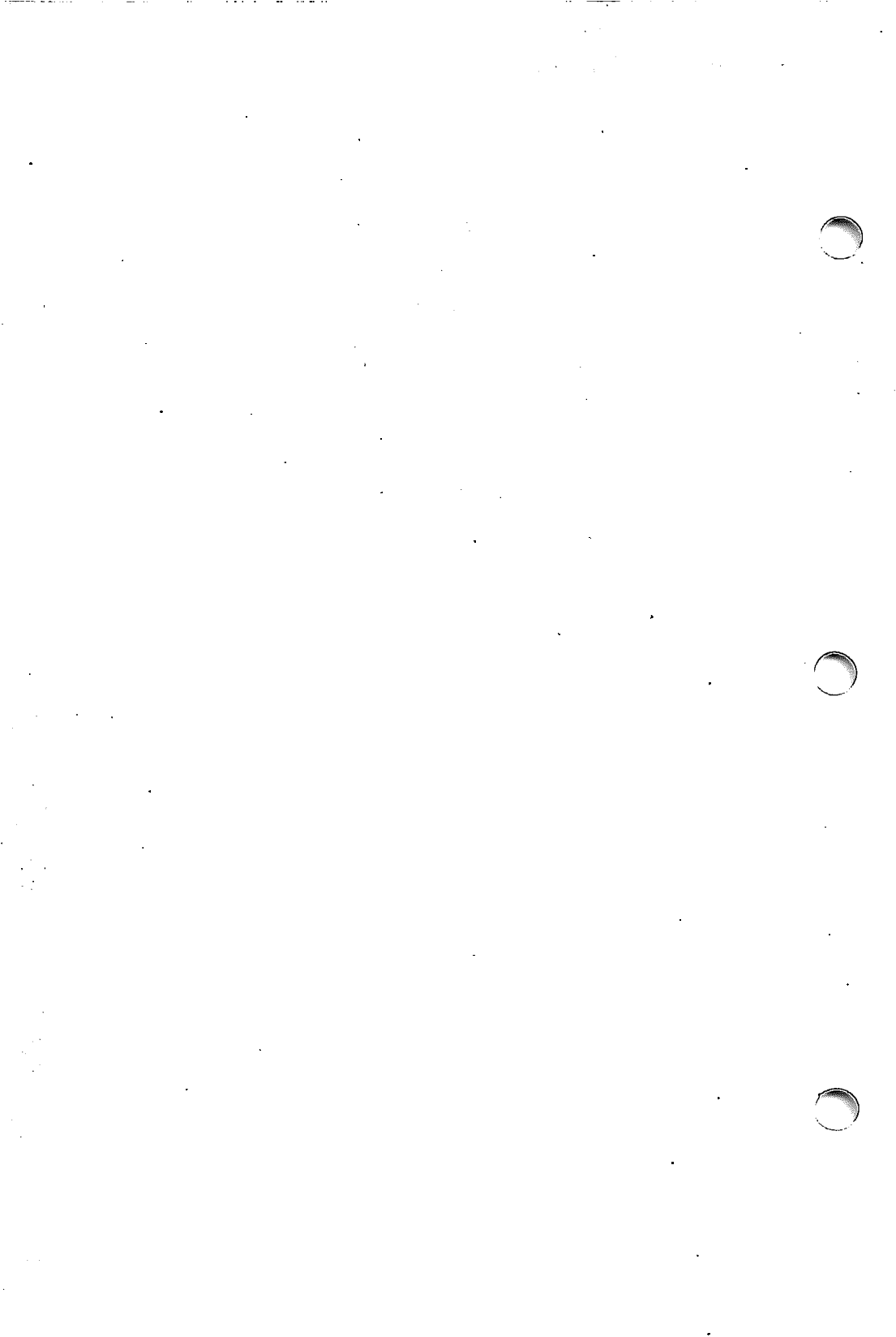
**POSIX** Portable Operating System Interface for Computer Environments.

**POSIX.1** International Standard (ISO/IEC 9945-1) defining system interfaces.

**POSIX.2** Draft standard for shell and utilities.



<i>QWERTY</i>	Name used to reference U.S. English keyboard layouts.
<i>QWERTZ</i>	Name used to reference German keyboard layouts.
<i>RAM</i>	Random Access Memory.
<i>ROM</i>	Read Only Memory.
<i>SVID</i>	System V Interface Definition.
<i>terminal</i>	A self-contained unit with a keyboard and a screen that is connected to a serial port of a computer.
<i>trigraph</i>	Three-letter sequences used in an ANSI C source file that are interpreted as a single symbol This is essential to the C language.
<i>XPG3</i>	<i>X/Open Portability Guide</i> , Issue 3.
<i>XSI</i>	X/Open System Interface.



# International Supplement Manual for Advanced Users

## CONTENTS

1. INTRODUCTION . . . . .	1
2. SETTING UP THE ENVIRONMENT FOR USERS' TERMINALS . . . . .	2
2.1 Motivation . . . . .	2
2.2 Mapping Features . . . . .	3
2.3 The <code>ttymap</code> Program . . . . .	4
2.3.1 A Sample <code>mapfile</code> . . . . .	5
2.4 Activating Mapping Prior to Login . . . . .	6
2.4.1 The System Console . . . . .	6
2.4.2 Changing the Default Font for the Console . . . . .	6
2.4.3 Other Terminals . . . . .	7
2.4.4 User-Specific Configuration . . . . .	7
2.4.5 General <code>ttymap</code> Guidelines . . . . .	8
3. SPECIFYING DATE AND TIME FORMATS . . . . .	10
3.1 When to Use the Date and Time <code>locale</code> Category . . . . .	10
3.2 Date and Time Formatting . . . . .	10
3.3 Creating a Date and Time Formatting Definition . . . . .	11
3.3.1 <code>abday</code> Keyword . . . . .	12
3.3.2 <code>day</code> Keyword . . . . .	12
3.3.3 <code>abmon</code> Keyword . . . . .	12
3.3.4 <code>mon</code> Keyword . . . . .	13
3.3.5 <code>d_t_fmt</code> Keyword . . . . .	13
3.3.6 <code>d_fmt</code> Keyword . . . . .	13
3.3.7 <code>t_fmt</code> Keyword . . . . .	13
3.3.8 <code>am_pm</code> Keyword . . . . .	13
3.3.9 <code>t_fmt_ampm</code> Keyword . . . . .	14
3.3.10 A Sample File . . . . .	14
3.3.11 How a Program Uses This Information . . . . .	14
4. SPECIFYING CHARACTER CLASSIFICATION INFORMATION . . . . .	15

4.1	Defining Character Classification . . . . .	15
4.2	When to Use the Character Classification <code>locale</code> Category . . . . .	15
4.3	Creating a Character Classification Category Definition . . . . .	15
4.3.1	An Example of a Character Classification Definition . . . . .	17
4.3.2	How a Program Uses This Information . . . . .	17
4.3.3	Use in Regular Expressions and Shell Pattern Matching . . . . .	17
5.	PREPARING AND INSTALLING A COLLATION SEQUENCE . . . . .	19
5.1	When to Use a Collation Sequence . . . . .	19
5.2	Defining Collation . . . . .	19
5.3	Capabilities . . . . .	20
5.4	Creating a Collation Sequence Definition . . . . .	21
5.4.1	<code>charmap</code> Files . . . . .	23
5.5	Source File Organisation . . . . .	24
5.5.1	<code>collating-element</code> Keyword . . . . .	25
5.5.2	<code>collating-symbol</code> Keyword . . . . .	25
5.5.3	<code>substitute</code> Keyword . . . . .	25
5.5.4	<code>order_start</code> Keyword . . . . .	26
5.5.5	<code>order_end</code> Keyword . . . . .	29
5.5.6	An Example . . . . .	30
5.5.7	Use in Regular Expressions and Shell Pattern Matching . . . . .	32
6.	SPECIFYING NUMERIC AND MONETARY INFORMATION . . . . .	33
6.1	Reasons for Defining Numeric and Monetary Formatting . . . . .	33
6.2	Defining Numeric and Monetary Formatting . . . . .	33
6.3	When to Use the Numeric and Monetary <code>locale</code> Category . . . . .	33
6.4	Numeric Editing . . . . .	33
6.5	Creating a Numeric Category Definition . . . . .	34
6.5.1	<code>decimal_point</code> Keyword . . . . .	35
6.5.2	<code>thousands_sep</code> Keyword . . . . .	35
6.5.3	<code>grouping</code> Keyword . . . . .	35
6.5.4	An Example of a Numeric Category Definition . . . . .	36

6.5.5	How a Program Uses This Information . . . . .	36
6.6	Monetary Editing . . . . .	36
6.7	Creating a Monetary Category Definition . . . . .	36
6.7.1	int_curr_symbol Keyword . . . . .	38
6.7.2	currency_symbol Keyword . . . . .	38
6.7.3	mon_decimal_point Keyword . . . . .	39
6.7.4	mon_thousands_sep Keyword . . . . .	39
6.7.5	mon_grouping Keyword . . . . .	39
6.7.6	positive_sign/negative_sign Keywords . . . . .	39
6.7.7	int_frac_digits Keyword . . . . .	39
6.7.8	frac_digits Keyword . . . . .	39
6.7.9	p_cs_precedes/n_cs_precedes Keywords . . . . .	40
6.7.10	p_sep_by_space/n_sep_by_space Keywords . . . . .	40
6.7.11	p_sign_posn/n_sign_posn Keywords . . . . .	40
6.7.12	An Example of a Monetary Category Definition . . . . .	41
6.7.13	How a Program Uses This Information . . . . .	41
7.	<b>SPECIFYING YES/NO RESPONSE INFORMATION . . . . .</b>	<b>42</b>
7.1	Reasons for Defining Yes/No Responses . . . . .	42
7.2	Defining Yes/No Responses . . . . .	42
7.3	When to Use the Yes/No Response locale Category . . . . .	42
7.4	Creating a Yes/No Response Category Definition . . . . .	42
7.4.1	yesexpr Keyword . . . . .	43
7.4.2	noexpr Keyword . . . . .	43
7.4.3	An Example of a Response Category Definition . . . . .	43
7.4.4	How a Program Uses This Information . . . . .	43
8.	<b>TIPS FOR PROGRAMMERS . . . . .</b>	<b>44</b>
8.1	Character Mapping . . . . .	44
8.2	Giving Programs Access to locales . . . . .	45
8.3	Date and Time . . . . .	45

8.4	Character Classification . . . . .	46
8.5	Collation . . . . .	46
8.6	Regular Expressions . . . . .	46
8.7	Numeric and Monetary Formatting . . . . .	46
8.8	Message Catalogues . . . . .	47
8.8.1	Extension of printf Syntax . . . . .	48

# International Supplement Manual for Advanced Users

## 1. INTRODUCTION

This document explains how to prepare and install a properly functioning international environment on an INTERACTIVE UNIX\* Operating System. It also summarizes the internationalisation features and provides tips for C programmers who want to develop internationalised applications. Developers of such applications should also consult the *X/Open Portability Guide*.

Note that before reading this document, you should have already read the "International Supplement User's Manual."

## 2. SETTING UP THE ENVIRONMENT FOR USERS' TERMINALS

This section describes how a system administrator can configure the terminals on the system to use the appropriate codesets and the keyboards supported by those terminals. It also explains the need for character mapping ability and give tips for establishing the correct mapping from boot time.

### 2.1 Motivation

The original UNIX Operating System and most systems derived from it have been based on the ASCII 7-bit coded character set and American English. The ASCII character set consists of 128 different characters, each represented by a single byte (the eighth bit is not used). Beginning with UNIX System V Release 3.1, most applications have been modified to properly support characters represented as a byte with the eighth bit set as well. This means that now 256 characters can be supported at the same time. A consistent coding convention needs to be applied, however. In the IBM\* PC world, an 8-bit coding scheme referred to as IBM extended ASCII has been used for several years. This codeset is currently referred to as IBM codepage 437. In heterogeneous UNIX System environments a different codeset, called ISO 8859-1, has been promoted. Both of these codesets are supersets of ASCII.

Although an 8-bit system meets most of the European requirements (for the major Asian Languages, a 16-bit system is necessary even to support a single language), it should function properly in conjunction with the available hardware and, in particular, with the terminals. To use characters from the French, German, Finnish, and other alphabets, several terminals are available that generate 7-bit codes but display the characters from those alphabets on the screen instead of the ones found on a U.S. terminal. Their keyboards have the same number of keys, but different characters are pictured on the key caps. Others, like the DEC\* VT220\*, support 256 characters at a time but use their own proprietary codeset and have an extra **COMPOSE** key.

To illustrate the problems that occur when trying to use such terminals in a mixed language environment, imagine an INTERACTIVE UNIX System with a console and a French 7-bit terminal connected to the serial port. When editing a file on the terminal and using the French character é in text, the terminal (hardware) actually generates the ASCII code 123, which is the code normally used for the



left curly brace (`{`). (This example assumes that the terminal uses the French national variant of ASCII called ISO 646f.) If the file that was edited is looked at on the console, the letter actually appears to be a curly brace. Therefore input and output mapping should be supported by the `tty` subsystem to allow consistent use of one single codeset throughout the system.

Implementing character mapping support inside the `tty` subsystem has the advantage that its features are automatically supported by all peripherals that use the standard line discipline, without modifying the device drivers for these peripherals.

## 2.2 Mapping Features

For each `tty` device, character mapping can be done on input as well as on output. The information is stored in a buffer, the size of which should not exceed 1K. The following mapping features are supported:

- *Input mapping*

On input, any byte can be mapped to any byte. Using the example from the previous section, 123 could be mapped to 130, the code used for `é` in the IBM extended ASCII codeset, or C9, its equivalent in the ISO 8859-1 codeset.

- *Output mapping*

On output, any byte can be mapped to either a byte or a string. In the previous example, 130 or C9 would be mapped back to 123 to properly display the character on the screen. If the connected device is a printer that does not support the `é` character, it can be mapped into the string `e BACKSPACE`.

- *Deadkeys*

Certain keys on typewriters behave differently from the others, because when these keys are pressed, the carriage of the typewriter does not move. `^` is such a character, for example. When it is followed by an `e`, the letter `ê` is generated. This is called a deadkey or a non-spacing character. The `tty` subsystem supports the use of deadkeys. Typically, the `^` character and the umlaut character are used as deadkeys.

- *Compose sequences*

Characters can also be generated using compose sequences. A dedicated character, called the `compose character`, followed by two other keystrokes, generates a single character. As an example, `[COMPOSE]` followed by the plus sign and the minus

sign could generate the plus/minus sign. Compose sequences can also be used as an alternative for deadkeys, for example, `COMPOSE ^ e` instead of `^ e` alone.

- *Decimal representation*

Rarely used characters can be generated by pressing `COMPOSE`, followed by three digits (which are the decimal representation of the character). This feature has been added by INTERACTIVE. This should alleviate most of the inconvenience caused by the 1K limitation of the mapping buffer.

- *Toggle key*

An optional toggle key can be defined to temporarily disable the current mapping at any time. This can be useful when a German programmer wants easy access to the curly braces and the brackets. A toggle key is also used by Greek users to switch between ASCII and Greek. The toggle key feature and the `ioctl` calls that implement this are INTERACTIVE enhancements.

## 2.3 The `ttymap` Program

`ttymap` is an INTERACTIVE utility that permits a user to activate character mapping for the user's terminal on input and output. This utility can be used for regular terminals as well as for scancode devices such as the AT\* console. It makes full use of all the features of the terminal (`tty`) driver and the keyboard display driver that support such mapping.

The keyboard of the console differs from the keyboards used with regular terminals in two ways: they contain a number of keys, such as the `ALT` key, that are not found on regular terminals, and they generate *scancodes* rather than ASCII or extended ASCII codes. Scancodes generated by PC keyboards typically represent the location of the key on the keyboard; the keyboard driver has to properly translate these scancodes. Without changing the scancode translation, if French users type an A, they see a Q on the screen. Several status keys can influence the translated code as well. The keyboard driver, and thus the `ttymap` program, make a distinction between two sets of key combinations that can be translated:

- *Regular keys*
- *Function keys*

Up to 60 key combinations are recognised as function keys. The first 12 are the 12 function keys of a 101-key PC keyboard.

**F13** to **F24** are the same keys used in combination with **SHIFT**, **F25** to **F36** when used with **CTRL**, and **F37** to **F48** when used with **CTRL** and **SHIFT** together. **F49** to **F60** are the keys on the numeric keypad.

On the console, it is more flexible to change the scancode translation than to use the general mapping features described earlier. It also reduces the risk of reaching the 1K limit of the mapping buffer.

*ttymap(1)* describes how the desired mapping should be laid out in a mapfile.

### 2.3.1 A Sample mapfile

Consider the following input to the *ttymap* program:

```
# sample file
input:
#
toggle: 0x14      # CTRL SHIFT F2
#
dead: ``         # circumflex
      ``         # <circumflex>
      ``         # <e-circumflex>
'e' 0x88         # <e-circumflex>
#
# compose key
#
compose: 0x18     # CTRL SHIFT F1
'e' ':' 0x89     # <e-diaeresis>
#
output:
'^U'             'K'I'L'L'
scancodes:
# map CTRL SHIFT F1 to be 0x18 for the compose character key
F37      0x18
# map CTRL SHIFT F2 to be 0x14 for the toggle key
F38      0x14
```

This file defines the compose and toggle keys, two deadkey sequences, one compose sequence, and “KILL” as the string to be displayed whenever **^U** is sent to the output.

Assuming this file is named *mapfile*, this mapping could be activated by typing:

```
ttymap mapfile
```

The terminal currently in use will then behave according to the mapping described. This has its drawbacks, however, for users with a French keyboard. For example, if a user with the login name *paul* can only use the keyboard correctly after typing this command, he is then forced to type *pqu1* to log in to the system, has to have chosen a password that can still be typed in, and has to type:

```
tty;qp ;qpfiler
```

to access the `ttymap` command itself.

To avoid this awkward situation, INTERACTIVE has enhanced the `getty` command to activate the mapping prior to login. A new option, `-m`, has been added. Refer to section 2.4 and `getty(1M)` for details.

## 2.4 Activating Mapping Prior to Login

### 2.4.1 The System Console

When the INTERACTIVE UNIX System is installed, the system asks for keyboard information. This automatically configures the system for the proper mapping on the console for the keyboard selected (providing IBM codepage 437 is used).

### 2.4.2 Changing the Default Font for the Console

When the system is booted, IBM codepage 437 is automatically used on the console. The system can be configured to automatically use a different font, without the need for any additional commands from the user.

To do this, create a shell script with a name that starts with `S` and a number (for example, `S95font`), with the appropriate `loadfont` command replacing the one in this example:

```
# set the appropriate loadfont

/usr/bin/loadfont 8859
```

Place this file in the directory `/etc/rc2.d`, which contains a number of shell scripts that are automatically executed when the system comes up in multi-user mode. The order of execution depends on the number in the file name. We recommend using a number greater than all the others for the script that changes the font. The directory also contains files with names that begin with the letter `K`; these are executed when the system is switched back to single-user mode. For example, this directory might contain:

<code>K36sendmail</code>	<code>S06TMPRAMD</code>	<code>S21perf</code>
<code>S01MOUNTFSYS</code>	<code>S11uname</code>	<code>S70uucp</code>
<code>S05RMTMPFILES</code>	<code>S20syssetup</code>	<code>S95font</code>

### 2.4.3 Other Terminals

When the system is booted, a `getty` program is started on every terminal that is configured in the system. This program prints `login:` or any other “herald” on the screen and waits until someone types input. It then calls the `login` program for password verification, which in turn executes the user’s login program, which is typically the UNIX System command interpreter, the shell.

Each such terminal is represented by one line in the system file `/etc/inittab`. By modifying such a line, mapping can be activated prior to logging in on any terminal. For example, a line for the console would be:

```
co:12345:respawn:/etc/getty -m /usr/lib/keyboard/437/en_US console console
```

To activate mapping on another terminal, simply add the `-m` option, followed by the name of the appropriate mapping file to the `getty` command on the line representing the terminal. Most terminal devices have a name that contains the string `tty`. For example:

```
00:2345:off:/etc/getty /dev/tty00 9600
```

represents the first serial port of the computer. To test the new configuration, first kill any existing `getty` processes for the devices with entries that have been changed, then as superuser, type:

```
# telinit q
```

This has the system reread the `/etc/inittab` file. This file is recreated each time a new UNIX System kernel is built, using information stored in other files. Therefore, one more step needs to be taken after the terminal setup has been successfully tested. Add the same line with `getty -m` to either `/etc/conf/cf.d/init.base` (the base `inittab` file that contains information about the console) or the file in the directory `/etc/conf/init.d` that corresponds to the device driver of the peripheral to which the terminal is attached (for example, `asy` for the serial port).

### 2.4.4 User-Specific Configuration

The configuration guidelines given in the previous section assume that all users of a particular terminal use the system in the same fashion. This may not always be the case. A French user using a U.S. terminal may want to see a circumflex defined as a deadkey; an American user would not. If this is the case, you can add the appropriate `loadfont` or `ttymap` commands to the user’s

`$HOME/.profile` file for Bourne Shell users or to the appropriate user-specific configuration files for other shells. These commands override the system-wide configuration.

#### 2.4.5 General `ttymap` Guidelines

INTERACTIVE supplies `ttymap` files for the console to support all major keyboard types. These files are delivered with the INTERACTIVE UNIX Operating System in the `/usr/lib/keyboard` directory and are named `*.map`. A number of other `ttymap` files and font files (which have names with the suffix `.bdf`, for example, `vga855.bdf`), some of which have been supplied to INTERACTIVE by third parties, are distributed with the International Supplement on an as-is basis. The `ttymap` files include:

Language/ Territory	Codesets						
	437	850	863	865	866	8859-1	8859-5
da_DK	x	x		x		x	
de_CH	x	x				x	
de_DE	x	x				x	
en_UK	x	x				x	
en_US	x	x				x	
es_ES	x	x				x	
fr_CA			x			x	
fr_CH	x	x				x	
fr_FR	x	x				x	
it_IT	x	x				x	
no_NO	x	x		x		x	
ru_RU					x		x
sv_SE	x	x				x	

These files are located in directories under the `/usr/lib/keyboard` directory that represent the codeset (437, 850, 863, and so on) and are named for the *language\_territory*, `de_DE`, for example.

In many cases, the experienced user or the system administrator needs to create or modify an existing `mapfile` to support a specific terminal or environment. The following categories determine how the mapping should be configured:

- The type of terminal used.
- The codeset used.
- The layout of the keyboard used.
- The country it is used in, or the language spoken by the user.

Each time one of these categories changes, a different `ttymap` file is required.

### 3. SPECIFYING DATE AND TIME FORMATS

Date and time formatting consists of rules that define how date and time strings appear. These rules are created by placing specifications in the `LC_TIME` file in a `locale` directory.

The default conventions for the date and time format, as well as the names for the days of the week and the months, follow the U.S. conventions and are rarely applicable in other countries. By defining and using the date and time `locale` category, you can ensure that the dates and times displayed by the system follow your conventions and use the local names of days and months.

#### 3.1 When to Use the Date and Time `locale` Category

A created and installed definition is not activated until the user specifies that it should be used. To do this, set the `LC_ALL`, `LC_TIME`, or `LANG` environment variable to the directory in which the files are stored. This must be done before a program using the stored definitions is executed. Note that the program must be set up to check and set the international environment (via the `setlocale` function). In the INTERACTIVE UNIX Operating System, the standard utilities that display the date and time, such as `date` and `ls`, have been modified to use the international environment.

#### 3.2 Date and Time Formatting

Date and time formatting controls the appearance of date and time strings created by the system. The following aspects of formatting are controlled via the `LC_TIME` `locale` category:

- Format of the time display.
- Format of the date display.
- Format of the combined date and time display.
- Format of the 12-hour time display.
- Names of the days of the week.
- Abbreviated names of the days of the week.
- Names of the months.
- Abbreviated names of the months.
- Format of the ante meridiem and post meridiem strings used in 12-hour clock time displays.



Note that the standard INTERACTIVE UNIX System library routine `strftime` (refer to `ctime(3P)`) is set up to use this information. The System V `cftime` routine, on the other hand, does not use the information created in this manner; it uses a different shell variable and searches in a different directory (refer to section 9, “THE SYSTEM V ENVIRONMENT,” in the “International Supplement User’s Manual” for more information).

### 3.3 Creating a Date and Time Formatting Definition

The source language for the date and time category in the INTERACTIVE UNIX System is the language defined by the POSIX.2 group for the `LC_TIME` locale category.

A date and time editing source definition consists of a header, a date and time editing body, and a trailer. The header consists of the word `LC_TIME`. The trailer consists of the string `END LC_TIME`.

The date and time editing body consists of one or more lines of text. Each line contains a keyword followed by one or more operands. Keywords are separated from the operands by one or more blank characters (space or tab).

Operands are characters, strings of characters, or digits. When a keyword is followed by more than one operand, the operands must be separated by semicolons (;). Blanks are allowed before and/or after a semicolon. Strings must be surrounded by quotes. Individual characters may be surrounded by quotes, but it is not required. Blank lines or lines containing a number sign (#) in the first column are ignored. A line can be continued by typing a backslash (\) as the last character on the line.

The following keywords are recognised:

<code>LC_TIME</code>	The header.
<code>abday</code>	Defines the abbreviated names of the weekdays, starting with Sunday.
<code>day</code>	Defines the names of the weekdays, starting with Sunday.
<code>abmon</code>	Defines the abbreviated names of the months, starting with January.
<code>mon</code>	Defines the names of the months, starting with January.

<code>t_fmt</code>	Defines the format of the time string.
<code>d_fmt</code>	Defines the format of the date string.
<code>d_t_fmt</code>	Defines the format of the combined date and time string.
<code>am_pm</code>	Defines the strings used to specify ante meridiem and post meridiem in a time string according to the 12-hour clock.
<code>t_fmt_ampm</code>	Defines the format of the 12-hour time display.
<code>END LC_TIME</code>	The trailer.

Refer to *date(1)* for more information about date field descriptors.

### 3.3.1 *abday* Keyword

This keyword defines the abbreviated weekday names, corresponding to the `date %a` field descriptor. The operand must consist of seven strings, separated by semicolons. The first string must be the abbreviated name of the first day of the week (Sunday), the second string must be the abbreviated name of the second day, and so on. For example:

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

### 3.3.2 *day* Keyword

This keyword is used to define the full weekday names, corresponding to the `date %A` field descriptor. The operand must consist of seven strings, separated by semicolons. The first string must be the full name of the first day of the week (Sunday), the second string must be the full name of the second day, and so on. For example:

```
day "Sonntag"; "Montag"; "Dienstag"; \
    "Mittwoch"; "Donnerstag"; "Freitag"; "Samstag"
```

### 3.3.3 *abmon* Keyword

This keyword is used to define the abbreviated month names, corresponding to the `date %b` field descriptor. The operand must consist of twelve strings, separated by semicolons. The first string must be the abbreviated name of the first month of the year (January), the second string must be the abbreviated name of the second month, and so on. For example:

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

### 3.3.4 mon Keyword

This keyword is used to define the full month names, corresponding to the `date %B` field descriptor. The operand must consist of twelve strings, separated by semicolons. The first string must be the full name of the first month of the year (January), the second the full name of the second month, and so on. For example:

```
mon "Januar"; "Februar"; "März"; "April"; \
    "Mai"; "Juni"; "Juli"; "August"; \
    "September"; "Oktober"; "November"; "Dezember"
```

### 3.3.5 d\_t\_fmt Keyword

This keyword is used to define the appropriate date and time representation, corresponding to the `date %c` field descriptor. The operand must consist of a string and may contain any combination of characters and `date` field descriptors. In addition, the string may contain the `date %n` and `%t` field descriptors for newline and tab characters, respectively. For example:

```
d_t_fmt "%a %b %d %H:%M:%S %Y"
```

### 3.3.6 d\_fmt Keyword

This keyword is used to define the appropriate date representation, corresponding to the `date %x` field descriptor. The operand must consist of a string and may contain any combination of characters and date field descriptors. For example:

```
d_fmt "%m/%d/%y"
```

### 3.3.7 t\_fmt Keyword

This keyword is used to define the appropriate time representation, corresponding to the `date %X` field descriptor. The operand must consist of a string and may contain any combination of characters and date field descriptors. For example:

```
t_fmt "%H:%M:%S"
```

### 3.3.8 am\_pm Keyword

This keyword is used to define the appropriate representation of the ante meridiem and post meridiem strings, corresponding to the `date %p` field descriptor. The operand must consist of two strings, separated by a semicolon. The first string must represent the ante meridiem designation; the last string, the post meridiem designation. For example:

```
am_pm    "AM";"PM"
```

### 3.3.9 t\_fmt\_ampm Keyword

This keyword is used to define the appropriate time representation in the 12-hour clock format with `am_pm`, corresponding to the `date %r` field descriptor. The operand must consist of a string and may contain any combination of characters and date field descriptors. If this keyword is not defined, the default (`%I:%M:%S %p`) is used. For example:

```
t_fmt_ampm "%I.%M.%S %p"
```

### 3.3.10 A Sample File

```
LC_TIME
#
#
abday      "Son";"Mon";"Die"; \
           "Mit";"Don";"Fre";"Sam"
day        "Sonntag";"Montag";"Dienstag"; \
           "Mittwoch";"Donnerstag";"Freitag";"Samstag"
abmon      "Jan";"Feb";"März";"Apr"; \
           "Mai";"Juni";"Juli";"Aug"; \
           "Sept";"Okt";"Nov";"Dez"
mon        "Januar";"Februar";"März";"April"; \
           "Mai";"Juni";"Juli";"August"; \
           "September";"Oktober";"November";"Dezember"
d_t_fmt    "%I.%M.%S %p %m/%d/%y"
d_fmt      "%m/%d/%y"
t_fmt      "%I.%M.%S %p"
am_pm      "VM";"NM"
t_fmt_ampm "%I.%M.%S %p"
#
END LC_TIME
```

### 3.3.11 How a Program Uses This Information

If a program needs to access the values in the current locale, it can do so via the library subroutine `nl_langinfo`, as well as by using the definition via the `strftime` library subroutine (refer to `ctime(3P)`). Refer to section 8, "TIPS FOR PROGRAMMERS," for more information.

## 4. SPECIFYING CHARACTER CLASSIFICATION INFORMATION

The character classification category determines classification of characters as letters, digits, and so on, as well as some other information about the codeset and character set used. The default character classification only recognises the 26 ASCII letters as such, which means that any program processing non-English text that depends on the classification will behave incorrectly. For example, take `vi`, which prints nonprintable characters using an octal notation. For `vi` to correctly display non-ASCII characters, you must change the character classification. Another example is programs that do uppercase to lowercase conversion; the standard table handles only ASCII.

### 4.1 Defining Character Classification

These definitions are created by placing a specification in the `LC_CTYPE` file in a `locale` directory. This specification is output by the `chrtbl` utility (refer to `chrtbl(1M)`). The created table should also be copied to the `/lib/chrclass` directory.

### 4.2 When to Use the Character Classification `locale` Category

The created and installed definitions are not activated until the user specifies that they should be used. To do this, the user must set the `LC_ALL`, `LC_CTYPE`, or `LANG` environment variable to the directory in which the files are stored. This must be done before a program using the stored definitions is executed. Note that the program must be set up to check and set the international environment (via the `setlocale` function). In the INTERACTIVE UNIX System, the standard utilities that depend on character classification, such as `grep`, `ls`, `ed`, and `sort`, have been modified to use the international environment. However, the `vi` program has not been modified to use the international environment; it uses the information in the `/lib/chrclass` directory and the value of the environment variable `CHRCLASS`. Refer to section 9, “THE SYSTEM V ENVIRONMENT,” in the “International Supplement User’s Manual” for more information.

### 4.3 Creating a Character Classification Category Definition

Character classification definitions are created using the `chrtbl` utility. The source language for the character classification category in the INTERACTIVE UNIX Operating System allows the

user to define the name of the data file created by `chrtbl`, the assignment of characters to character classifications, and the relationship between uppercase and lowercase letters. The character classifications recognised by `chrtbl` are:

<code>chrclass</code>	Name of the data file to be created by <code>chrtbl</code> .
<code>isupper</code>	Character codes to be classified as uppercase letters.
<code>islower</code>	Character codes to be classified as lowercase letters.
<code>isdigit</code>	Character codes to be classified as numeric.
<code>isspace</code>	Character codes to be classified as spacing (delimiter) characters.
<code>ispunct</code>	Character codes to be classified as punctuation characters.
<code>isctrl</code>	Character codes to be classified as control characters.
<code>isblank</code>	Character code for the space character.
<code>isxdigit</code>	Character codes to be classified as hexadecimal digits.
<code>ul</code>	Relationship between uppercase and lowercase characters.

Any lines with a number sign (#) in the first column are treated as comments and are ignored. Blank lines are also ignored.

A character can be represented as a hexadecimal or octal constant (for example, the letter *a* can be represented as `0x61` in hexadecimal or `0141` in octal). Hexadecimal and octal constants may be separated by one or more space or tab characters.

The dash character (-) can be used to indicate a range of consecutive numbers. Zero or more space characters may be used for separating the dash character from the numbers. The backslash character (\) is used for line continuation. Only a carriage return is permitted after the backslash character.

The relationship between uppercase and lowercase letters, `ul`, is expressed as ordered pairs of octal or hexadecimal constants: `<uppercase_character lowercase_character>`. These two constants may be separated by one or more space characters. Zero or

more space characters may be used for separating the angle brackets (< >) from the numbers.

### 4.3.1 An Example of a Character Classification Definition

The following is an example of an input file:

```
chrclass      LC_CTYPE
isupper       0x41 - 0x5a
islower       0x61 - 0x7a
isdigit       0x30 - 0x39
isspace       0x20 0x9 - 0xd
ispunct       0x21 - 0x2f  0x3a - 0x40  \
              0x5b - 0x60  0x7b - 0x7e
iscntrl       0x0 - 0x1f  0x7f
isblank       0x20
isxdigit      0x30 - 0x39  0x61 - 0x66  \
              0x41 - 0x46
ul            <0x41 0x61> <0x42 0x62> <0x43 0x63>  \
              <0x44 0x64> <0x45 0x65> <0x46 0x66>  \
              <0x47 0x67> <0x48 0x68> <0x49 0x69>  \
              <0x4a 0x6a> <0x4b 0x6b> <0x4c 0x6c>  \
              <0x4d 0x6d> <0x4e 0x6e> <0x4f 0x6f>  \
              <0x50 0x70> <0x51 0x71> <0x52 0x72>  \
              <0x53 0x73> <0x54 0x74> <0x55 0x75>  \
              <0x56 0x76> <0x57 0x77> <0x58 0x78>  \
              <0x59 0x79> <0x5a 0x7a>
```

### 4.3.2 How a Program Uses This Information

Programs access this information by using the character classification and conversion library interfaces (refer to *ctype(3C)*). As *vi* does not use the information via the *locale*, we recommend that the table also be copied to the */lib/chrclass* directory and given the same name as the *locale*.

### 4.3.3 Use in Regular Expressions and Shell Pattern Matching

The information in the character classification definition can be directly used in regular expressions, via the character class syntax inside a bracket expression. The syntax is:

```
[ :class-name: ]
```

where *class-name* is the name of one of the following:

```
alpha      a letter
upper      an uppercase letter
lower      a lowercase letter
digit      a decimal digit
xdigit     a hexadecimal digit
```

<b>a l n u m</b>	an alphanumeric (letter or digit)
<b>s p a c e</b>	a character that produces white space in displayed text
<b>p u n c t</b>	a punctuation character
<b>p r i n t</b>	a printing character
<b>g r a p h</b>	a character with a visible representation
<b>c n t r l</b>	a control character

For example, the following command will find all file names in the current directory that begin with an uppercase letter:

```
ls "[[:upper:]]*"
```

These specifications are primarily intended to replace the current use of expressions like [A-Z], which are not portable (Z is not the last letter in all alphabets).



## 5. PREPARING AND INSTALLING A COLLATION SEQUENCE

A collation sequence specifies how characters and collating elements should be sorted, that is, the order between characters and collating elements. Collation sequences are created using the `colldef` processor (refer to `colldef(1P)` for more information). This section describes how to set up a source collation sequence definition and use it to create a collation sequence. Once the source definition is created and tested, you can use it to create “object” collation sequences, which are stored in a file named `LC_COLLATE` in a `locale` directory.

### 5.1 When to Use a Collation Sequence

A created and installed collation sequence definition is not activated until the user specifies that it should be used. To do this, set the `LC_ALL`, `LC_COLLATE`, or `LANG` environment variable to the directory in which the files are stored. This must be done before a program using the stored definitions is executed. Note that the program must be set up to check and set the international environment (via the `setlocale` function).

User-defined collation is supported through the `colldef` utility and the library functions `strxfrm` and `strcoll` (refer to `strxfrm(3P)` and `strcoll(3P)` for more information). These functions are used to compare strings based on the defined collation order and rules. Traditional programs that need to do sorting use `strcmp`, which does byte-to-byte comparison. In the INTERACTIVE UNIX Operating System, the standard utilities that depend on collation, such as `sort` and `ls`, have been modified to use the international environment (refer to `string(3P)` for more information).

### 5.2 Defining Collation

Collation, according to a dictionary, is the “act of putting things in their proper order.” Collation rules define how the data are put in the proper order, or *sorted*. Traditionally, the collating order in the UNIX System has been ASCII order, that is, the order in which the characters appear in the ASCII codeset. This is also the natural collating order for the English language.

For most languages in the world, however, this is not enough. Most European languages contain more letters than the 26 in the English language, with the additional letters typically collating between the letters in the ASCII set. For example, an `á` sorts between `a` and `b`.

The European user expects sorted lists (for instance, the output from the `ls` command) to appear in the collation order of his or her language.

The INTERACTIVE UNIX Operating System provides users with the ability to define their own collation order. This capability is a superset of the X/Open\* requirement for an internationalised system, and it is expected to satisfy the requirements for dictionary ordering for most European languages and non-European alphabetic languages.

### 5.3 Capabilities

The following capabilities are provided:

1. *Multicharacter collating elements.*

The term *collating element* is used to describe the basic entities that are compared in collation. All characters in the character set are automatically collating elements. In addition, the user can define multicharacter collating elements (sequences of two or more characters to be collated as a single entity). For example, the Spanish `ch` collates as an entity between `c` and `d`.

2. *User-defined ordering of collating elements.*

The user has complete control over the order in which characters (and multicharacter collating elements) are sorted.

3. *Multiple weights and equivalence classes.*

For many languages, the basic ordering is sufficient, but others require more complex rules. For example, in German, the `ö` and the `o` collate as the same character, but if two words are equal except for the `o` and the `ö`, then the word with `o` comes first. In French, all accented letters collate equally with the base character; if the words are equal, there is a defined "secondary ordering" among these characters. All characters (or collating elements) that initially collate equally are said to belong to an *equivalence class*. Such characters typically have more than one "weight." The first (primary) weight is that of the equivalence class; the second weight is determined by their relative order. The INTERACTIVE UNIX System supports up to `{COLL_WEIGHTS_MAX}` (defined as 4 in `/usr/include/sys/limits.h`) different weights for each character or collating element.

4. *One-to-many mapping.*

A single character is mapped into a string of collating elements. An example of this is the German  $\beta$ , which collates as `ss`.

5. *Many-to-many substitution.*

A string is substituted for another string of one or more characters. The string that is substituted can be an empty string. In other words, the character or characters are ignored for collation purposes.

6. *Ordering by weights.*

To determine their relative order, two strings are first compared based on the primary weight. If they are equal, and more than one weight has been assigned, then the strings are compared again and again until the strings either compare unequally or the weights are exhausted. Comparisons may proceed either from the beginning of the strings toward the end, or from the end toward the beginning.

## 5.4 Creating a Collation Sequence Definition

The source language for collation definitions in the INTERACTIVE UNIX System is the language specified by the POSIX.2 group for the `LC_COLLATE` locale category.

A collation sequence definition describes the relative order among collating elements (characters and multicharacter collating elements) in the locale. This order is expressed in terms of collation values or weights by assigning each element one or more collation values. The collation sequence definition is used by regular expressions, pattern matching, and sorting.

A collation source definition consists of a collation header, a collation body, and a collation trailer. The collation header is the word `LC_COLLATE`. The collation trailer is the string `END LC_COLLATE`.

The collation body consists of one or more lines of text, each of which contains an identifier, optionally followed by one or more operands. Identifiers are either keywords or collating elements. Identifiers are separated from the operands by one or more blank characters (space or tab).

Operands are characters, collating elements, or strings of characters. When a keyword is followed by more than one operand, the

operands must be separated by semicolons; blanks are allowed before and/or after a semicolon.

A line modifying the comment character (the default is #) can be inserted before the header. The format is:

```
comment_char new-comment-character
```

starting in the first column. Empty lines and lines containing the *new-comment-character* in the first position are ignored.

A line modifying the escape character (the default is a backslash, \) can also be inserted before the header. The format is:

```
escape_char escape-character
```

starting in the first column. A line can be continued by placing an escape character as the last character on the line. Comment lines cannot be continued on a subsequent line using an escaped newline character.

Individual characters, characters in strings, or collating elements can be represented in operands in any of the following formats:

1. *Symbolic notation.*

A character is specified via a symbolic character name, enclosed within angle brackets (< >). A symbolic name, including the angle brackets, must either be a symbol defined via a *collating-symbol* or *collating-element* keyword or must exactly match a symbolic name defined in the *charmap* file specified via the *colldef -f* option. It is not an error to specify a collating element via a *charmap* symbol that does not exist in the current *charmap* file (refer to *charmap(5P)*). The processor assumes that the definition is a “generic” one, intended for use with many codesets. Such a generic definition may contain characters not present in all codesets. Therefore, the *colldef* processor assumes that the character should simply be ignored and issues a warning message to that effect. Note that any escape character or right angle bracket in a symbolic name must be preceded by the escape character.

Using symbolic names rather than any other notation makes it possible to use the same source definition with several codesets. For example:

```
<c>;<a \>;<c-cedilla> "◀<a><y>"
```

2. *Character notation.*

A character is specified by the character itself. The quote, comma, semicolon, angle brackets, and escape character (" , ; < > and *escape-character*) must be escaped (preceded by the escape character) if they are found outside strings enclosed by double quotes; only the double quote must be escaped inside quoted strings. For example:

```
c;ç;â "May"
```

3. *Octal notation.*

An octal constant must be specified as the escape character, followed by two or three octal digits. For example:

```
\143;\347 "\115\141\171"
```

4. *Hexadecimal notation.*

A hexadecimal constant must be specified as the escape character, followed by an *x*, followed by one or two hexadecimal digits. For example:

```
\x63;\xe7 "\x4d\x61\x79"
```

5. *Decimal notation.*

A decimal constant must be specified as the escape character, followed by a *d*, followed by one, two, or three decimal digits. For example:

```
\d99;\d231 "\d77\d97\d121"
```

### 5.4.1 *charmap Files*

The `colldef` processor (as well as the `iconv` utility) can use the information stored in a `charmap` file. (Refer to `iconv(1P)` for more information.) These files are used to document the supported codesets. Each character in the coded character set is described with a symbolic name and the character encoding. The following is an excerpt from the `charmap` file describing IBM codepage 437. Refer to `charmap(5P)` for more information.

<C-cedilla>	\d128	LATIN CAPITAL LETTER C WITH CEDILLA
<u-diaeresis>	\d129	LATIN SMALL LETTER U WITH DIAERESIS
<e-acute>	\d130	LATIN SMALL LETTER A WITH ACUTE
<a-circumflex>	\d131	LATIN SMALL LETTER A WITH CIRCUMFLEX
<a-diaeresis>	\d132	LATIN SMALL LETTER A WITH DIAERESIS
<a-grave>	\d133	LATIN SMALL LETTER A WITH GRAVE
<a-ring>	\d134	LATIN SMALL LETTER A WITH RING ABOVE
<c-cedilla>	\d135	LATIN SMALL LETTER C WITH CEDILLA
<e-circumflex>	\d136	LATIN SMALL LETTER E WITH CIRCUMFLEX
<e-diaeresis>	\d137	LATIN SMALL LETTER E WITH DIAERESIS
<e-grave>	\d138	LATIN SMALL LETTER E WITH GRAVE
<i-diaeresis>	\d139	LATIN SMALL LETTER I WITH DIAERESIS
<i-circumflex>	\d140	LATIN SMALL LETTER I WITH CIRCUMFLEX
<i-grave>	\d141	LATIN SMALL LETTER I WITH GRAVE
<A-diaeresis>	\d142	LATIN CAPITAL LETTER A WITH DIAERESIS
<A-ring>	\d143	LATIN CAPITAL LETTER A WITH RING ABOVE
<E-acute>	\d144	LATIN CAPITAL LETTER E WITH ACUTE
<ae>	\d145	LATIN SMALL LETTER AE
<AE>	\d146	LATIN CAPITAL LETTER AE
<o-circumflex>	\d147	LATIN SMALL LETTER O WITH CIRCUMFLEX

## 5.5 Source File Organisation

The source file contains the following keywords, described in detail in the following sections:

### LC\_COLLATE

The header.

### collating-element

A `collating-element` keyword is used to specify multicharacter collating elements. This keyword is optional.

### collating-symbol

A `collating-symbol` keyword is used to specify collation symbols for use in collation order statements. This keyword is optional.

### substitute

Zero or more `substitute` keywords define mapping between strings. This keyword is optional.

### order start

This keyword is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.

### order end

This keyword terminates the collation order lines.

### END LC\_COLLATE

The trailer.

### 5.5.1 collating-element *Keyword*

Every character in the character set is also a collating element. If the language (or application) for which this collation sequence definition is intended also recognises multicharacter collating elements (such as the Spanish *ch*), these must be specified via a `collating-element` keyword. The syntax is:

```
collating-element symbol from string
```

The *symbol* operand must be a string of one or more characters, enclosed between angle brackets (< >), which cannot duplicate any symbolic name in the current `charmap` file or any other symbolic name defined in this collation definition. The *string* operand is a string of two or more characters to be collated as an entity. For example:

```
collating-element <ch> from <c><h>
collating-element <ss> from ss
```

### 5.5.2 collating-symbol *Keyword*

In addition to characters and multicharacter collating elements, you can also define special symbols for use in collation sequence statements, that is, between the `order_start` and the `order_end` keywords. Such a symbol does not have any character associated with it, as the `charmap` symbols do. However, placing such a symbol in the collating sequence assigns to it a relative order that can be used in other collation collating element specifications. The syntax is:

```
collating-symbol symbol
```

The *symbol* is a string of one or more characters, surrounded by angle brackets, which must not duplicate any symbolic name in the current `charmap` file or any other symbolic name defined in this collation definition. For example:

```
collating-symbol <UPPER_CASE>
collating-symbol <LOWER_CASE>
collating-symbol <NO_ACCENT>
collating-symbol <GRAVE>
collating-symbol <ACUTE>
```

### 5.5.3 substitute *Keyword*

The `substitute` keyword is used to define a substring substitution in a string to be collated. The syntax is:

```
substitute "regexp" with "rep"
```

The first operand is treated as a simple regular expression. The replacement operand consists of zero or more characters and regular expression backreferences (for example, \1 through \9).

When strings are collated based on a collation definition containing substitute statements, any substitutions are performed before strings are compared. For instance, if you have a substitute statement:

```
substitute "Mc" with "Mac"
```

and you compare the two strings McArthur and MacArthur, the substitute is first applied to both strings. As a result, the first string is replaced by MacArthur and the two strings compare as equals.

Ranges in the regular expression are interpreted according to the current character collation sequence, and character classes are interpreted according to the character classification specified via the LC\_CTYPE environment variable at collation time. If more than one substitute statement is present in the collation definition, the substitute statements are applied in the order in which they occur in the source definition.

Both operands must be enclosed within double-quotes (") or a null replacement is indicated by two adjacent double-quotes. For example:

```
substitute "Mc" with ""
```

#### 5.5.4 order\_start *Keyword*

The order\_start keyword precedes collation order entries and also defines the number of weights for this collation sequence definition and other collation rules.

The syntax of the order\_start keyword is:

```
order_start  sort-rules;sort-rules; ...
```

The operands to the order\_start keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands defines how many weights each element is assigned; if no operands are present, one *forward* operand is assumed. If present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second, when comparing strings using the second weight; and so on. Operands are separated by semicolons (;). Each operand consists of one or more collation directives, separated by commas (,). If the number of operands exceeds the {COLL\_WEIGHTS\_MAX}



limit, the utility ignores the operands in excess of the limit and issues a warning message. The following directives are supported:

- forward** Specifies that comparison operations for the weight level proceed from the beginning of the string to the end of the string.
- backward** Specifies that comparison operations for the weight level proceed from the end of the string to the beginning of the string.
- position** Specifies that comparison operations for the weight level will consider the relative position of non-IGNOREed elements in the string such that, if strings compare as equals, the element with the shortest distance from the starting point of the string is collated first.

The directives **forward** and **backward** are mutually exclusive. For example:

```
order_start forward;backward;forward
```

The absence of operands for this keyword is taken as a directive to perform comparisons on a character basis rather than on a string basis.

**5.5.4.1 Collation Order.** The **order\_start** keyword is followed by **collating-element** entries. The syntax for the **collating-element** entries is:

```
collating-element weight;weight; ...
```

Each **collating-element** consists of either a character (in any of the forms defined above), a **collating-element** symbol, a **collating-symbol** 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. The **NULL** character compares lower than any other character.

A **collating-element** symbol is used to specify multicharacter collating elements and indicates that the character sequence specified via the **collating-element** symbol is to be collated as a unit and in the relative order specified by its place. A **collating-symbol** symbol is used to define a position in the relative order for use in weights.

The ellipsis symbol (...) specifies that a sequence of characters collates according to their encoded character values, that is, all characters with a coded character set value higher than the value of the character in the preceding line and lower than the coded character set value for the character in the following line are placed in the character collation order between the previous and the following character in ascending order according to their coded character set values. An initial ellipsis is interpreted as if the line preceding it specified the NULL character, and a trailing ellipsis is interpreted as though the line following it specified the highest coded character set value in the current coded character set. An ellipsis is treated as invalid if the lines preceding or following it do not specify characters in the current coded character set. Note that the use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable. The `colldef` utility issues a warning to this effect if an ellipsis is detected. The explicit specification elsewhere of a character automatically included via an ellipsis symbol is treated as an error.

All characters not defined in the order sequence (either explicitly or via an ellipsis) are placed in the collation order via the special symbol `UNDEFINED`. All such characters are placed in to the existing order at the point of the `UNDEFINED` symbol, and ordered according to their coded character set values. If no `UNDEFINED` symbol is specified, and the current coded character set contains characters not specified in this clause, `colldef` issues a warning message and places such characters at the end of the character collation order.

The optional operands for each `collating-element` are used to define the primary, secondary, or subsequent weights for the `collating-element`. The first operand specifies the relative primary weight, the second the relative secondary weight, and so on. Two or more `collating-elements` can be assigned the same weight. They are said to belong to the same *equivalence class*. In string collation, each pair of strings is first compared based on primary weight. If equal, `collating-elements` belonging to primary equivalence classes are compared again based on their secondary weights. If still equal, secondary equivalence class elements are compared again based on tertiary weights, up to the limit `{COLL_WEIGHTS_MAX}`.

Weights must be expressed as characters (in any of the forms specified above), `collating-symbols`, `collating-`

elements, an ellipsis, or the special symbol IGNORE. A single character, a `collating-symbol` symbol, or a `collating-element` symbol represents the relative order in the character collating sequence of the character or symbol, rather than its absolute value. Multiple characters or symbols indicate one-to-many mapping.

The special symbol IGNORE means that this character is to be ignored at the defined weight level for collation purposes. For example, if the dash (-) is IGNORED, then the two strings

`co-ordinate`

and

`coordinate`

collate as equals. In regular expressions, such characters are never ignored. Ranges are based on the order in which elements are listed in the definition (basic character ordering sequence), and all characters are explicitly or implicitly listed.

All characters specified via an ellipsis are assigned unique weights and are ordered according to their coded character set values. Characters specified via an explicit or implicit UNDEFINED special symbol are by default assigned the same primary weight (that is, they belong to the same equivalence class). An ellipsis symbol as a weight is interpreted to mean that each character in the sequence must have unique weights, equal to the relative order of the character in the character collation sequence. Secondary and subsequent weights have unique values. The use of the ellipsis as a weight is treated as an error if the `collating-element` is neither an ellipsis nor the special symbol UNDEFINED.

An empty weight implies that the `collating-element` will be assigned a weight equal to the current position in the order. In other words, the `collating-element` “collates as itself.”

### 5.5.5 `order_end` Keyword

The `order_end` keyword terminates the ordering statements.

## 5.5.6 An Example

```

LC_COLLATE
#
collating-element <ch> from <c><h>                # See Note 1
collating-element <ss> from ss
#
collating-symbol <UPPER_CASE>
collating-symbol <LOWER_CASE>
collating-symbol <NO_ACCENT>
collating-symbol <GRAVE>
collating-symbol <ACUTE>
#
substitute "Mc" with "Mac"
#
order_start  forward;backward;forward
#
<UPPER_CASE>                # See Note 2
<LOWER_CASE>
<NO_ACCENT>
<GRAVE>
<ACUTE>
<space>
\...
<A>          IGNORE;IGNORE;IGNORE                # See Note 3
<a>          <A>;<UPPER_CASE>;<NO_ACCENT>         # See Note 4
<a-acute>    <A>;<LOWER_CASE>;<NO_ACCENT>
<a-grave>    <A>;<LOWER_CASE>;<ACUTE>
<B>
<b>
<C>          <C>;<C>;<C>                          # See Note 5
<C-cedilla> <C>;<C>;<C-cedilla>
<c>          <C>;<c>;<c>
<c-cedilla> <C>;<c>;<c-cedilla>
<ch>         <ch>;<ch>;<ch>                      # See Note 6
<S>          <S>;<S>;<S>
<s>          <S>;<s>;<s>
<ss>         <S><S>;<s><s>;<s><s>
<sharp-s>    <S><S>;<s><s>;<s><s>                # See Note 7
UNDEFINED    IGNORE;IGNORE;IGNORE                # See Note 8
order end
END LC_COLLATE

```

## Notes

1. The character sequences `ch` and `ss` are defined as collating elements.
2. The collating-symbols `<UPPER_CASE>`, `<LOWER_CASE>`, `<NO_ACCENT>`, `<GRAVE>`, and `<ACUTE>` are placed first in the ordering sequence, followed by the space symbol.
3. Characters with code values between `space` and `A` are placed in the basic ordering sequence after the space, but are ignored for collation purposes.

4. The accented and unaccented A's have the same primary weight, that is, they belong to an equivalence class. The secondary weight is based on case, but ignores accents. The third weight considers accents. This definition uses the collating symbols and their relative order (uppercase before lowercase, no accents before accents).

The definition can be viewed as a directive to transform strings by weight before comparing them. For example, when comparing the strings *abbà* and *Abba*, the two strings are first compared using the primary weight. This equates to comparing *ABBA* with *ABBA*, that is, they compare as equals. On secondary weighting, they compare as follows:

`<LOWER_CASE><LOWER_CASE><LOWER_CASE><LOWER_CASE>`

against:

`<UPPER_CASE><LOWER_CASE><LOWER_CASE><LOWER_CASE>`

The first collates after the second.

5. The accented and unaccented C's also belong to an equivalence class. Secondary ordering and tertiary ordering are defined using the characters themselves. The uppercase letters collate before the lowercase ones and the accented letters after the unaccented ones.

The two strings *Ça* and *Ca* first compare as *CA* versus *CA*. Based on secondary weights, they still compare as equals: (`C<LOWER_CASE>` versus `C<LOWER_CASE>`). On tertiary weight comparison, the two strings compare as `Ç<LOWER_CASE>` versus `C<LOWER_CASE>`, that is, the second compares lower.

6. The string *ch* compares as a single element. The string *Bach* consists of three collating elements and collates after the string *Back*.
7. The character *ß* (eszet or "sharp s") is a German character that collates as two "esses" (*ss*). This means that the two strings *Strasse* and *Straße* should collate as equals.
8. All characters not explicitly defined (or implicitly included via an ellipsis) are placed last in the collation sequence, in order according to their coded values. They are ignored for collation purposes.

### 5.5.7 Use in Regular Expressions and Shell Pattern Matching

The collation sequence determines how bracket expressions in regular expressions are interpreted:

1. All characters are valid in a bracket expression. Multicharacter collating elements (such as <ch> in the example above) are also recognised.
2. Multicharacter collating elements must be entered using a special "bracket-dot" syntax, for example, [ .ch. ], to distinguish the multicharacter element from the sequence "ch".
3. All characters belonging to an equivalence class can be referenced using the special "bracket-equal" syntax; [=a=] is shorthand for A, a, à á in the example above.
4. Range expressions are interpreted according to the basic character collation order, that is, the order in which the characters are listed in the definition. In the previous example, all characters not explicitly specified collate last via the UNDEFINED statement. This means that, using the previous example, [a-s] only specifies the characters in the list between a and s:

```
a à á B b c ç c ç "ch" s s
```

Likewise, a range such as [r-t] will not contain s.

5. To be able to find both "Strasse" and "Straße" in text with one expression, it is necessary to make ss into a collating element. Then, the following regular expression will find both strings: "Stra[[.ss.][.β.]]e".

## 6. SPECIFYING NUMERIC AND MONETARY INFORMATION

Numeric and monetary formatting determines how numeric and monetary items appear. This section explains how it can be used and how the files that contain the information should be set up.

### 6.1 Reasons for Defining Numeric and Monetary Formatting

The default conventions for decimal delimiter and other numeric formatting rules are seldom appropriate in an international environment. For example, the default decimal delimiter is a period, but in most European countries the comma is used instead. By defining numeric and monetary formatting with the correct values, programs display fractions using the appropriate decimal delimiter.

### 6.2 Defining Numeric and Monetary Formatting

These definitions are created by placing a specification in the appropriate file (either `LC_NUMERIC` or `LC_MONETARY`) in a `locale` directory.

### 6.3 When to Use the Numeric and Monetary `locale` Category

The created and installed definitions are not activated until the user specifies that they should be used. The user must set the `LC_NUMERIC` environment variable to the directory in which that file is stored and the `LC_MONETARY` environment variable to the directory in which that file is stored. Alternately, the user can set the `LC_ALL` or `LANG` environment variable to the directory to specify both. This must be done before a program using the stored definitions is executed. Note that the program must be set up to check and set the international environment (via the `setlocale` function). In the INTERACTIVE UNIX System, the standard utilities that depend on numeric editing, such as `awk`, have been modified to use the international environment.

### 6.4 Numeric Editing

Numeric editing controls the appearance of (nonmonetary) numbers, as well as the input format. The following three aspects of numeric editing are controlled via the `LC_NUMERIC` `locale` category:

1. The character used as a decimal delimiter.
2. The character used to separate groups of digits (thousands separator).
3. The size of such groups.

It should be noted that, while the standard INTERACTIVE UNIX System library subroutines `printf`, `scanf`, and `strtod` (refer to `printf(3P)`, `scanf(3P)`, and `strtod(3C)` for more information) are sensitive to the decimal delimiter, they do not support grouping of digits. Consequently, while user-developed functions can (and should) take into account grouping and thousands separators, the standard functions do not.

## 6.5 Creating a Numeric Category Definition

The source language for the numeric category in the INTERACTIVE UNIX System is the language defined by the POSIX.2 group for the `LC_NUMERIC` locale category.

A numeric editing source definition consists of a header, a numeric editing body, and a trailer. The header is the word `LC_NUMERIC`. The trailer is the string `END LC_NUMERIC`.

The numeric editing body consists of one or more lines of text. Each line contains a keyword followed by one or more operands. Keywords are separated from the operands by one or more blank characters (space or tab).

Operands are characters, strings of characters, or digits. When a keyword is followed by more than one operand, the operands must be separated by semicolons (;). Blank characters are allowed before and/or after a semicolon. Strings must be surrounded by quotes. Individual characters may be surrounded by quotes, but it is not required. Blank lines or lines containing a number sign (#) in the first column are ignored.

The following keywords are recognised:

`LC_NUMERIC`

The header.

`decimal_point`

Defines the decimal delimiter character.

`thousands_sep`

Defines the thousands separator character.



**grouping**

Defines the grouping of digits.

**END LC NUMERIC**

The trailer.

**6.5.1 decimal\_point Keyword**

This keyword specifies the character to use as the decimal delimiter in the editing of floating-point numbers (both on input and output). The format is:

```
decimal_point character
```

where *character* is the character chosen as the decimal delimiter.

**6.5.2 thousands\_sep Keyword**

This keyword specifies the character to be used as the thousands separator. The format is:

```
thousands_sep character
```

where *character* is the character chosen to separate groups of digits to the left of the decimal delimiter in formatted nonmonetary quantities. Note that none of the standard INTERACTIVE UNIX System subroutines or commands recognises a thousands separator.

**6.5.3 grouping Keyword**

The **grouping** keyword defines the size of each group of digits in formatted nonmonetary quantities. The format is:

```
grouping digit [ ; digit ] ...
```

where the operands are integers separated by semicolons. Each integer specifies the number of digits in a group, with the initial integer defining the size of the group immediately preceding the decimal delimiter and the following integers defining the preceding groups. Grouping is performed only for groups with a defined size unless the last integer is zero, in which case the size of the last group is used repeatedly for the remainder of the digits.

As an example of the interpretation of the **grouping** keyword, assume that the value to be formatted is 123456789 and the **thousands\_sep** is ". The following are the results with the various groupings shown:

grouping	Formatted Value
3	123456 798
3;0	123 456 789
3;2	1234 56 789
3;2;0	12 34 56 789

#### 6.5.4 An Example of a Numeric Category Definition

```
LC_NUMERIC
#
decimal_point      ", "
#
thousands_sep     ". "
#
grouping           3;0
#
END LC_NUMERIC
```

#### 6.5.5 How a Program Uses This Information

If a program needs to access the values in the current locale, it can do so via the library interfaces `localeconv` and `nl_langinfo`. Refer to `localeconv(3P)` and `nl_langinfo(3P)` for more information.

## 6.6 Monetary Editing

Monetary editing controls the appearance of monetary numbers. Note that no standard INTERACTIVE UNIX System library routines or commands take into account monetary editing. The following aspects of monetary editing are controlled via the `LC_MONETARY` locale category:

1. The character used as a monetary decimal delimiter.
2. The number of fractional digits.
3. The character used to separate groups of digits (thousands separator).
4. The size of such groups.
5. The content (and placement) of strings used to denote the currency.
6. Positive and negative signs and their placement.

## 6.7 Creating a Monetary Category Definition

The source language for the monetary category in the INTERACTIVE UNIX Operating System is the language defined by the POSIX.2 group for the `LC_MONETARY` locale category.

A monetary editing source definition consists of a header, a monetary editing body, and a trailer. The header is the word `LC_MONETARY`. The trailer is the string `END LC_MONETARY`.

The monetary editing body consists of one or more lines of text. Each line contains a keyword followed by one or more operands. Keywords are separated from the operands by one or more blank characters (space or tab).

Operands are characters, strings of characters, or digits. When a keyword is followed by more than one operand, the operands must be separated by semicolons. Blank characters are allowed before and/or after a semicolon. Strings must be surrounded by quotes. Individual characters may be surrounded by quotes, but it is not required. Blank lines or lines containing a number sign (#) in the first column are ignored.

The following keywords are recognised:

- `int_curr_symbol`  
Defines the ISO standard four-character (three letters and a space) code for currency, for example, "USD " for U.S. dollar.
- `currency_symbol`  
Defines the character to be used as the currency symbol, for example "\$".
- `mon_decimal_point`  
Defines the decimal delimiter for monetary quantities.
- `mon_thousands_sep`  
Defines the thousands separator for monetary quantities.
- `mon_grouping`  
Defines the grouping of digits.
- `positive_sign`  
Defines the positive sign.
- `negative_sign`  
Defines the negative sign.
- `int_frac_digits`  
Defines the number of fractional digits displayed when formatting using the `int_curr_symbol`.

`frac_digits`

Defines the number of fractional digits displayed when formatting using the `currency_symbol`.

`p_cs_precedes`

Defines whether the `currency_symbol` succeeds or precedes a positive quantity.

`p_sep_by_space`

Defines whether a space separates the `currency_symbol` from a positive quantity.

`n_cs_precedes`

Defines whether the `currency_symbol` succeeds or precedes a negative quantity.

`n_sep_by_space`

Defines whether a space separates the `currency_symbol` from a negative quantity.

`p_sign_posn`

Defines the placement of the sign and a positive quantity.

`n_sign_posn`

Defines the placement of the sign and a negative quantity.

### 6.7.1 `int_curr_symbol` Keyword

This keyword is used to define the international currency symbol. The operand must be a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in ISO 4217 (*Codes for the representation of currencies and funds*). The fourth character must be the character used to separate the international currency symbol from the monetary quantity, normally a space. For example:

```
int_curr_symbol "FMK "
```

### 6.7.2 `currency_symbol` Keyword

This keyword defines the string to be used as the local currency symbol. For example:

```
currency_symbol $
```

### 6.7.3 `mon_decimal_point` Keyword

The operand is the character to be used as the decimal delimiter to format monetary quantities. For example:

```
mon_decimal_point "$"
```

is the Portuguese monetary decimal delimiter.

### 6.7.4 `mon_thousands_sep` Keyword

This operand is the string to be used as the separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. For example:

```
mon_thousands_sep " "
```

### 6.7.5 `mon_grouping` Keyword

This keyword is used to define 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. Grouping is performed only for groups with a defined size, unless the last integer is zero, in which case the size of the last group is repeatedly used for the remainder of the digits. For example:

```
mon_grouping 3;0
```

### 6.7.6 `positive_sign/negative_sign` Keywords

The operand is a string used to indicate positive or negative values. For example:

```
positive_sign ""  
negative_sign "c"
```

### 6.7.7 `int_frac_digits` Keyword

This keyword is an integer that represents the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `int_curr_symbol`. For example:

```
int_frac_digits 2
```

### 6.7.8 `frac_digits` Keyword

This keyword is an integer that represents the number of fractional digits (those to the right of the decimal delimiter) to be displayed in

a formatted monetary quantity using `currency_symbol`. For example:

`frac_digits`            2

**6.7.9 `p_cs_precedes/n_cs_precedes` Keywords**

Each keyword is an integer that is set to 1 if the `currency_symbol` precedes the value for a positive or negative formatted monetary quantity, respectively, and set to 0 if the symbol succeeds the value. For example:

`p_cs_precedes`        1

**6.7.10 `p_sep_by_space/n_sep_by_space` Keywords**

Each keyword is an integer that is set to 1 if a space separates the `currency_symbol` from the value for a positive or negative formatted monetary quantity, respectively. They are set to 0 if no space separates the symbol from the value.

**6.7.11 `p_sign_posn/n_sign_posn` Keywords**

Each keyword is an integer that is set to a value indicating the positioning of the `positive_sign` or `negative_sign` for a positive or negative formatted monetary quantity, respectively. The following integer values are recognised:

- 0     Parentheses enclose the quantity and the `currency_symbol`.
- 1     The sign string precedes the quantity and the `currency_symbol`.
- 2     The sign string succeeds the quantity and the `currency_symbol`.
- 3     The sign string immediately precedes the `currency_symbol`.
- 4     The sign string immediately succeeds the `currency_symbol`.

### 6.7.12 An Example of a Monetary Category Definition

```

LC_MONETARY
#
int_curr_symbol      "CHF "
currency_symbol      "SFrs."
mon_decimal_point    "."
mon_thousands_sep   ","
mon_grouping         3;0
positive_sign        ""
negative_sign        "C"
int_frac_digits      2
frac_digits          2
p_cs_precedes        0
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

```

With the above definition, a monetary quantity should be edited as follows:

```

Positive      SFrs.1,234.56
Negative      SFrs.1,234.56C

```

### 6.7.13 How a Program Uses This Information

If a program needs to access the values in the current locale, it can do so via the library interfaces `localeconv` and `nl_langinfo`. Refer to *localeconv(3P)* and *nl\_langinfo(3P)* for more information.

## 7. SPECIFYING YES/NO RESPONSE INFORMATION

The “yes/no” response category determines the correct string to be used as affirmative (yes) and negative (no) responses to program queries.

### 7.1 Reasons for Defining Yes/No Responses

The standard UNIX System utilities that require this kind of interaction (such as `rm`) normally expect either a `y` or an `n`. In countries that do not normally use the English language, this is not the obvious response. In France, for instance, the obvious affirmative response would be `o` (for `oui`); in Spain, it would be `s` (for `si`).

### 7.2 Defining Yes/No Responses

These definitions are created by placing a specification in the `LC_MESSAGES` file in a `locale` directory.

### 7.3 When to Use the Yes/No Response `locale` Category

The created and installed definitions are not activated until the user specifies that they should be used. To do this, the user must set the `LC_ALL`, `LC_MESSAGES`, or `LANG` environment variable to the directory in which the files are stored. This must be done before a program using the stored definitions is executed. Note that the program must be set up to check and set the international environment (via the `setlocale` function). In the INTERACTIVE UNIX System, the standard utilities that depend on a yes/no response, such as `ln` and `rm`, have been modified to use the international environment. Note that while the internationalised yes/no response is required by XPG3 for certain commands, the `LC_MESSAGES` category is not part of the `locale` as defined by XPG3.

### 7.4 Creating a Yes/No Response Category Definition

The source language for the yes/no response category in the INTERACTIVE UNIX Operating System is the language defined by the POSIX.2 group for the `LC_MESSAGES` category.

A yes/no response source definition consists of a header, a response body, and a trailer. The header is the word `LC_MESSAGES`. The trailer is the string `END LC_MESSAGES`.

The response body consists of one or more lines of text. Each line contains a keyword, followed by one or more operands. Keywords are separated from the operands by one or more blank characters (space or tab).



Operands are characters, strings of characters, or digits. When a keyword is followed by more than one operand, the operands must be separated by semicolons (;). Blank characters are allowed before and/or after a semicolon. Strings must be surrounded by quotes. Individual characters may be surrounded by quotes, but it is not required. Blank lines or lines containing a number sign (#) in the first column are ignored.

The following keywords are recognised:

<code>LC_MESSAGES</code>	The header.
<code>yesexpr</code>	Defines the affirmative (yes) response.
<code>noexpr</code>	Defines the negative (no) response.
<code>END LC_MESSAGES</code>	The trailer.

#### 7.4.1 `yesexpr` Keyword

This keyword specifies the character or string to use as the affirmative (yes) response. The format is:

```
yesexpr regular-expression
```

where *regular-expression* is a regular expression which, when used to match affirmative responses, will report a match.

#### 7.4.2 `noexpr` Keyword

This keyword specifies the character or string to use as the negative (no) response. The format is:

```
noexpr regular-expression
```

where *regular-expression* is a regular expression which, when used to match negative responses, will report a match.

#### 7.4.3 An Example of a Response Category Definition

```
LC_MESSAGES
#
yesexpr          "[Yy].*"
#
noexpr           "[Nn]on"
#
END LC_MESSAGES
```

#### 7.4.4 How a Program Uses This Information

If a program needs to access the values in the current `locale`, it can do so via the `nl_langinfo` library interface. Refer to `nl_langinfo(3P)` for more information.

## 8. TIPS FOR PROGRAMMERS

This section is written for programmers who want to take advantage of the INTERACTIVE Software Development System capabilities that support features that deal with internationalisation, in particular those described in the *X/Open Portability Guide*. It is not designed as a programmer's guide, but simply points programmers to the appropriate references where these features are described. Manual entries that deal with the features appear in the "International Supplement Reference Manual" and in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*. To be able to use all the features described, programs should always be compiled and linked using the `-Xp` option and contain the following line in the source file before the inclusion of any header files:

```
#define _XOPEN_SOURCE
```

### 8.1 Character Mapping

We do not recommend trying to change the active character mapping from an application. However, some programs (the VP/ix\* Environment or `vpix`, for example, which uses MS-DOS\*-style (DOS) mapping) might want to disable the mapping and set it back before exiting. `ioctl` commands are available to do this. The following syntax is used:

```
ioctl(fd, COMMAND, buffer);
```

`fd` is the file descriptor for the tty port for which the `COMMAND` is intended. `buffer` is a pointer of type `unsigned char` pointing to a buffer of size 1K. The following `ioctl` commands can be used:

- **LDSMAP**

The buffer is checked for correctness. If some pointers have the wrong value or the size of the buffer exceeds 1K, the `ioctl` call fails and returns -1. Otherwise, the buffer is copied into kernel space and mapping is activated.

- **LDGMAP**

If no mapping buffer is present for the terminal port corresponding to `fd`, the `ioctl` returns -1. Otherwise, the content of the mapping buffer is copied from kernel space into `buffer` and the `ioctl` returns 0.

- **LDNMAP**

If no mapping buffer is present for the terminal port corresponding to `fd`, the `ioctl` returns `-1`. Otherwise, the content of the mapping buffer is freed and mapping is disabled.

- **LDDMAP**

If no mapping buffer is present for the terminal port corresponding to `fd`, the `ioctl` returns `-1`. Otherwise, mapping is temporarily disabled.

- **LDEMAP**

If no mapping buffer is present for the terminal port corresponding to `fd`, the `ioctl` returns `-1`. Otherwise, it is reenabled.

A description of all `ioctl` commands listed here and the structure of the mapping buffer can be found in the file `/usr/include/sys/emap.h`.

## 8.2 Giving Programs Access to locales

The `setlocale` function sets, changes, or queries the program's locale according to the values of the `category` and `locale` arguments. Therefore, every program that wants to take advantage of the internationalisation features described in this document and the "International Supplement User's Manual" should, at a minimum, contain the following statements:

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

and

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

The latter statement causes the program to find out the current locale value. If the second argument is not an empty string, it sets the locale instead. Refer to `setlocale(3P)` for more information.

## 8.3 Date and Time

In order to have access to the date and time information, a `setlocale` statement must be part of the program. If all other locale categories are not to be used,

```
setlocale("LC_TIME, "");
```

is sufficient. In addition, the `strftime` function should be used instead of the traditional `cftime`. Refer to `ctime(3P)` for more information.

When, in the flow of the program, the value of the local day or month is needed, the `nl_langinfo` function can be used. It returns a string with the value requested. Refer to `nl_langinfo(3P)` for more information.

## 8.4 Character Classification

At a minimum, use the following statement in your program:

```
setlocale("LC_CTYPE", "");
```

Make sure you also use the family of `toupper`, `isupper`, and similar functions. No further changes have to be made to the program. Refer to `ctype(3C)` for more information.

## 8.5 Collation

There are two functions for handling international sorting: `strcoll` and `strxfrm`. They are also part of the ANSI C standard. They differ from the traditional `strcmp` in that they use the sorting rules defined in a given locale rather than using the internal byte representation inside the computer. At a minimum, the following statement should be part of the program:

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

`strcoll` is very similar to `strcmp`, but is slower than the older function since it is table-driven. `strxfrm` is a different type of function in that it transforms the data it gets and returns a string of characters that can be given to `strcmp` to be sorted. It is useful when performance is an issue and the same set of data needs to be compared several times. Refer to `strcoll(3P)`, `strxfrm(3P)`, and `string(3P)` for more information.

## 8.6 Regular Expressions

Programs have access to internationalised regular expressions when they are compiled with the `-Xp` option and include the following statements in the program:

```
#define _XOPEN_SOURCE
#include <regex.h>
```

## 8.7 Numeric and Monetary Formatting

`printf` and other functions have been modified to use numeric formatting. It is accessed using the statement:

```
setlocale("LC_NUMERIC", "");
```

in the program. Although no functions currently use monetary formatting, applications can do so by using the statement:

```
setlocale("LC_MONETARY,");
```

in the program. Note that using `LC_ALL` is sufficient to do the job for all `locale` categories.

When the value of one of the numeric or monetary conventions is needed in the flow of the program, the `localeconv` function can be used. It returns a data structure containing all the relevant values. Refer to `localeconv(3P)` for more information.

## 8.8 Message Catalogues

Three functions should be used to write programs that use message catalogues rather than hardcoded text:

- `catopen`

This function takes two arguments, the second of which should always be zero. The first argument, *name*, of type `char*`, specifies the name of the message catalogue to be opened. If *name* contains a slash (/), it specifies a complete name for the message catalogue. Otherwise, the environment variable `NLSPATH` is used with *name* substituted for `%N` (refer to `environ(5P)` for the description of `NLSPATH`). If `NLSPATH` does not exist in the environment, or if a message catalogue cannot be opened in any of the components specified by `NLSPATH`, then the default used by this implementation is `/lib/locale/ISC/msgcat/name`. The function returns a message catalogue descriptor (type `n1_catd`, defined in the include file `n1_types.h`). Refer to the `hello.c` sample file later in this section and to `catopen(3P)` for more information.

- `catgets`

This key function takes four arguments. The first is the message catalogue descriptor returned by a previous `catopen`. The second is the set number or identifier (the default set identifier, `NL_SETD`, is defined in `n1_types.h`). The third is the message number or identifier. The fourth is the default message in case no message catalogue is found or the specified message is not in the message catalogue. Refer to `catgets(3P)` for more information.

- **catclose**

This should be used at the end of the program to close the previously opened message catalogue. It takes one argument, which is a message catalogue descriptor returned by a previous `catopen`. Refer to *catclose(3P)* for more information.

A message catalogue can then be created containing the text of the local language. This is a text file with a particular format (refer to *gencat(4P)* for details). The `gencat` utility (see *gencat(1P)*) should then be used to convert the message catalogue source into a real (binary) message catalogue.

INTERACTIVE has added a utility, `showcat`, that can be used to translate the contents of a message catalogue into its message text source (that is, the opposite of the `gencat` utility), unless an option to prevent this translation was used when `gencat` was used to create the message catalogue. Refer to *showcat(1P)* for more information.

The following example lists the source of the famous `hello.c` program when fully internationalised:

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>
main(argc,argv)
int argc;
char **argv;
{
    nl_catd catd;
    setlocale(LC_ALL,"");
    catd = catopen(argv[0],0);
    printf("%s\n",catgets(catd,NL_SETD,1,"hello, world"));
    catclose(catd);
}
```

The message catalogue source looks like this:

```
$set 1
1 hello, world
```

### 8.8.1 Extension of `printf` Syntax

The example shown handles a simple case of a message catalogue — a string without parameters to be filled in. However, many messages do have parameters. When text is translated, the words in the translated version often have to be in a different order than in the original because of grammatical differences. For example, in English, adjectives precede nouns (white lady, a cocktail), whereas in French, they usually follow nouns (dame blanche, a famous ice

cream dish). When program messages are translated and the program uses `printf`, X/Open extensions provided in the INTERACTIVE UNIX System can be used to indicate the order.

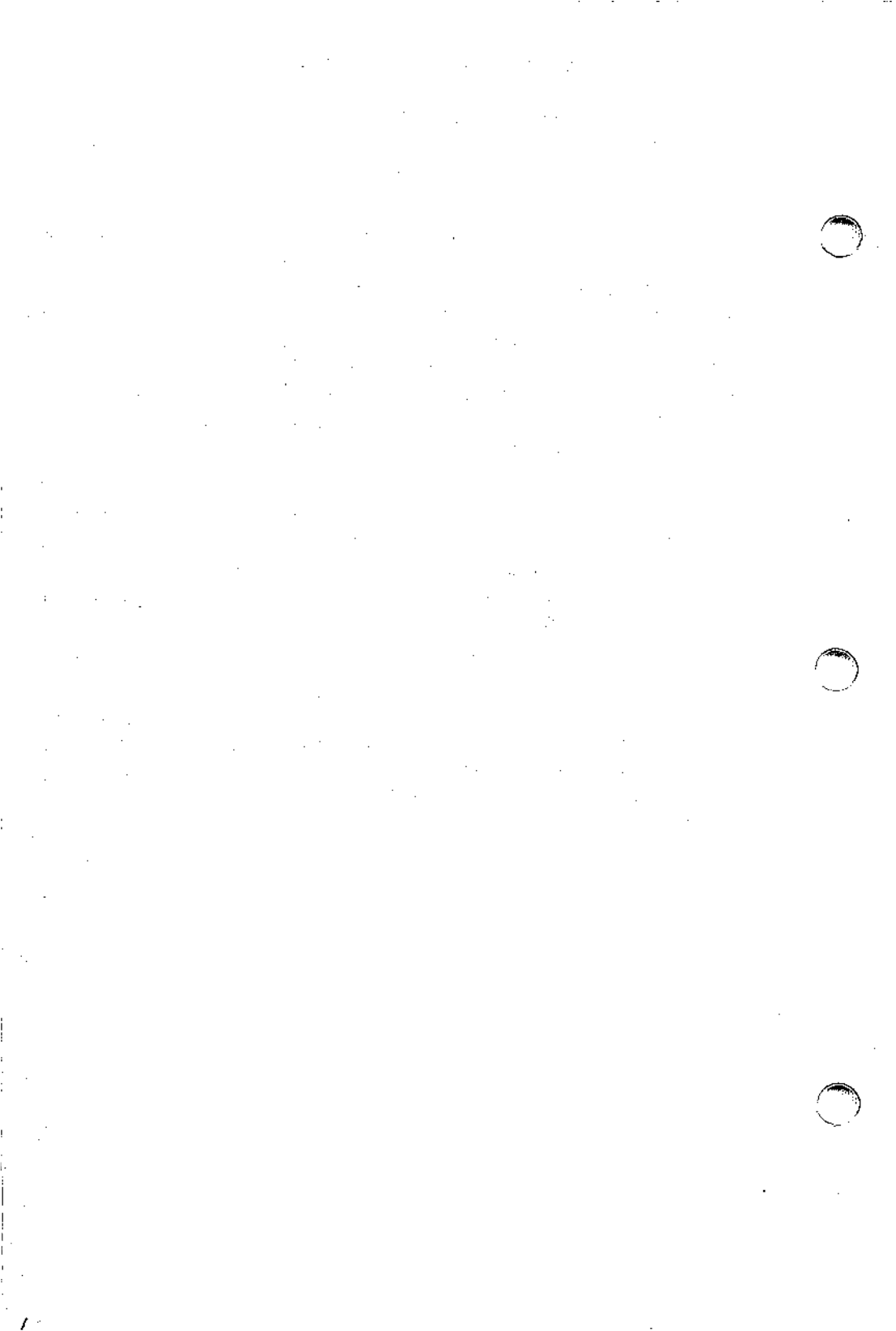
Normally, conversions in a format string are performed in the order they are specified in the format statement, that is, the first argument is applied to the first conversion specification, the second argument to the second format specification, and so on. However, the conversions can be applied to the  $n$ th argument in the argument list, rather than to the next unused one, if the conversion character `%` is replaced by the sequence `%digit$`, where *digit* is a decimal integer  $n$  in the range between 1 and `{NL_ARGMAX}` (defined in the include file `limits.h`), giving the position of the argument in the argument list. For example:

```
printf("%1$s %2$s\n", adjective, noun);
```

In format strings containing the `%digit$` form of a conversion specification, a field width or precision may be indicated by the sequence `*digit$`, where *digit* is a decimal integer  $n$  in the range between 1 and `{NL_ARGMAX}`, giving the position of the argument containing the field width or precision. For example:

```
printf("%1$d:%2$. *3$d:%4$. *3$d\n", hour, min, precision, sec);
```

The format string can contain either numbered argument specifications (`%digit$` and `*digit$`) or unnumbered argument specifications, but not both. When numbered argument specifications are used, specifying the  $n$ th argument requires that all the leading arguments, from the first to the  $(n-1)$ th, be specified in the format string.





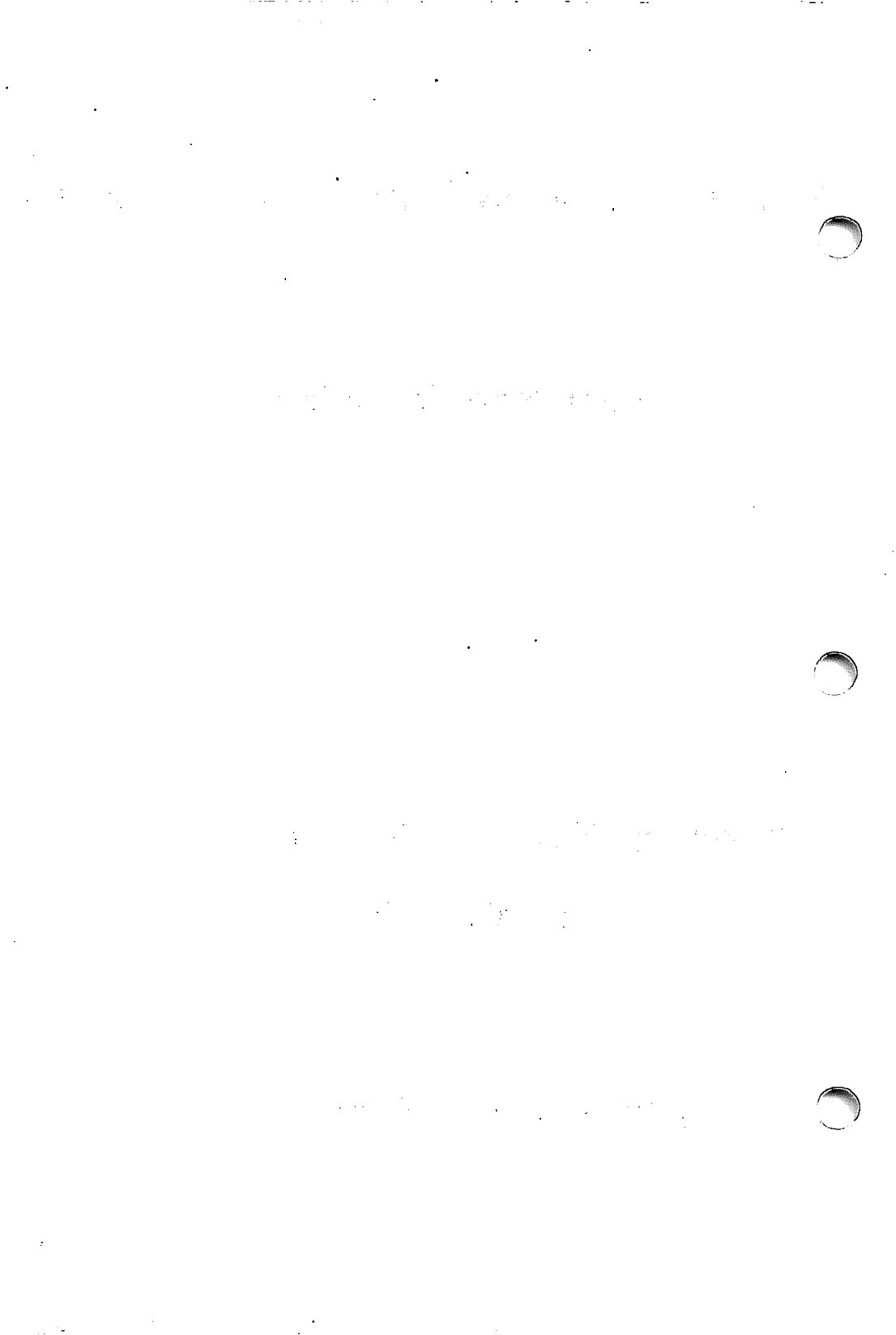
# **X/Open Conformance Statement - Questionnaire**

## **X/Open Portability Guide 3**

**Completed by INTERACTIVE Systems Corporation**

**September, 1991**

**Document Revision Number 3.2**



## **Contents**

### **Chapter 2: Internationalised System Calls and Libraries**

Section 2.1: General Attributes

Section 2.2: Process Handling

Section 2.3: File Handling

Section 2.4: General Terminal Interface

Section 2.5: Internationalised System Interfaces

### **Chapter 3: Commands and Utilities**

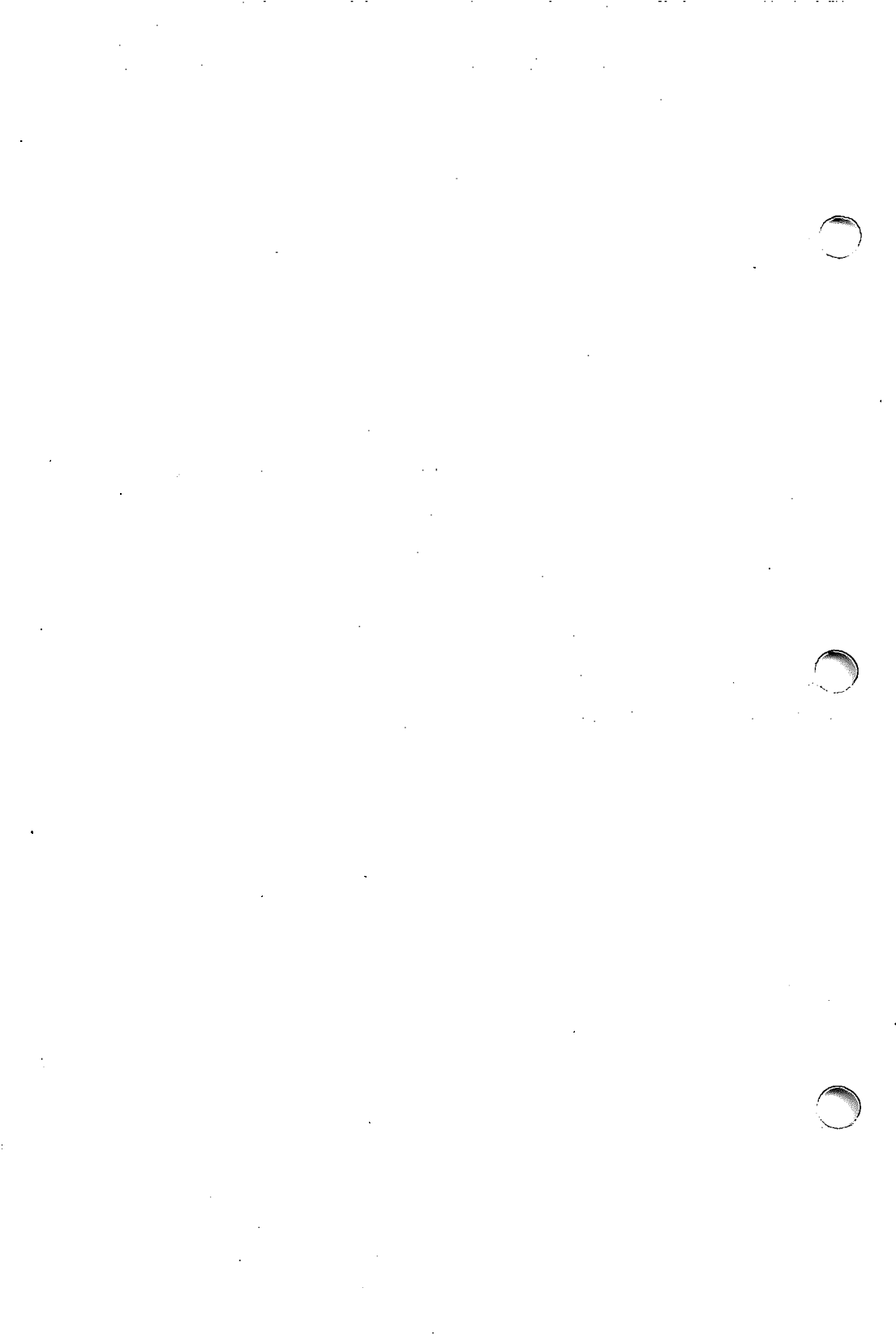
Section 3.1: Basic Utilities

Section 3.2: Development Utilities

Section 3.3: Internationalisation Option

### **Chapter 4: C Language**

### **Chapter 15: Source Code Transfer**



## Chapter 2: Internationalised System Calls and Libraries

### Product Identification

Product Identification INTERACTIVE UNIX System V/386 Release 3.2  
Version/Release No. 3.0

If you do not supply this component yourself, please identify below the supplier you reference.

### Conformance Reference

#### Indicator of Compliance

VSX Test Suite Release 3.204  
Testing Agency Name UniSoft Corporation  
Address 6121 Hollis Street  
Emeryville, CA 94608-2092

### Environment Specification

Enter below details of the hardware and software environment in which testing took place, including compilation routines and installation procedures (if any). Sufficient detail must be supplied to enable conformant behaviour and any test results to be reproduced.

Any 386/486-compatible system with at least 4 MB of RAM and with the following INTERACTIVE UNIX System V/386 Release

**3.2, Version 3.0 subsets and extensions installed (approximately  
40 MB of disk space is needed):**

**Core  
Kernel Configuration  
File Management  
International Supplement  
INTERACTIVE Software Development System**

## **Temporary Waivers**

**List below references to any temporary waivers granted by  
X/Open in respect of minor errors in the product referenced  
above. This should include the X/Open reference and the waiver  
expiry date. The waivers as granted shall be made available with  
this document on request.**

**PG3.239    expiration date April 2, 1992**

## Section 2.1: General Attributes

### 2.1.1 POSIX.1 Supported Features

**Question 1:** *Which of the following options, specified in the <unistd.h> header file, are available on the system?*

**Answer:**

Macro Name	Meaning	Provided
<code>__POSIX_CHOWN_RESTRICTED</code>	The use of <code>chown()</code> is restricted	Yes
<code>__POSIX_JOB_CONTROL</code>	Job Control option	Yes
<code>__POSIX_NO_TRUNC</code>	Long path name components generate an error	Yes
<code>__POSIX_SAVED_IDS</code>	Effective user and group IDs are saved	Yes
<code>__POSIX_VDISABLE</code>	Terminal special characters can be disabled	Yes

Options:

When the option is variable a description is required for the cases over which the variations occur.

**Rationale**

For an X/Open conforming implementation, the `_POSIX_SAVED_IDS` option must be provided. The other options may or may not be provided. The provision of the file system related options can vary within a system. For example, a system which has traditionally supported both System V and BSD type file systems may provide a mechanism whereby the option is enforced for certain files or processes but not for others. This technique can be used to achieve a degree of backwards compatibility that would not otherwise be possible.

**Reference**

XPG3 Volume 2 Page 579 – `<unistd.h>`

## 2.1.2 C Standard

**Question 2:** *Does the implementation only support Common Usage C or also support ANSI C Standard interface definitions?*

**Answer:**

Only Common Usage C.

1. Only Common Usage C
2. Both Common Usage C and ANSI C

**Rationale**

The POSIX.1 standard allows for a conforming system to support either Common Usage C or ANSI C Standard interface definitions. The XPG is based on a Common Usage C definition but does not prohibit an ANSI C implementation. A Common Usage C definition must provide function declarations for the C language functions in the XPG as well as providing function semantics that conform to the XPG. An ANSI C Standard interface must provide function prototypes and ANSI C semantics as well as providing XPG semantics. There are no known areas of contradiction between the ANSI C and XPG semantics.



Reference

XPG3 Volume 2 Page 12 - The Compilation Environment

### 2.1.3 Limit Values

**Question 3:** *What are the values associated with the following limits specified in the <limits.h> header file?*

**Answer:**

<u>Macro Name</u>	<u>Meaning</u>	<u>Minimum</u>	<u>Maximum</u>
ARG_MAX	Max length of argument list and environment data	5120	5120
CHILD_MAX	Max number of processes per user ID	15	60
LINK_MAX	Max number of links to a single file	1000	1000
MAX_CANON	Max bytes in a terminal canonical input line	255	255
MAX_INPUT	Max bytes in a terminal input queue	255	255

# X/Open Conformance Statement Questionnaire

XCS-QUE-3.2

NAME_MAX	Max characters in a file name	14	14
OPEN_MAX	Max number of files open in a process	20	100
PASS_MAX	Max significant characters in a password	8	8
PATH_MAX	Max characters in a path name	255	255
PIPE_BUF	Max bytes in an atomic write to a pipe	10240	10240
NGROUPS_MAX	Max number of supplementary group IDs	16	16
TMP_MAX	Max number of unique temporary file names	17576	17576

## Options:

Specify a minimum and maximum limit for each limit value. The minimum limit should be the result of evaluating the associated macro in `<limits.h>`. The maximum limit should be the largest value that is returned from `sysconf()` or `pathconf()`. The maximum values can be specified as indeterminate.

**Rationale**

Each of these limits can vary within bounds set by the X/Open Portability Guide. The minimum value that a limit can take on any X/Open conforming system is given in the corresponding `_POSIX_` value. A specific conforming implementation may provide a higher minimum value than this and the maximum value that it provides can differ from the minimum. Some conforming implementations may provide a potentially infinite value as the maximum, in which case the value is considered to be indeterminate. The minimum value must always be definitive since the `_POSIX_` value provides a known lower bound for the range of possible values.

**Reference**

XPG3 Volume 2 Page 538 – `<limits.h>`

**Question 4:** *What are the values associated with the following constants specified in the `<limits.h>` header file?*

**Answer:**

Macro Name	Meaning	Value
<code>CHAR_BIT</code>	Number of bits in a char	8
<code>LONG_BIT</code>	Number of bits in a long	32
<code>WORD_BIT</code>	Number of bits in a word	32
<code>DBL_DIG</code>	Digits of precision of a double	15
<code>DBL_MAX</code>	Maximum decimal value of a double	1.7976931348623157e+308

FLT_DIG	Digits of precision of a float	6
FLT_MAX	Maximum decimal value of a float	3.4028234663852885e+38

**Rationale**

This set of constants provides useful information regarding the underlying architecture of the implementation.

**Reference**

XPG3 Volume 2 Page 537 – <limits.h>

## 2.1.4 Error Conditions

**Question 5:** *Which of the following optional errors listed in the XPG are detected in the circumstances specified?*

**Answer:**

Function	Error	Detected
access()	EINVAL† ETXTBSY	No Yes
atof()	ERANGE	Yes
atoi()	ERANGE	No
atol()	ERANGE	No
cfsetispeed()	EINVAL	No
cfsetospeed()	EINVAL	No
chmod()	EINVAL	No
chown()	EINVAL†	No

Function	Error	Detected
closedir()	EBADF†	Yes
exec	ENOMEM†	Yes
	ETXTBSY	Yes
fcntl()	EDEADLK†	Yes
fdopen()	EBADF	No
	EINVAL	No
feof()	EBADF	No
ferror()	EBADF	No
fileno()	EBADF	No
fopen()	EINVAL	No
	ETXTBSY	Yes
freopen()	EINVAL	No
	ETXTBSY	Yes
fork()	ENOMEM	Yes
fseek()	EINVAL	Yes
ftw()	EINVAL	No
getcwd()	EACCES†	Yes
isatty()	EBADF	No
	ENOTTY	No
open()	EINVAL	Yes
	ETXTBSY	Yes
opendir()	EMFILE†	Yes
	ENFILE†	Yes
pathconf()	EACCES†	No
	EINVAL†	No
	ENAMETOOLONG†	No

Function	Error	Detected
	ENOENT†	No
	ENOTDIR†	No
fpathconf()	EBADF†	No
	EINVAL†	Yes
printf()	EINVAL	Yes
readdir()	EBADF†	Yes
rename()	ETXTBSY	No
scanf()	EINVAL	Yes
setvbuf()	EBADF	No
sigaddset()	EINVAL†	Yes
sigdelset()	EINVAL†	Yes
sigismember()	EINVAL†	Yes
strcoll()	EINVAL	No
strerror()	EINVAL	Yes
strtol()	EINVAL	Yes
	ERANGE	Yes
strxfrm()	EINVAL	Yes
unlink()	ETXTBSY	Yes

### Rationale

Each of the above error conditions is marked as optional in the XPG and an implementation may return this error in the circumstances specified or may not provide the error indication. Those items marked with a † are also considered to be optional error conditions in POSIX.1. The EINVAL error condition for the three functions **sigaddset()**, **sigdelset()**, and **sigismember()** are mandated in the XPG but are considered optional in POSIX.1. An X/Open-conforming implementation will always produce these errors, but a POSIX.1-conforming implementation may not.

Reference

XP33 Volume 2 Page 32 – Error Numbers

## 2.1.5 Mathematical Interfaces

**Question 6:** *What format of floating point numbers are supported by this implementation?*

**Answer:**

IEEE floating point format.

**Options:**

1. IEEE floating point format.
2. Description of floating point format supported.

**Rationale**

Most implementations support IEEE floating point format either in hardware or software. Some implementations support other formats with different exponent and mantissa accuracy. These differences need to be defined.

**Question 7:** *Is long double form supported and what precision is associated with this form?*

**Answer:**

Not supported. Long double equates to double.

**Options:**

1. Not supported. Long double equates to double.
2. Description of exponent and mantissa precision and number of bits associated with the long double format.

**Rationale**

The long double format can both vary in length and precision. If it is supported, other than as a synonym for double, the format needs to be described.

**Reference**

XPG3 Volume 2 Page 328 – printf()  
XPG3 Volume 2 Page 362 – scanf()

## 2.1.6 Data Encryption

**Question 8:** *Are the optional data encryption interfaces provided?*

**Answer:**

crypt()	No
encrypt()	No
setkey()	No

**Rationale**

Normally an implementation will either provide all three of these routines or will provide none of them at all. If the routines are not provided, then the implementation must provide a dummy interface which always raises an ENOSYS error condition.

**Reference**

XPG3 Volume 2 Page 3 – Status of Interfaces



## Section 2.2: Process Handling

### 2.2.1 Process Generation

**Question 9:** *Which file types (regular, directory, FIFO special etc.) are considered to be executable?*

**Answer:**

Regular.

**Options:**

A list of the types of file that are considered to be executable.

**Rationale**

The EACCES error associated with `exec` functions occurs in circumstances when the implementation does not support execution of files of the type specified. A list of these file types needs to be provided.

**Example**

Only regular file types may be executed.

**Reference**

XPG3 Volume 2 Page 129 – `exec`

## 2.2.2 Process Termination

**Question 10:** *Is the SIGCHLD signal sent to the parent process when a child exits?*

**Answer:**

Yes

**Rationale**

Some systems support the sending of SIGCHLD in these circumstances. This is mandatory if job control is supported.

**Reference**

XPG3 Volume 2 Page 132 – exit()

## 2.2.3 Process Environment

**Question 11:** *Is the setpgid() interface provided?*

**Answer:**

Yes

**Rationale**

This interface is mandatory on systems which support job control and may be provided on other systems.

**Reference**

XPG3 Volume 2 Page 3 – Status of Interfaces

## Section 2.3: File Handling

### 2.3.1 Access Control

**Question 12:** *What file access control mechanisms does the implementation provide?*

**Answer:**

Standard access control is provided.

**Options:**

1. Standard access control is provided.
2. Refer to: POSIX.1 Conformance Document Section 2.4.
3. Provide a definition of the additional or alternate access mechanisms.

**Rationale**

The XPG (and POSIX) allow an implementation to provide either additional or alternate file access control mechanisms other than the standard access control mechanism. The document should either describe or provide a reference to the details of alternate or additional access mechanisms. In particular, the method by which an application can execute using standard file access control should be explained and details of the changes required to utilise the alternate or additional access mechanisms should be given.

**Reference**

XPG3 Volume 2 page 16 – File Access Permissions

## 2.3.2 Files and Directories

**Question 13:** *Are any extended security controls implemented that could cause `fstat()` or `stat()` to fail?*

**Answer:**

No

**Rationale**

The XPG notes that there could be an interaction between extended security controls and the success of `fstat()` and `stat()`. This would suggest that an implementation can allow access to a file but not allow the process to gain information about the status of the file.

**Reference**

XPG3 Volume 2 Page 478 – `tempnam()`

## 2.3.3 Formatting Interfaces

**Question 14:** *Is the `L` modifier to `printf()` and `scanf()` supported on this implementation?*

**Answer:**

No

**Rationale**

The XPG notes that the `L` modifier which is exactly equivalent to the `l` modifier when the implementation does not differentiate between double and long double, is not supported on all systems and is only included for compatibility with ANSI C.

**Reference**

XPG3 Volume 2 Page 328 – `printf()`

XPG3 Volume 2 Page 362 – `scanf()`

**Question 15:** *Does the printf() function produce character string representations for Infinity and NaN to represent the respective special double precision values?*

**Answer:**

Yes

**Rationale**

This behaviour is often provided on systems with mathematical functions that produce these results.

**Reference**

XPG3 Volume 2 Page 331 – printf()

## Section 2.4: General Terminal Interface

### 2.4.1 Interfaces Supported

**Question 16:** *Are the following terminal control interfaces provided?*

tcgetpgrp() tcsetpgrp()

**Answer:**

Yes

**Rationale**

These interfaces are mandatory for implementations that support **job control**. Implementations that do not support **job control**, may either always return the error indication [ENOSYS] or may provide the interface with the behaviour specified for an implementation that supports **job control**. This later case is useful for implementations which support only part of the **job control** specifications.

**Reference**

XPG3 Volume 2 Page 471 – tcgetpgrp

XPG3 Volume 2 Page 475 – tcsetpgrp

## Section 2.5: Internationalised System Interfaces

### 2.5.1 Codesets

**Question 17:** *Does the implementation support the ISO 8859-1:1987 codeset for data transmission?*

**Answer:**

Yes

**Rationale**

The XPG defines the ISO 8859-1:1987 as the major Western European transmission codeset and also recommends its use as the corresponding internal codeset.

**Reference**

XPG3 Volume 3 Page 19 – Character Codesets and Text Transfer

**Question 18:** *Does the implementation use the ISO 8859-1:1987 as its internal codeset?*

**Answer:**

The implementation does not prescribe a specific internal codeset. Any single-byte codeset that is a true superset of ISO 646 (IRV), including ISO 8859-1:1987, can be used as the internal codeset.

**Rationale**

The XPG defines the ISO 8859-1:1987 as the major Western European transmission codeset and also recommends its use as the corresponding internal codeset.

**Reference**

XPG3 Volume 3 Page 19 – Character Codesets and Text Transfer

## 2.5.2 Regular Expression Interfaces

**Question 19:** *What form of regular expression syntax is supported by the `regexp()` interface?*

**Answer:**

Simple Internationalised (assuming this is in regard to the `regexp.h` interface)

**Rationale**

The `regexp()` interface may support either the simple regular expression or the simple internationalised regular expression syntax as defined in the XPG3 Volume 3 – Supplementary Definitions.

**Reference**

XPG3 Volume 3 Pages 49-51 – Regular Expressions



## Chapter 3: Commands and Utilities

### Product Identification

Product Identification    INTERACTIVE UNIX System V/386 Release 3.2  
Version/Release No.    3.0

If you do not supply this component yourself, please identify below the supplier you reference.

### Conformance Reference

Indicator of Compliance  
None

### Environment Specification

Enter below details of the hardware and software environment in which conformance is claimed, including compilation routines and installation procedures (if any). Sufficient detail must be supplied to enable conformant behaviour to be reproduced.

Any 386/486-compatible system with at least 4 MB of RAM and the following INTERACTIVE UNIX System V/386 Release 3.2, Version 3.0 subsets and extensions installed (approximately 40 MB of disk space is needed):

**Core  
Kernel Configuration  
File Management  
International Supplement  
INTERACTIVE Software Development System**

## **Conformance Expectations**

Volume 1 of XPG3 recognises that convergence of implementations towards a common specification for commands and utilities is not yet complete and therefore does not require a vendor to supply all of the commands and utilities (and individual options) specified in XPG3.

This chapter explicitly identifies those commands and utilities not supplied by the vendor and any supplied which do not conform to the published specification. (Reference : XPG3 Volume 1 Page 1).

## Section 3.1: Basic Utilities

### 3.1.1 Supported Commands

**Question 1:** *Which of the basic utilities (non-development utilities) defined in the XPG are not provided with the implementation?*

**Answer:**

All are provided.

**Options:**

A list of utilities that are not provided.

**Rationale**

The XPG Volume 1 states that “this volume in its current form is useful only as a guide to portability, but it is not possible to precisely define or test conformance to it.” This question determines whether or not the implementation provides a command of the name specified in the XPG, it does not attempt to determine whether it supports the semantics of that command. The (optional) development utilities are excluded from this question and are dealt with in the next section of the questionnaire.

**Example**

The **mailx** and **newgrp** commands are not provided.

**Reference**

XPG3 Volume 1 Page 1 – Introduction

### 3.1.2 Command Behaviour

**Question 2:** *In what ways do the commands provided by the implementation behave differently from the specifications contained in the XPG?*

**Answer:**

The commands behave in the manner specified for each of the command options detailed in the XPG.

**Options:**

1. The commands behave in the manner specified for each of the command options detailed in the XPG.
2. A list of deviances for each of the commands is provided. This list should be in a tabular form giving the name of the command, the command option and a description of the deviant behaviour.

**Rationale**

This question provides a greater degree of granularity than the previous question, requiring the semantic differences associated with the commands to be specified. Again, the question relates to the basic utilities rather than the development utilities. The question only relates to the semantics of the options specified within the XPG, implementation specific extensions should not be documented.

## Section 3.2: Development Utilities

### 3.2.1 Supported Commands

**Question 3:** *Which of the development utilities defined in the XPG are not provided with the implementation?*

**Answer:**

All are provided.

**Options:**

1. All are provided
2. None are provided
3. A list of utilities that are not provided

**Rationale**

The XPG Volume 1 states that “The development utilities might not be present in all X/Open compliant systems; in designated (DEVELOPMENT) systems all of the development utilities must be present and must conform to the published definition.”

**Reference**

XPG3 Volume 1 Page 2 – Status of Interfaces

### 3.2.2 Command Behaviour

**Question 4:** *In what ways do the development utilities provided by the implementation behave differently from the specifications contained in the XPG?*

**Answer:**

<i>Command</i>	<i>Option</i>	<i>Description</i>
cc	-Xp	compiles and links for the POSIX and XPG3 environments
mailx		does not support internationalised behavior

**Options:**

1. The development utilities behave in the manner specified for each of the options detailed in the XPG.
2. A list of deviances for each of the utilities is provided. This list should be in a tabular form giving the name of the utilities, the option and a description of the deviant behaviour.

**Rationale**

This question provides a greater degree of granularity than the previous question, requiring the semantic differences associated with the development utilities to be specified.

## Section 3.3: Internationalisation Option

### 3.3.1 Commands and Utilities

**Question 5:** *Is an internationalised environment, reflecting changes in the locale setting as described in XPG Volume 1 - XSI Commands and Utilities, supported?*

**Answer:**

Except for **mailx**, the commands listed below support Internationalisation in the manner specified in XPG3.

**Options:**

1. The commands listed below support Internationalisation in the manner specified in XPG3.
2. A list of deviations in the Internationalised behaviour of the following commands, compared to that specified in XPG3, is provided.

Command	Behaviour Specified in XPG3	Supported
ar	LC_TIME affects date format	Yes
awk	LC_COLLATE, LC_CTYPE affect regular expression matching	Yes
	LC_COLLATE affects the behaviour of string comparisons	Yes
	LC_NUMERIC affects the behaviour of the radix character	Yes
	As per POSIX.1, awk only recognizes the period (.) as the radix character in scripts	

Command	Behaviour Specified in XPG3	Supported
comm	LC_COLLATE affects sorting sequence	Yes
cp,ln,mv	LANG affects yes string	Yes
cpio	LC_COLLATE, LC_CTYPE affect filename pattern matching	Yes
	LC_TIME affects date format	Yes
date	LC_TIME affects date formatting options	Yes
ed,red	LC_COLLATE, LC_CTYPE affect regular expression matching	Yes
	LC_CTYPE is used to determine whether characters are printable	Yes
egrep	LC_COLLATE, LC_CTYPE affect regular expression matching	Yes
	LC_CTYPE is used to determine character classification (alphabetic, upper-case, lower-case)	Yes
expr	LC_COLLATE, LC_CTYPE affect regular expression matching	Yes
	LC_COLLATE affects the behaviour of relational operators	Yes
fgrep	LC_CTYPE is used to determine character classification (alphabetic, upper-case, lower-case)	Yes
find	LANG affects yes string	Yes
	LC_COLLATE, LC_CTYPE affect filename pattern matching	Yes
grep	LC_COLLATE, LC_CTYPE affect regular expression matching	Yes



Command	Behaviour Specified in XPG3	Supported
	LC_CTYPE is used to determine character classification (alphabetic, upper-case, lower case)	Yes
join	LC_COLLATE affects sorting sequence	Yes
lpstat	LC_TIME affects date format	Yes
ls	LC_COLLATE affects sorting sequence	Yes
	LC_CTYPE is used to determine whether a character is printable	Yes
	LC_TIME affects date format	Yes
mail	LC_TIME affects date format	Yes
mailx	LC_COLLATE, LC_CTYPE affect file name pattern matching	No
	LC_TIME affects date format	No
pg	LC_COLLATE, LC_CTYPE affect filename pattern matching	Yes
pr	LC_TIME affects date format	Yes
	LC_CTYPE is used to determine whether a character is printable	Yes
ps	LC_TIME affects date format	Yes
rm,rmdir	LANG affects yes string	Yes
sed	LC_COLLATE, LC_CTYPE affect regular expression matching	Yes
	LC_CTYPE is used to determine whether a character is printable	Yes
sh	LC_COLLATE, LC_CTYPE affect filename pattern matching	Yes
	LC_CTYPE is used to determine whether a character is alphabetic	Yes

Command	Behaviour Specified in XPG3	Supported
sort	LC_COLLATE affects sorting sequence	Yes
	LC_CTYPE affects character classification (alphabetic, uppercase, printing)	Yes
	LC_NUMERIC affects the determination of the radix character	Yes
tar	LC_TIME affects date format	Yes
	LANG affects yes string	Yes
tr	LC_COLLATE, LC_CTYPE affect bracketed expressions	Yes
	LC_CTYPE affects the definition of the character universe	Yes
uniq	LC_COLLATE affects sorting sequence	Yes
uucp	LC_TIME affects date format	Yes
uustat	LC_TIME affects date format	Yes
wc	LC_CTYPE is used to determine white-space characters	Yes
who	LC_TIME affects date format	Yes
yacc	LC_CTYPE is used to determine character classification	Yes

### Rationale

This behaviour is collectively optional, that is, it should be provided for all commands listed (subject to sections 3.1 and 3.2 which identify those commands not supplied by the vendor and those which do not fully support the X/Open specification).

### Reference

XPG3 Volume 1 Pages 4-5 – Status of Interfaces

### 3.3.2 Regular Expressions in Commands

**Question 6:** *Which form of regular expression syntax is supported by those commands which use regular expressions?*

**Answer:**

<u>Command</u>	<u>Regular Expression Syntax Supported</u>
awk	<b>Extended Internationalised</b>
csplit	Simple Internationalised
ed	<b>Simple Internationalised</b>
egrep	<b>Extended Internationalised</b>
ex	Simple
expr	<b>Simple Internationalised</b>
grep	<b>Simple Internationalised</b>
lex	Extended
pg	<b>Simple Internationalised</b>
sdb	Simple
sed	<b>Simple Internationalised</b>
vi	Simple

Note: An XPG 3 conforming system which claims support for internationalised commands should provide the regular expression syntax marked in **bold** in the above table. Where neither options are marked in **bold**, either may be provided.

#### Rationale

The XPG Volume 3 – XSI Supplementary Definitions requires that an internationalised set of commands will provide regular expression syntax for the above commands in one of the forms specified for that command. The XPG encourages the implementation of

internationalised regular expressions for all of the above utilities. It should be noted that the **sdb** command is an optional development utility and may not be available on all XPG conforming systems.

**Reference**

**XPG3 Volume 3 Pages 49-51 – Regular Expressions**

## Chapter 4: C Language

### Product Identification

Product Identification    INTERACTIVE UNIX System V/386 Release 3.2  
Version/Release No.    3.0

If you do not supply this component yourself, please identify below the supplier you reference.

### Conformance Reference

#### Indicator of Compliance

VSX Test Suite Release    3.204  
Testing Agency Name    UniSoft Corporation  
Address    6121 Hollis Street  
Emeryville, CA 94608-2092

### Environment Specification

Enter below details of the hardware and software environment in which testing took place, including compilation routines and installation procedures (if any). Sufficient detail must be supplied to enable conformant behaviour and any test results to be reproduced.

Any 386/486-compatible system with at least 4 MB of RAM and the following INTERACTIVE UNIX System V/386 Release 3.2,

**Version 3.0 subsets and extensions installed (approximately 40 MB of disk space is needed):**

**Core  
Kernel Configuration  
File Management  
International Supplement  
INTERACTIVE Software Development System**

## **Temporary Waivers**

List below references to any temporary waivers granted by X/Open in respect of minor errors in the product referenced above. This should include the X/Open reference and the waiver expiry date. The waivers as granted shall be made available with this document on request.

## Section 4.1: Implementation Limits

**Question 1:** *What limits does the implementation impose on the significant part of a identifier?*

**Answer:**

External identifiers            an infinite number of characters

Non-External identifiers      an infinite number of characters

**Rationale**

The XPG states that, while there is no limit to the length of an identifier, only a certain number of characters are significant. The XPG points out that there must be at least eight characters for a non-external name, but may be less for external names.

**Reference**

XPG3 Volume 4 Page 3 – Lexical Conventions

## Section 4.2: General

**Question 2:** *What truncation rules are applied when a floating value is converted to an integral value?*

**Answer:**

Truncation toward zero.

**Options:**

A description of the manner in which floating values are converted. The description should address the rules for truncation of both positive and negative values.

**Rationale**

The XPG states that such conversions are machine dependent. In particular, the XPG points out the differences related to the truncation of negative numbers.

**Reference**

XPG Volume 4 Page 10 – Conversions

**Question 3:** *What truncation rules are applied when using the division operator and either of the operands is negative?*

**Answer:**

Truncation toward zero.

**Rationale**

The XPG states that such truncations are machine dependent.

**Reference**

XPG Volume 4 Page 16 – Expressions



## Chapter 15: Source Code Transfer

### Section 15.1: Utilities

#### Product Identification

Product Identification    INTERACTIVE UNIX System V/386 Release 3.2  
Version/Release No.    3.0

If you do not supply this component yourself, please identify below the supplier you reference.

#### 15.1.1 Conformance Reference

Indicator of Compliance  
None.

#### Environment Specification

Enter below details of the hardware and software environment in which conformance is claimed, including compilation routines and installation procedures (if any). Sufficient detail must be supplied to enable conformant behaviour to be reproduced.

Any 386/486-compatible system with at least 4 MB of RAM and the following INTERACTIVE UNIX System V/386 Release 3.2, Version 3.0 subsets and extensions installed (approximately 40 MB of disk space is needed):

**Core  
Kernel Configuration  
File Management  
International Supplement  
INTERACTIVE Software Development System**

1600 bpi PE magnetic tape is supported with the INTERACTIVE UNIX Operating System when using a controller card and a tape unit for which a device driver is available. Several vendors provide such hardware/software.

## **Temporary Waivers**

List below references to any temporary waivers granted by X/Open in respect of minor errors in the product referenced above. This should include the X/Open reference and the waiver expiry date. The waivers as granted shall be made available with this document on request.

## Formats

**Question 1:** *Which exchange media format(s) may be written* by the system?

**Answer:**

80 track diskettes	Yes
40 track diskettes	Yes
1600bpi PE magnetic tape	Yes

**Rationale**

XPG3 states that standards are referenced for transfer of diskettes and magnetic tapes between machines. Because of the different nature of X/Open conformant systems, it is not possible to define a single portable medium that is supported across the whole range of systems.

**Reference**

XPG3 Volume 3 Chapters 15, 16, and 17

**Question 2:** *Which exchange media format(s) may be read* by the system?

**Answer:**

80 track floppy disk	Yes
40 track floppy disk	Yes
1600bpi PE magnetic tape	Yes

**Rationale**

XPG 3 states that standards are referenced for transfer of diskettes and magnetic tapes between machines. Because of the different nature of X/Open conformant systems, it is not possible to define a single portable medium which is supported across the whole range of systems. In addition, some systems can read a wider range of formats that they can write.

Reference

XPG3 Volume 3 Chapters 15, 16, and 17

## Utilities

**Question 3:** *Which utilities are used to create and read the archive formats specified in XPG Volume 3 – XSI Supplementary Definitions?*

**Answer:**

Format	Creating	Reading
Extended tar	tar	tar
cpio	cpio	cpio

Options:

A definition of the commands used to create and read these formats. If a special option is required to produce the specified format this must be detailed.

Refer to: POSIX.1 Conformance Document Section 10.1

Rationale

There is no explicit definition as to the commands that must be used to create and retrieve these archives. On most systems this will be achieved by the **tar** and **cpio** commands. There are other commands available that produce these archives. On some implementations the command may need a special option to enable reading of the specified formats with the “standard” option being to create archives which are backwards compatible with previous versions of the command.

Reference

XPG3 Volume 3 Page 151-2 – Utilities

## Invalid File Names

**Question 4:** *What file name is used to contain data from the archive in the case that the file name on the archive is invalid for the system on which the file hierarchy is being created?*

**Answer:**

Format	File
Extended tar	The archive reading utility relies on standard file and directory creating system interfaces to create files and directories. On extraction from the archive, the only case where a filename would be changed is if a pathname component exceeds the system filename length limit of NAME_MAX (14 characters), in which case it would be truncated to NAME_MAX characters.
cpio	The archive reading utility handles invalid file and directory names in the same manner as extended tar.

**Options:**

1. Definition of the file name used.
2. None, if the file is not stored on the archive.
3. Refer to: POSIX.1 Conformance Document Sections 10.1.1 and 10.1.2.2.

**Rationale**

Because an archive can contain non-portable file names, it is necessary for an archive reading utility to be able to generate a file and store the data associated with a non-portable file name when this is encountered on the archive. There may be a need to generate a number of such file names in the same directory and the specification should detail the algorithm used to generate these file names.

Reference

XPG3 Volume 3 Page 151 – Utilities

## MULTI VOLUME ARCHIVES

**Question 5:** *How does the archive reading utility determine which file to read as the next volume when an end-of-file or end-of-media condition is encountered?*

**Answer:**

Format	Method
Extended tar	Prompts when ready for the next volume and asks the user to type “go” when ready to proceed. There is no way to specify the device – the initial device is used.
cpio	Prompts that it has reached the end of the medium and asks the user to type the device/file name for the next archive when ready.

**Options:**

Description of method used by each utility.

Refer to: POSIX.1 Conformance Document Section 10.1.3.

**Rationale**

In many cases the utility will prompt the user for the path name of the device to use for the next volume. There may be extensions to the utility syntax which allow the definition of alternate addresses for subsequent volumes.

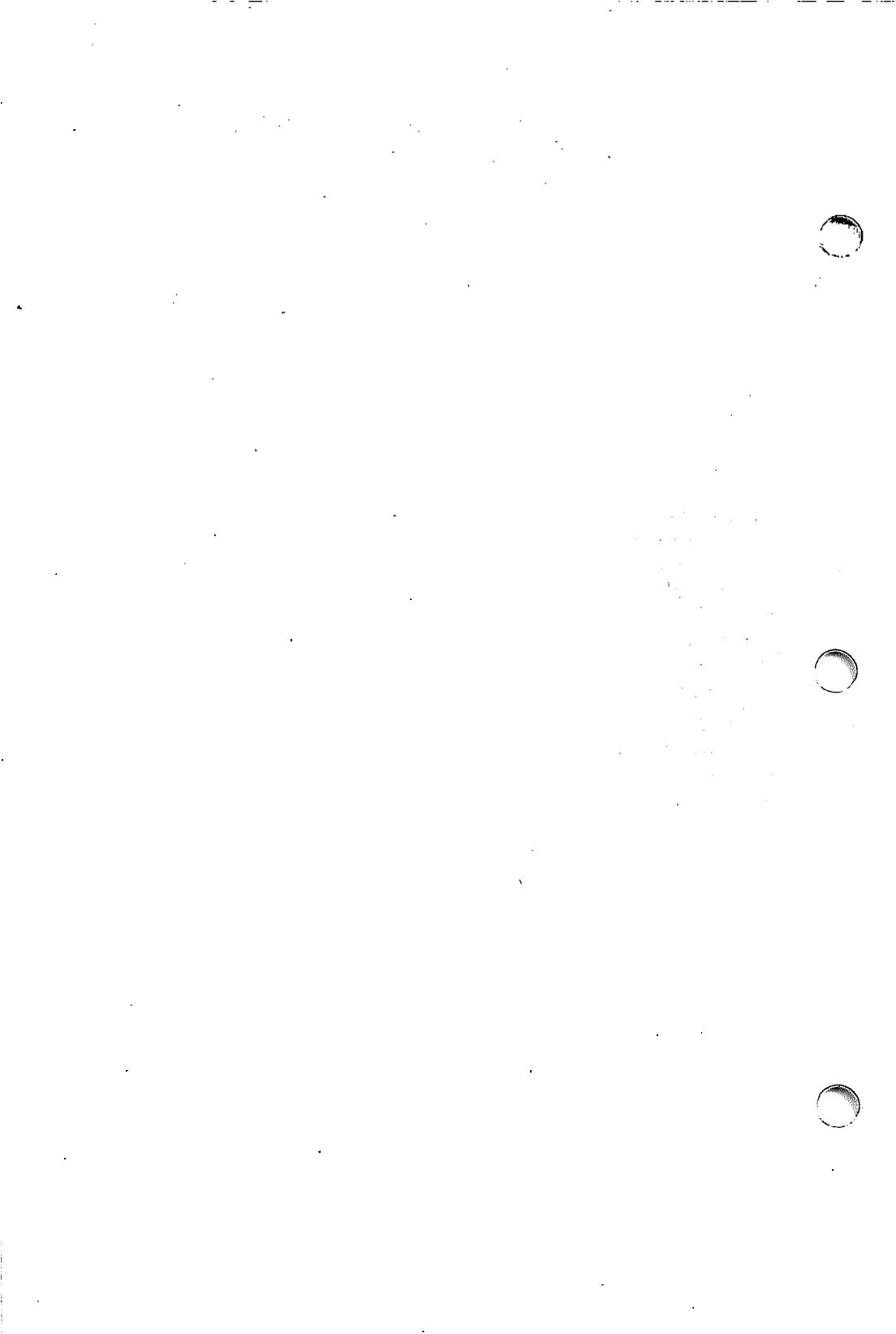
**Reference**

XPG3 Volume 3 Pages 151-2 – Utilities

# International Supplement Reference Manual

## CONTENTS

chrtbl(1M)  
colldf(1P)  
gencat(1P)  
iconv(1P)  
loadfont(1)  
showcat(1P)  
ttypmap(1)  
catclose(3P)  
catgets(3P)  
catopen(3P)  
localeconv(3P)  
nl\_langinfo(3P)  
setlocale(3P)  
strcoll(3P)  
strerror(3P)  
strxfrm(3P)  
gencat(4P)  
loadfont(4)  
charmap(5P)  
langinfo(5P)  
locale(5P)





## NAME

chrtbl – generate character classification and conversion tables

## SYNOPSIS

chrtbl [file]

## DESCRIPTION

The *chrtbl* command creates a character classification table and an upper/lowercase conversion table. The tables are contained in a byte-sized array encoded such that a table lookup can be used to determine the character classification of a character or to convert a character (see *ctype(3C)*). The size of the array is 257\*2 bytes: 257 bytes are required for the 8-bit code set character classification table and 257 bytes for the uppercase to lowercase and lowercase to uppercase conversion table.

*chrtbl* reads the user-defined character classification and conversion information from *file* and creates two output files in the current directory. One output file, *ctype.c* (a C-language source file), contains the 257\*2-byte array generated from processing the information from *file*. You should review the content of *ctype.c* to verify that the array is set up as you had planned. (In addition, an application program could use *ctype.c*.) The first 257 bytes of the array in *ctype.c* are used for character classification. The characters used for initialising these bytes of the array represent character classifications that are defined in */usr/include/ctype.h*; for example, **\_L** means a character is lower case and **\_S|\_B** means the character is both a spacing character and a blank. The last 257 bytes of the array are used for character conversion. These bytes of the array are initialised so that characters for which you do not provide conversion information will be converted to themselves. When you do provide conversion information, the first value of the pair is stored where the second one would be stored normally, and vice versa; for example, if you provide **<0x41 0x61>**, then **0x61** is stored where **0x41** would be stored normally, and **0x41** is stored where **0x61** would be stored normally.

The second output file (a data file) contains the same information, but is structured for efficient use by the character classification and conversion routines (see *ctype(3C)*). The name of this output file is the value of the character classification *chrclass* read in from *file*. This output file must be installed in the */lib/chrclass* directory under this name by someone who is superuser or a member of group *bin*. This file must be readable by user, group, and other; no other permissions should be set. To use the character classification and conversion tables on this file, set the environmental variable **CHRCLASS** (see *environ(5)*) to the name of this file and export the variable; for example, if the name of this file (and character class) is *xyz*, you should issue the commands: **CHRCLASS=xyz ; export CHRCLASS** .

If no input file is given, or if the argument **-** is encountered, *chrtbl* reads from the standard input file.

The syntax of *file* allows the user to define the name of the data file created by *chrtbl*, the assignment of characters to character classifications and the relationship between uppercase and lowercase letters. The character classifications recognised by *chrtbl* are:

<b>chrclass</b>	name of the data file to be created by <i>chrtbl</i> .
<b>isupper</b>	character codes to be classified as uppercase letters.
<b>islower</b>	character codes to be classified as lowercase letters.
<b>isdigit</b>	character codes to be classified as numeric.
<b>isspace</b>	character codes to be classified as a spacing (delimiter) character.
<b>ispunct</b>	character codes to be classified as a punctuation character.
<b>iscntrl</b>	character codes to be classified as a control character.
<b>isblank</b>	character code for the space character.
<b>isxdigit</b>	character codes to be classified as hexadecimal digits.
<b>ul</b>	relationship between uppercase and lowercase characters.

Any lines with the number sign (#) in the first column are treated as comments and are ignored. Blank lines are also ignored.

A character can be represented as a hexadecimal or octal constant (for example, the letter a can be represented as 0x61 in hexadecimal or 0141 in octal). Hexadecimal and octal constants may be separated by one or more space and tab characters.

The dash character (–) may be used to indicate a range of consecutive numbers. Zero or more space characters may be used for separating the dash character from the numbers.

The backslash character (\) is used for line continuation. Only a carriage return is permitted after the backslash character.

The relationship between uppercase and lowercase letters (**ul**) is expressed as ordered pairs of octal or hexadecimal constants: *<upper-case\_character lower-case\_character>*. These two constants may be separated by one or more space characters. Zero or more space characters may be used for separating the angle brackets (< >) from the numbers.

#### EXAMPLE

The following is an example of an input file used to create the ASCII code set definition table on a file named *ascii*:

```
chrclass  ascii .
isupper   0x41 - 0x5a
islower   0x61 - 0x7a
isdigit   0x30 - 0x39
isspace   0x20 0x9 - 0xd
ispunct   0x21 - 0x2f  0x3a - 0x40 \
           0x5b - 0x60  0x7b - 0x7e
iscntrl   0x0 - 0x1f  0x7f
```

```

isblank  0x20
isxdigit 0x30 - 0x39 0x61 - 0x66 \
          0x41 - 0x46
ul       <0x41 0x61> <0x42 0x62> <0x43 0x63> \
          <0x44 0x64> <0x45 0x65> <0x46 0x66> \
          <0x47 0x67> <0x48 0x68> <0x49 0x69> \
          <0x4a 0x6a> <0x4b 0x6b> <0x4c 0x6c> \
          <0x4d 0x6d> <0x4e 0x6e> <0x4f 0x6f> \
          <0x50 0x70> <0x51 0x71> <0x52 0x72> \
          <0x53 0x73> <0x54 0x74> <0x55 0x75> \
          <0x56 0x76> <0x57 0x77> <0x58 0x78> \
          <0x59 0x79> <0x5a 0x7a>

```

**FILES**

`/lib/chrclass/*` data file containing character classification and conversion tables created by *chrtbl*

`/usr/include/ctype.h` header file containing information used by character classification and conversion routines

**SEE ALSO**

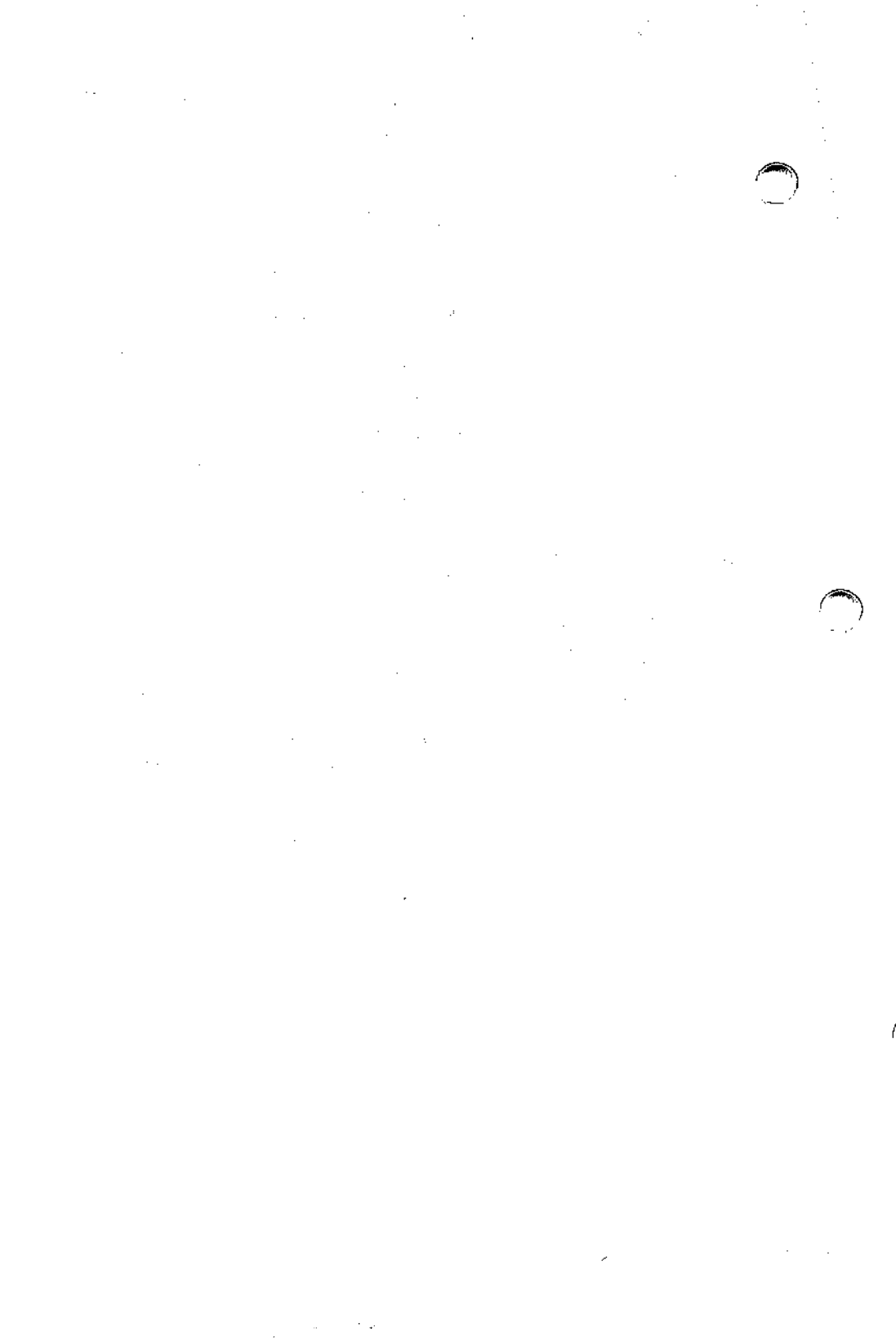
`ctype(3C)`, `environ(5)` in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**DIAGNOSTICS**

The error messages produced by *chrtbl* are intended to be self-explanatory. They indicate errors in the command line or syntactic errors encountered within the input file.

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.



**NAME**

colldf – generate collation table

**SYNOPSIS****colldf** [-c] [-fcharmap] [-iinputfile] [-s] locale**DESCRIPTION**

The *colldf* utility converts collation source definitions into a format usable by the *strcoll*(3P) and *strxfrm*(3P) functions, as well as in sorting and regular expression processing.

The *colldf* command has the following options:

- c** A collation table is created if warning messages have been issued. (Normally both error and warning messages cause the command to terminate without creating the collation table.)
- f charmap** The path name of a file containing a mapping of character symbols and collating element symbols to actual character encodings. This option must be specified if symbolic names (other than collating symbols defined in a collating-symbol keyword) are used. If the name does not contain a "/", the program will assume that the charmap is located in the directory */lib/charmap*.
- i inputfile** The path name of a file containing the source definitions. If this option is not present, source definitions are read from standard input.
- s** When this flag is used, the *colldf* command will not print warning messages.

The *locale* argument identifies the target *locale*. If the argument contains one or more slash characters or consists of dot (.), it will be interpreted as an absolute path name for the directory in which the created collation table will be stored. Otherwise, the argument is interpreted as the name of a directory under */lib/locale/ISC*. The created collation table is stored in a file named *LC\_COLLATE* within the *locale* directory.

The character set mapping file specified as the *charmap* option-argument is described under *charmap*(5P).

The collation source definition file contains statements describing the desired collation behaviour. Each statement consists of a keyword, optionally followed by arguments and by collation order entries. The following keywords are recognised:

- LC\_COLLATE** This keyword must be the first in the file.
- collating-symbol** This keyword names symbolic names used in collation order entries.
- collating-element** This keyword defines multi-character collating elements.
- substitute** This keyword describes regular expression-type substitutes.

**order\_start** This keyword defines the collation evaluation direction and immediately precedes the collation order entries.

**order\_end** This keyword immediately follows the last collation order entry.

**END LC\_COLLATE**

This keyword must be the last in the file.

Each collation order entry consists of a character, a collating symbol, or a multi-character collating element, followed by weight information.

The detail format of the collation definition source is described in the “International Supplement User’s Manual.”

The setting of the `LC_*` environment variables does not affect the behaviour of the `colldef` command.

**ERRORS**

If an error is detected, no collation tables are created.

If warnings occur, specifying the `-c` option will cause permanent output to be created. The following conditions will cause warning messages to be issued:

1. If a symbolic name not found in the `charmap` file is used to define a collating element, the element is discarded and a warning message issued.
2. If the number of arguments to the `order` keyword exceeds the `{COLL_WEIGHTS_MAX}` limit, which is defined in the file `/usr/include/sys/limits.h`, a warning message will be issued.

**FILES**

`/lib/locale/ISC/*/LC_COLLATE`  
`/lib/charmap/*`

**SEE ALSO**

`strcoll(3P)`, `strxform(3P)`, `charmap(5P)`, `locale(5P)`.  
 “International Supplement User’s Manual.”

**NAME**

**gencat** — generate a formatted message catalogue

**SYNOPSIS**

**gencat** **-c** *catfile* *msgfile* ...

**DESCRIPTION**

The *gencat* utility merges the message text source file(s) *msgfile* into a formatted message catalogue *catfile*. The file *catfile* will be created if it does not already exist. If *catfile* does exist, its messages will be included in the new *catfile*. If set and message numbers collide, the new message text defined in *msgfile* will replace the old message text currently contained in *catfile*.

If the **-c** option is specified on the command line or the existing *catfile* was generated with the **-c** option, the *catfile* will be “confidential,” that is, it will not be translatable into a message text source file by the *showcat*(1P) utility.

In this implementation, *gencat* makes the following interpretations with respect to the format of a message text source file (see *gencat*(4P) for the format of a message text source file as defined in the *X/Open Portability Guide, Volume 3, XSI Supplementary Definitions*, Section 5.2.1, “Message Text Source Files”):

1. Set number ordering relates to set numbers from both **\$set** and **\$delset** directives. Thus, the following is illegal:

```
$delset 2
$set 1
```

2. A set or message number can be equal to the preceding one. Thus, the following is legal:

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

3. If any line in a message text source file (not just a text string) ends with a backslash (\), that is treated as a line continuation.

This utility operates in an 8-bit transparent manner.

**ERRORS**

If there are any errors in the course of processing any *msgfile* or, if it exists, *catfile*, *gencat* will not generate a new *catfile* and its exit status will be 1. Under certain error conditions, *gencat* will continue processing all *msgfiles* before exiting with an error status. These conditions include:

1. If *catfile* exists, either it cannot be opened, there is an error reading it, or it has corrupted data.
2. For any *msgfile*, either it cannot be opened or it has a syntax error. For any other errors, exit will be immediate.

**WARNINGS**

The following conditions will not generate an error but will cause a warning message to be printed:

1. There is an attempt to delete a message or set that doesn't exist.
2. The specified *catfile* is an empty file.
3. A temporary file cannot be unlinked.

**NOTES**

Using non-contiguous set or message numbers, using a set number other than 1 as the first set, or using a message number other than 1 as the first message of a set will cause the size of *catfile* to be larger than using only contiguous numbers starting with 1.

Message catalogues produced by *gencat* are binary encoded, which means that their portability cannot be guaranteed between different types of machines. Thus, just as C programs need to be recompiled for each type of machine, so message catalogues must be recreated via *gencat*.

**SEE ALSO**

showcat(1P), gencat(4P).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.



**NAME**

iconv – codeset conversion

**SYNOPSIS**

iconv [ **-S** default-char-specification ] **-f** fromcode **-t** tocode [ file ]

**DESCRIPTION**

The *iconv* utility converts the encoding of characters in *file* from one codeset to another and writes the results to standard output. The input and output codesets are identified by *fromcode* and *tocode*, respectively. If no *file* argument is specified on the command line, *iconv* reads the standard input.

Character encodings in either codeset may include single-byte values (e.g., for ISO standard **ISO 8859-1:1987** characters) or multi-byte values (e.g., for certain characters in ISO standard **ISO 6937:1983**). A character in the input stream that does not have a corresponding conversion in the “to” codeset defaults to the underscore character (`_`) in the output stream.

The *iconv* utility contains six built-in conversion tables. When the **-f** and **-t** file specifications are *both* taken from the following list, the built-in conversion tables are used:

<b>437</b>	IBM codepage 437
<b>850</b>	IBM codepage 850
<b>8859</b>	ISO/IEC 8859-1 codeset

If a path name does not contain a slash (`/`), the program assumes that the file is located in the directory `/lib/charmap`. Otherwise, *fromcode* and *tocode* are path names for the *charmap* files.

The **-S** command option allows the default character to be dynamically changed. The format of *default-char-specification* is either of the following:

```
< new-default-char >
"\dnnn"
"\xnn"
"\nnn"
```

The first specification, which must be a valid charmap symbol from the file defined as the “to” file, is only valid if charmap files rather than the built-in tables are specified.

The latter three formats can only be used with the built-in tables and specify the code value of the new default character. When the character following the “\” is `d`, then *nnn* is a decimal value, e.g., 43 for the plus sign. When the character following the “\” is `x`, then *nn* is a hexadecimal value, e.g., 2B for the plus sign. When the character following the “\” is numeric, then *nnn* is an octal value, e.g., 53 for the plus sign.

**EXAMPLES**

1. The following example uses the built-in tables to convert from the ISO/IEC 8859-1 codeset to the IBM codepage 437 codeset and uses the plus character (+) as the default output character:

```
iconv -f 8859 -t 437 -S "\d43" file
```

2. In the following example, both the *fromcode* file **8859-4.cmap** and the *tocode* file **865.cmap** must exist in the directory **/lib/charmap**:

```
iconv -f 8859-4.cmap -t 865.cmap -S <plus-sign> infile >outfile
```

3. In the following example, the *fromcode* file is located in the current directory. The *tocode* file being utilized is in the **mydir** subdirectory of the current directory:

```
iconv -f ./8859-5.cmap -t mydir/866.cmap file
```

4. The following example converts the contents of the file **mail.x400** from codeset **ISO 6937:1983** to **ISO 8859-1:1987** and stores the results in the file **mail.local**:

```
iconv -f 6937.cmap -t 8859.cmap mail.x400 > mail.local
```

#### NOTE

8859 is used as a synonym for 8859-1, both in the built-in table (**8859**) and in the *charmap* file (**/lib/charmap/8859.cmap**).

#### SEE ALSO

**charmap(5P)**.

**NAME**

**loadfont** – list or change font information in the RAM of the video card

**SYNOPSIS**

**loadfont**

**loadfont** **-f** *filename*

**loadfont** *codepage*

**loadfont** **-l**

**loadfont** **-d**

**loadfont** **-m** *mode*

**DESCRIPTION**

The *loadfont* utility allows a user to load and activate a different font into the RAM of the video card used by the console of the INTERACTIVE UNIX Operating System. It can also be used to display information about the font currently in use. In addition, the **-m** option can be used to change the size of the characters on the screen; it can also be used to change the number of lines or colors, e.g., to run an application at the console at 43 lines at a time instead of 25. *loadfont* will always read from standard output; this will allow a system administrator to use it from a remote terminal.

**Options**

*loadfont*

When used without arguments, *loadfont* displays the different ways the command can be used, as shown in the synopsis.

*loadfont* **-f** *filename*

This command reads the contents of *filename* and subsequently loads the font specified in the file into the RAM of the video card. If the file does not have the correct format, an error message is produced.

*loadfont* *codepage*

If *codepage* is the name of a hard-coded font available for the current font size, this font will be loaded into the RAM of the video card and activated. Available font names are listed when the **-l** option is used. If the *codepage* argument specified is not the name of a valid font, an error message will be produced.

*loadfont* **-l**

This option displays a short description of the fonts that are hard-coded into the program and the name that can be passed as a *codepage*. Only the fonts that match the current font size are listed. *loadfont* **-l** also displays the different character modes supported by *loadfont* and the exact name that should be used with the **-m** option. Here is a sample output:

Codepages supported for this size font are:

<i>Name</i>	<i>Description</i>
437	IBM 437 codepage
8859	ISO 8859-1 codeset
8859g	ISO 8859-1 with graphics
850	IBM 850 codepage

Different possible text modes supported are:

<i>Name</i>	<i>Description</i>
E80x43	EGA 80 columns 43 lines
E40x25	EGA 40 columns 25 lines
E80x25	EGA 80 columns 25 lines
V40x25	VGA 40 columns 25 lines
V80x25	VGA 80 columns 25 lines

8859g means the 8859-1 codeset with box-drawing characters in column 9 of the table (characters 0x90 to 0x9a).

### *loadfont -d*

This *reads* the font information from the video RAM and *writes* it to standard output in a format compatible with the Binary Distribution Format version 2.1 as developed by Adobe Systems, Inc.

### *loadfont -m mode*

This will attempt to change the mode of the console as specified. This will result in having a different font size and/or different number of lines and columns on the screen. The *mode* that can be specified should be one of the choices listed above in the *loadfont -l* output. If an invalid argument is specified, an error message is produced.

## Fonts

A font is the representation of characters by images. The need to use different fonts can be imposed by:

1. The codeset used to represent the characters internally.
2. The resolution used to display the characters.

Each font contains exactly 256 images. All fonts supported are fixed size (constant width and constant height), i.e., each character takes the same amount of space on the screen. When the monitor is not being used in graphics mode, the *loadfont* utility allows a user to modify the font used by the video card, so different images are displayed on the screen of the console for the various characters. Depending on the type of video card used, different text modes can be supported by the same video card. They typically differ by the number of pixels used to represent a single character. For each character, the same number of pixels is used. For the standard video cards, the different resolutions supported (all or a subset) are:

- 8 by 8 ( 8 horizontally and 8 vertically)
- 8 by 14
- 8 by 16

When *loadfont* is invoked to modify the existing font, it will attempt to do so for the font size currently in use. Use the `-m` option to switch to another font size.

### *loadfont* and *ttymap*

There is an almost one-to-one relationship between the use of the *loadfont* utility and the *ttymap* utility. Whereas *loadfont* is used to list or modify the images that correspond with the various characters, the *ttymap* utility is used to determine how characters are generated from the keyboard and which code (a single byte code) will be used to represent the character internally. The default representation is the IBM extended ASCII codeset, often also referred to as "IBM codepage 437." A *ttymap* sample input file is supplied that can be used for this codeset on a console with a U.S. keyboard (`usa.map`). When a different keyboard is used, a different *ttymap* input file is required (e.g., `french.map` for a French keyboard).

When a different codeset is used, both a different *ttymap* input file and a different font are required. For the most popular *codesets*, fonts are hard-coded into the *loadfont* program for the 8 by 16 resolution (see "Fonts"). If these fonts do not satisfy your needs (because you want to use a different font size or because a customized font is required, e.g., a Greek font), a *loadfont* description file to be used with the `-f` option is needed. A sample file that describes the IBM extended ASCII font for an 8 by 16 resolution is supplied (`vga437.bdf`). A second sample file, `646g.bdf`, contains a font file for German ASCII. See *ttymap(1)* and *loadfont(4)* for additional details.

### WARNING

When an attempt is made to switch to a mode that the video card does not support (e.g., a switch to EGA on a VGA card that has no EGA mode) you will get a blank screen. There is nothing wrong with the system; simply type in the command to set the mode back, e.g.:

```
loadfont -m V80x25
```

### FILES

```
/usr/lib/loadfont/vga437.bdf
```

sample Bitmap Distribution Format (BDF) file for IBM 437 font on a VGA

```
/usr/lib/loadfont/646g.bdf
```

sample BDF file for German ASCII

### SEE ALSO

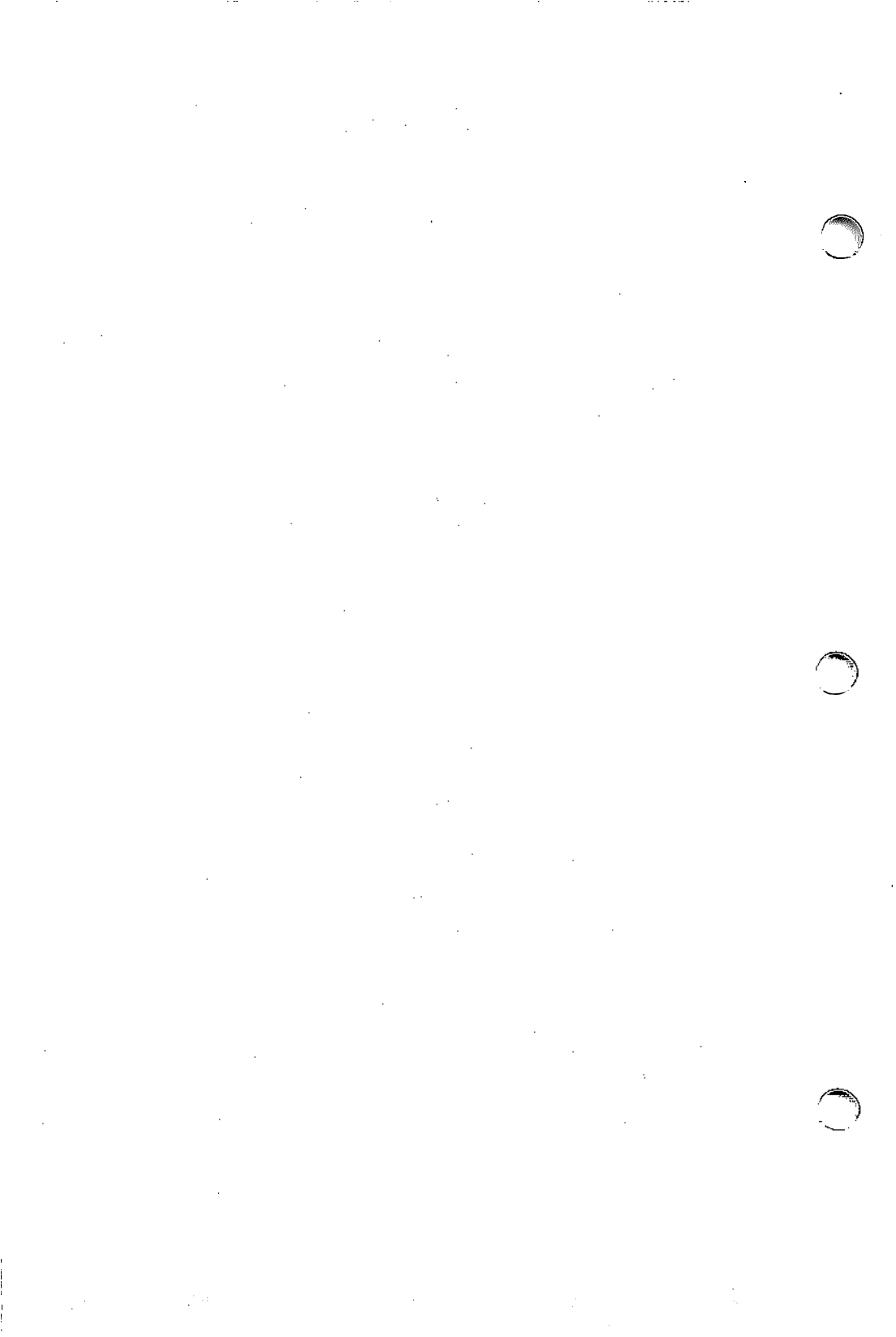
*ttymap(1)*.

*display(7)* in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.

*loadfont(4)* in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

### NOTE TO USERS

This entry is reprinted from the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.



**NAME**

showcat — generate a message catalogue source file from a binary message catalogue

**SYNOPSIS**

showcat msgfile catfile

**DESCRIPTION**

*showcat* generates a message catalogue source file from a binary message catalogue (i.e., the opposite of *genocat*(1P)). If the binary file is “confidential” (i.e., it was generated by *genocat* -c), no attempt is made to translate it to source and a corresponding message is printed. If the binary file is not confidential but is not in the proper format (i.e., it is corrupted), then the source file will not be generated.

The generated source file uses quoting, with the double quote as the quote character. For the message text, printable characters in the locale are written as-is in the source file. For the other characters, if there is a defined escape sequence, that is written; otherwise, an octal bit pattern is written.

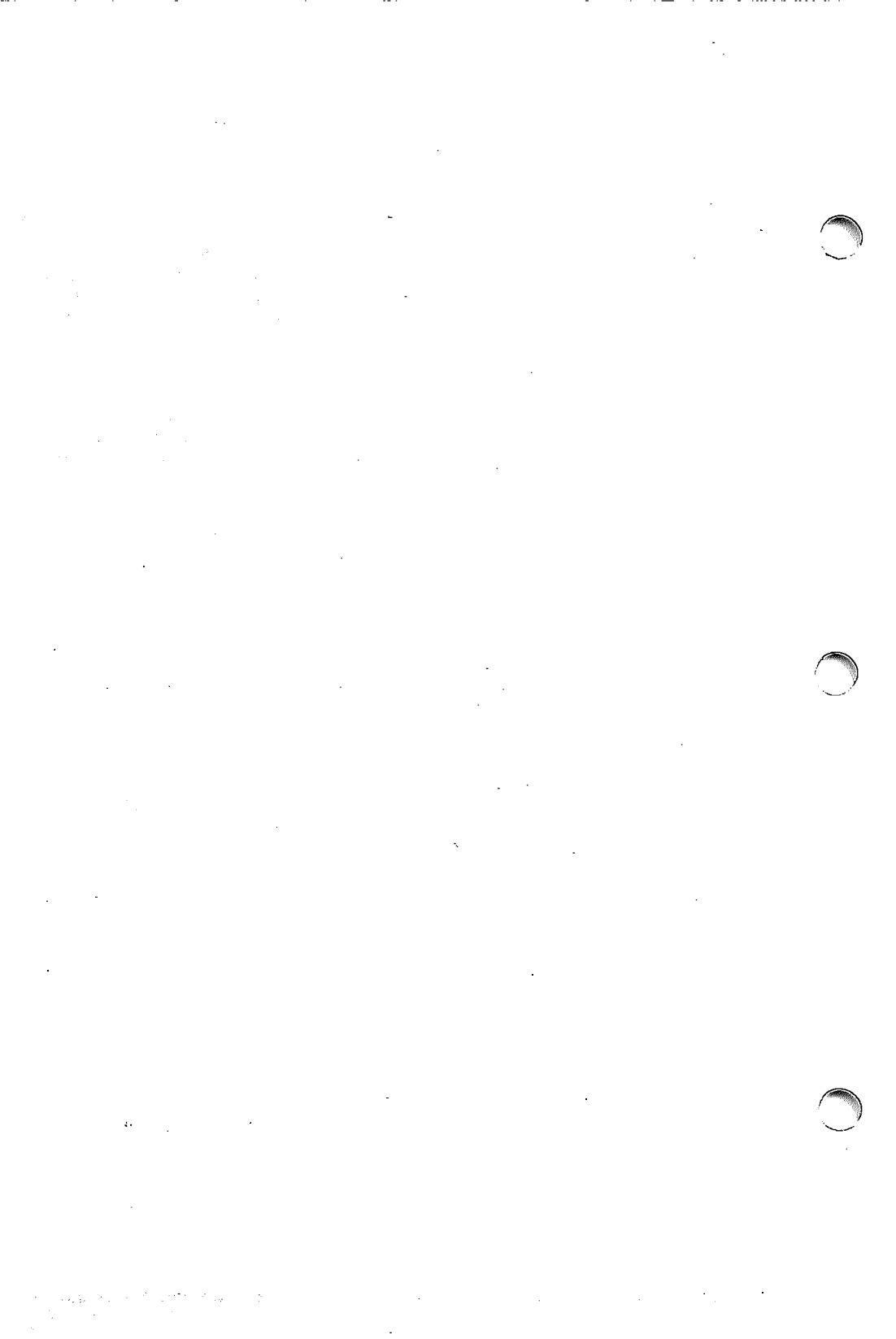
**EXAMPLE**

The following is an example of the source file format generated by *showcat*:

```
$quote "
$set 1
1 "This is set 1, message 1."
2 "This is set 1, message 2."
3 "This is set 1, message 3. It is continued where there was a\n\
newline character in the input."
$set 3
1 "This is set 3, message 1."
3 "This is set 3, message 3."
5 "This is set 3, message 5. The following, within single quotes, is\n\
the representation of the character with value 200 octal\n\
when showcat is run in the C locale: '\200'."
```

**SEE ALSO**

*genocat*(1P).





**NAME**

ttypmap — set terminal mapping and scancode translation

**SYNOPSIS**

ttypmap *mapfile*

ttypmap -r

ttypmap -d

**DESCRIPTION**

*ttypmap* is a utility that permits a user to activate character mapping on input and output for the user's terminal. This same utility can be used for regular terminals as well as for scancode devices such as the AT console. It makes full use of all the features of the terminal (tty) driver and the keyboard display driver that support such mapping.

The command *ttypmap mapfile* reads the contents of the file *mapfile* and sets the corresponding mapping as supported by the terminal driver and/or keyboard/display driver. The layout of the *mapfile* and the functionality supported by both drivers are described below.

**ttypmap -d** disables the current mapping by the terminal driver.

**ttypmap -r** resets the scancode translation back to that of a U.S. PC keyboard.

**Terminal Mapping**

The original UNIX operating system was written to support the ASCII codeset. ASCII is one of many standards to represent a number of characters internally as certain numbers. Typical for ASCII is that it supports 128 different characters, each represented by a single byte of which the 8<sup>th</sup> bit is not used. Many UNIX system applications, including the shell, took advantage of this. Starting with UNIX System V Release 3.1, most of these applications have been modified to properly support characters represented as a byte with the 8<sup>th</sup> bit set as well. This means that now 256 characters can be supported at the same time. However, a consistent coding convention needs to be applied. In the IBM PC world, an 8-bit coding referred to as IBM extended ASCII has been used for several years; MS-DOS users are quite familiar with that. In heterogeneous UNIX System environments, a different codeset, called ISO8859, has been promoted. In both codesets, characters found in the ASCII codeset are represented in the same way. The other 128 characters are encoded differently, however, and some characters found in one codeset will be missing in the other. The INTERACTIVE UNIX Operating System supports both codesets; actually, it supports any 8-bit one byte codeset.

To be able to use characters from the French, German, Finnish, and other alphabets, several terminals are available on the market that generate 7-bit codes but display the above-mentioned characters on the screen instead of the ones found on a U.S. terminal. On the keyboard there are an equal number of keys, but there are different characters on the key caps. Others, such as a DEC VT220, will support 256 different characters at a time but use their own proprietary codesets.

Assume you are using the INTERACTIVE UNIX Operating System with a console and a French 7-bit terminal connected to the serial port. If you edit a file on the terminal and use the French character *é* in

text, the terminal will actually generate the ASCII code 123, which is the code normally used for the left curly brace. If you look at the edited file on the console, the letter will actually appear to be a curly brace. Therefore, input and output mapping should be supported by the terminal driver to allow the consistent use of one single codeset throughout the system. The INTERACTIVE UNIX Operating System supports all mapping features that are now standard in the System V Release 3.2 terminal driver, as well as some enhancements by INTERACTIVE Systems Corporation.

### Input mapping

On input, any byte can be mapped to any byte. Using the example above, you could map 123 to 130, the code used for *é* in the IBM extended ASCII codeset.

### Output mapping

On output, any byte can be mapped to either a byte or a string. In the above example, 130 would be mapped back to 123 to properly display the character on the screen. If the connected device is a printer that does not support the *é* character, it could be mapped to the string:

e BACKSPACE '

### Dead keys

On typewriters, keys can be found that behave slightly differently than all the others, because when you press them, the printing wheel of the typewriter does not move. CTRL (^) is such a character. When it is followed by an *e*, the letter *ê* is generated. This is called a deadkey or a non-spacing character. The terminal driver supports the use of deadkeys. Typically, the ^ character and the umlaut character are used as deadkeys.

### Compose sequences

Characters can also be generated using a compose sequence. A dedicated character called the "compose character" followed by two other keystrokes will generate a single character. As an example, COMPOSE followed by the plus and the minus sign could generate the plus/minus sign ( $\pm$ ). Compose sequences can also be used as an alternative for deadkeys, e.g., "COMPOSE ^ e" instead of "e" to get *ê*.

### Decimal representation

Rarely used characters can be generated by pressing the compose key followed by three digits.

### Toggle key

An optional toggle key can be defined to temporarily disable the current mapping from within an application. This can be useful when, for example, a German programmer wants easy access to the curly braces and the brackets.

### Scancode Mapping

The keyboards of the console and some other peripherals such as SunRiver workstations behave differently than those of regular terminals. They generate what are called *scancodes* and you will also find a number of keys on these keyboards, such as the ALT key, that are not

found on regular terminals. Scancodes generated by PC keyboards typically represent the location of the key on the keyboard. The keyboard driver has to properly translate these scancodes. The different national variants of a PC keyboard not only have non-English characters printed on some of the keycaps, but the order of some of the keys is different as well. Without changing the scancode translation, a French user would type A and see a Q on his screen. Several status keys can influence the translated code as well. The keyboard driver, and thus the *ttymap* program, makes a distinction between two sets of key combinations that can be translated.

### Function keys

Up to 60 key combinations are recognised as function keys. The first 12 are the 12 function keys of a 101-key PC-keyboard (the first 10 on an 84-key keyboard).

If you do not know whether you have an 84- or 101-key keyboard, you can use the following scheme to determine which type you have:

If your keyboard has arrow keys that are separate from the ones on the numeric keypad, then you have a 101-key keyboard.

If the arrow keys on your keyboard are located on the numeric keypad only, then you have an 84-key keyboard.

F13 to F24 are the same keys used in combination with **SHIFT**, F25 to F36 when used with **CTRL**, and F37 to F48 when used with **CTRL** and **SHIFT** together. F49 to F60 are the keys on the numeric keypad, in the following order:

```

7
8
9
-
4
5
6
+
1
2
3
INS
```

Each of these function keys can be given a string as a value. The total length of all strings should not exceed 512 characters. See *keyboard(7)* for a list of default values.

### Regular keys

Scancodes generated by all keys on the PC keyboard can be translated in a different way as well. For each key, a different translation can be specified for each of the following four cases:

1. The key is pressed.
2. The key and the **SHIFT** key are pressed simultaneously.
3. The key and the **ALT** key are pressed simultaneously.
4. The key, the **SHIFT**, and the **ALT** keys are pressed simultaneously.

For each of these cases, the scancode can be translated into one of the following:

- a single byte
- a single byte preceded by ESC N
- a single byte preceded by ESC O
- a single byte preceded by ESC I

Internally, special bits are set to indicate that an escape sequence needs to be generated. Other bits are used to indicate whether the translated code should be influenced by some special keys.

#### NUM LOCK

If the **NUM LOCK** bit is set, the regular and **SHIFT** values are swapped, as are the **ALT** and **SHIFT ALT** values, whenever the **NUM LOCK LED** is on. By default, only the keys on the numeric keypad have this bit set. That is why these keys generate 7, 8, 9, etc. when the **NUM LOCK LED** is on, which is the same value that would be produced if **SHIFT** were used with these keys.

#### CAPS LOCK

This has the same effect as the **NUM LOCK** key. By default, this bit is set for all letters and not set for punctuation signs.

**CTRL** When a key is translated into a single byte (no escape sequence) and this bit is set, the corresponding control character will be generated when the **CTRL** key is pressed simultaneously. This is equally valid for the **SHIFT**, **ALT**, and **SHIFT ALT** combination. When this bit is not used, the **CTRL** key combination will not generate anything.

#### mapfiles

This section describes the layout of a *mapfile* that is read by the *ttymap* program.

A *mapfile* is a text file that consists of several **sections**. A sharp sign (**#**) can be used to include comments. Everything following the **#** until the end of the line will be ignored by the *ttymap* program. Inside a line, C-style comments can be used as well. The beginning of each section is indicated by a *keyword*. Spaces and tabs are silently ignored and can be used at all times to improve readability. All but one section, the one that defines the *compose character*, can be left out. The order in which the different sections should appear is predefined. Here is the list of keywords in the order they should appear:

**input:**  
**toggle:**  
**dead:**  
**compose:**  
**output:**  
**scancodes:**

Characters can be described in several different ways. ASCII characters can be described by putting them between single quotes. For example:

'a' '{'

Between single quotes, control characters can be listed by using a circumflex sign before the character that needs to be quoted. For example:

```
'^x'
```

When a backslash (\) is used, what follows will be interpreted as a decimal, octal (leading zero), or hexadecimal (leading x or X) representation of the character, although in this case the use of single quotes is not mandatory. For example:

```
'\x88'
```

is the same as:

```
0x88 (zero needed when not quoted)
```

and:

```
'\007'
```

is the same as:

```
007
```

When strings are needed, a list of character representations should be used. Quoted strings will be supported in the future.

The following paragraphs describe what goes in each section.

#### **Input section**

The input section describes which input characters should be mapped into a single byte. A very small sample input section could be:

input:

```
'A' 'B'      # map A into B on input
'#' 0x9c     # map sharp sign into pound sign
```

#### **Toggle section**

The toggle section is a one-line section that defines which key is to toggle between mapping and no mapping. For example:

toggle:

```
'^y'        # ctrl y is the toggle key
```

#### **Deadkey section**

The deadkey section defines which keys should be treated as deadkeys. A **dead:** keyword followed by the specification of the character appears in this section for each deadkey. The subsequent lines describe what key should be generated for each key following the deadkey. A deadkey followed by a key not described in this part of the *mapfile* will not generate any key and a beep tone will be produced on the terminal. For example:

```
dead: '^'      # circumflex is a deadkey
' ' '^'       # circumflex followed by space generates circumflex
'e' 0x88      # circumflex followed by e generates e circumflex
dead: '"'     # double quote used as a deadkey
' ' '"        # double quote space generates double quote
'a' 0x84      # double quote a generates an umlaut
```

**Compose section**

The first line of this section describes what the compose character is. That line should always be present in the *mapfile*. Subsequent lines consist of three character representations indicating each time that the third character needs to be generated on input when the compose character is followed by the first two. Compose sequences with the same first character should be grouped together. For example:

```
compose: '^x'
''' 'e' 0x89 # e with umlaut is generated when typing ^x " e
''' 'a' 0x84 # a with umlaut
'e' ''' 0x89 # e with umlaut is generated when typing ^x e "
'a' ''' 0x84 # a with umlaut
```

The following example would give the wrong result. All lines starting with the same character specification should be grouped together.

```
compose: '^x'
''' 'e' 0x89 # e with umlaut is generated when typing ^x " e
'e' ''' 0x89 # e with umlaut is generated when typing ^x e "
''' 'a' 0x84 # a with umlaut
'a' ''' 0x84 # a with umlaut
```

**Output section**

This section describes the mapping on output, either single byte to single byte, or single byte to string. A string is specified as a series of character specifications. For example:

```
output:
0x82 '{' # map e with accent to { to display e with accent
'^u' '("K"l"l")' # print (KILL) when kill character is used
```

**Scancodes section**

This section will only have an effect when your terminal is a scancode device. No error message will be produced when this section is mistakenly in your *mapfile*, because the *ttymap* program will find out whether the terminal is a scancode device or not. The lines in this section can have two different formats. One format will be used to describe what the values of the function keys must be. The other format describes the translation of scancodes into a byte or an escape sequence. No specific order is required.

**Function keys**

Here is an example of a line defining a string for a function key:

```
F13 'd"a"t"e"\n' # SHIFT F1 is the date command
```

The numbering convention of the functionkeys is described in a previous section. Currently, the use of quoted strings such as "*date*\n" is not supported.

**Scancodes**

Specifying how to translate a scancode is a more complex task. The general format of such a line is:

```
scancode normal shift alt shiftalt flags
```

**scancode** should list the hexadecimal representation of a scancode generated by a key (unquoted). How keys correspond with scancodes can be found in *keyboard*(7).

**normal**, **shift**, **alt** and **shifalt** are character representations in one of the formats described throughout this document, optionally followed by one of the following special keywords:

**IC** This indicates that the key is influenced by the **CTRL** key.

**IN** This indicates that **ESC N** should precede the specified character.

**IO** This indicates that **ESC O** should precede the specified character.

**I[** This indicates that **ESC [** should precede the specified character.

The **normal** field defines how the scancode is translated when no other key is pressed, the **shift** field defines the translation for when the **SHIFT** key is used simultaneously, the **alt** field specifies what to do when the **ALT** key is pressed together with this and the **shifalt** field contains the information on what to generate when both the **SHIFT** and **ALT** keys are pressed.

All five fields must be filled in. When no translation is requested (that is, the current active translation does not need to be changed) a dash (-) can be used. The sixth field is optional. This field can contain the special keyword **CAPS** or **NUM** or both, to indicate whether or not the **CAPS LOCK** key or **NUM LOCK** key status have any effect. Here is a sample line that describes the default translation for the 'Q' key:

```
0x10 'q'IC 'Q'IC 'q'IN 'Q'IN CAPS
```

If the normal or shift field is filled out for a scancode that represents a function key, a self-explanatory message will be produced and that translation information will be ignored.

A more detailed example of a **scancodes** section is:

```
scancodes:
# the w key
0x11 'w'IC 'W'IC 'w'IN 'W'IN CAPS
# left square bracket and curly brace key
# control shift [ does not generate anything (no C flag)
0x1a '['IC '[' '['IN '['IN
# 9 on numeric keypad
0x49 'V'[ '9' '9'IN '9'IN NUM
F13 'd"a"t"e"0 # SHIFT F1
```

More complete examples of *mapfiles* can be found in `/usr/lib/keyboard/usa.map` and `/usr/lib/keyboard/*.map`.

**FILES**

<code>/usr/lib/keyboard/usa.map</code>	sample <i>mapfile</i> for using compose character sequences and deadkeys on a U.S. keyboard
<code>/usr/lib/keyboard/*.map</code>	sample <i>mapfiles</i> for European keyboards without compose and deadkey sections
<code>/usr/lib/keyboard/keys</code>	dump of default keytable for PC keyboard
<code>/usr/lib/keyboard/strings</code>	dump of default stringtable for PC keyboard

**SEE ALSO**

`stty(1)`, `keyboard(7)`, `termio(7)` in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.



**NAME**

catclose – close a message catalogue descriptor

**SYNOPSIS**

```
#include <nL_types.h>
```

```
int catclose (catd)  
nL_catd catd;
```

**DESCRIPTION**

The *catclose* function closes the message catalogue identified by *catd*. The file descriptor underlying the message catalogue descriptor will be closed.

**RETURN VALUE**

Upon successful completion, a value of 0 is returned.

**ERRORS**

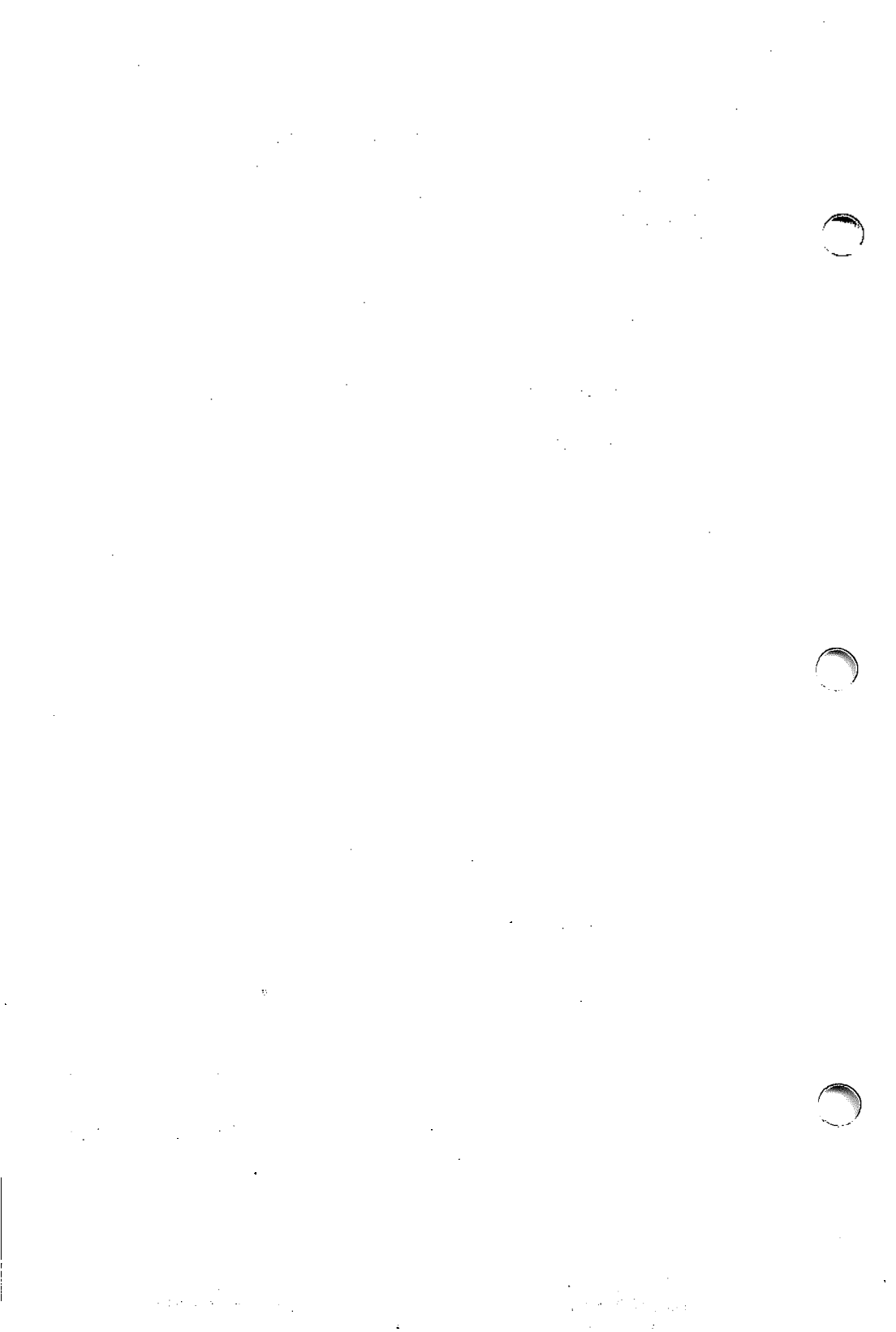
No errors are defined.

**SEE ALSO**

catopen(3P).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.



**NAME**

catgets – read a program message

**SYNOPSIS**

```
#include <nl_types.h>
```

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

**DESCRIPTION**

The *catgets* function attempts to read message *msg\_id*, in set *set\_id*, from the message catalogue identified by *catd*. The *catd* argument is a message catalogue descriptor returned from an earlier call to *catopen*(3P). The *s* argument points to a default message string that will be returned by *catgets* if it cannot retrieve the identified message.

**RETURN VALUES**

If the identified message is retrieved successfully, *catgets* returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful for any reason, *s* is returned.

**ERRORS**

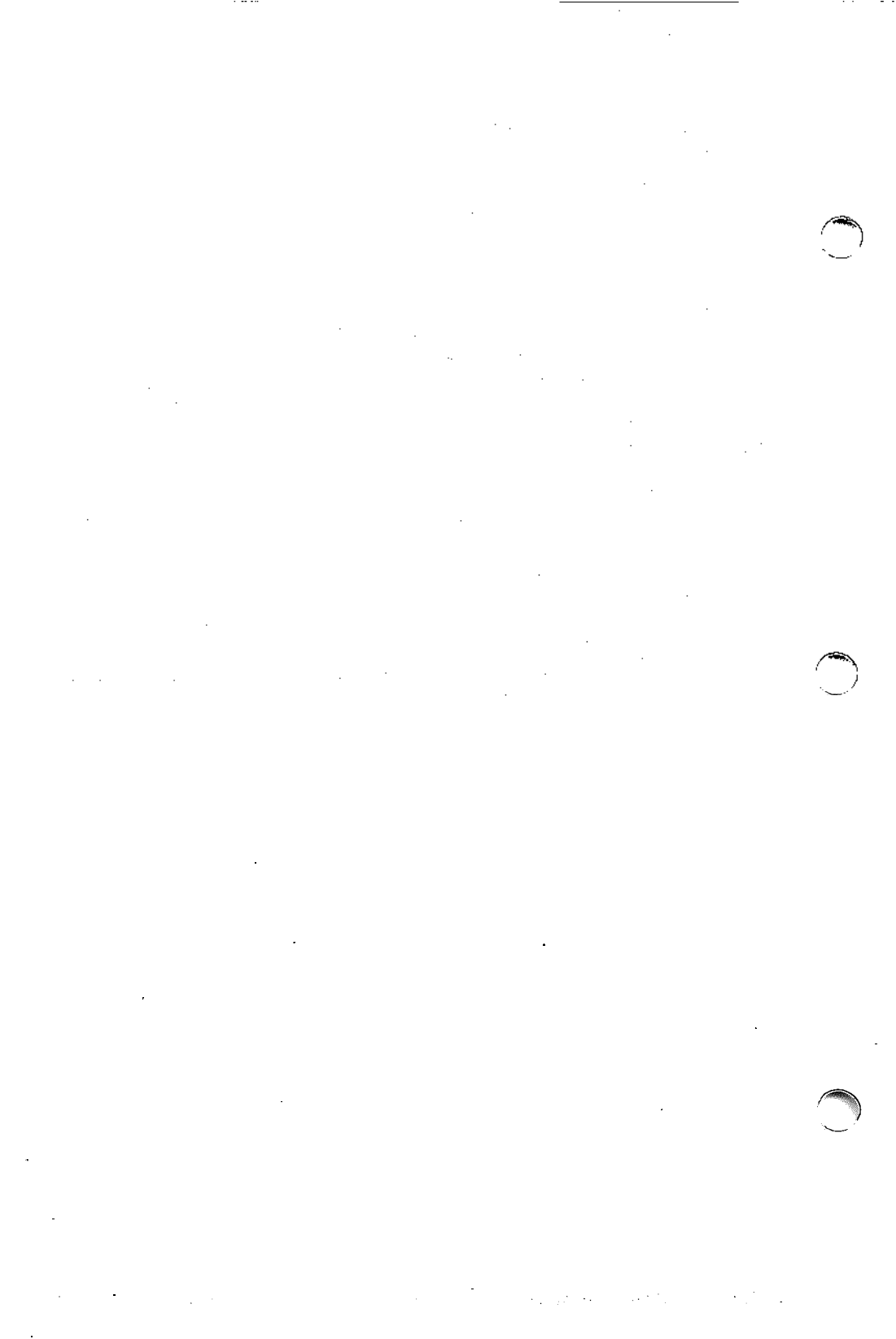
No errors are defined.

**SEE ALSO**

*catopen*(3P).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.



**NAME**

catopen – open a message catalogue

**SYNOPSIS**

```
#include <nl_types.h>

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

**DESCRIPTION**

The *catopen* function opens a message catalogue and returns a message catalogue descriptor. The *name* argument specifies the name of the message catalogue to be opened. If *name* contains a slash (/), then *name* specifies a complete name for the message catalogue. Otherwise, the environment variable NLSPATH is used with *name* substituted for %N (see *environ*(5P) for the description of NLSPATH from the *X/Open Portability Guide, Volume 2, XSI System Interface and Headers*). If NLSPATH does not exist in the environment, or if a message catalogue cannot be opened in any of the components specified by NLSPATH, then the default used by this implementation is */lib/locale/ISC/msgcat/name*.

In this implementation, *catopen* makes the following interpretations with respect to the processing of NLSPATH:

1. If the result from evaluating a %c, a %l, or a %t substitution field in NLSPATH exceeds NL\_LANGMAX characters (see the file */usr/include/limits.h*), it will be truncated to NL\_LANGMAX characters.
2. The result from evaluating a template in NLSPATH must not exceed PATH\_MAX characters (see */usr/include/limits.h*).
3. A % in NLSPATH not followed by a defined keyword or another % will be ignored.

The FD\_CLOEXEC flag will be set for the file descriptor underlying the message catalogue descriptor.

The *oflag* argument is reserved for future use and should be set to 0 (zero). The results of setting this field to any other value are undefined.

**RETURN VALUES**

Upon successful completion, *catopen* returns a message catalogue descriptor for use on subsequent calls to *catgets*(3P) and *catclose*(3P). Otherwise, *catopen* returns *(nl\_catd) -1* and sets *errno* to indicate the error, unless the message catalogue is corrupted, in which case *errno* may not be set.

**ERRORS**

In this implementation, *catopen* will fail if:

**[EINVAL]**

- 1) *name* contains a slash and exists but is not a message catalogue, or 2) *name* does not contain a slash, a message catalogue was not found using NLSPATH, and the system default, */lib/locale/ISC/msgcat/name*, exists but is not a message catalogue.

**[ENOMEM]**

Insufficient storage space is available (for internal buffer areas).

The following are possible failures from the underlying *fopen*(3) of the message catalogue:

**[EACCES]**

Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

**[EINTR]**

A signal was caught during the *fopen* function.

**[EMFILE]**

{FOPEN\_MAX} file descriptors, directories, and message catalogues are currently open in the calling process.

**[ENAMETOOLONG]**

The length of the *filename* string exceeds {PATH\_MAX}, or a path name component is longer than {NAME\_MAX} while {\_POSIX\_NO\_TRUNC} is in effect.

**[ENFILE]**

The system file table is full.

**[ENOENT]**

The named file does not exist, or the *filename* argument points to an empty string.

**[ENOTDIR]**

A component of the path prefix is not a directory.

**[ENXIO]**

The named file is a character special or block special file, and the device associated with this special file does not exist.

**SEE ALSO**

catclose(3P), catgets(3P).

environ(5P) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**NAME**

localeconv – numeric formatting convention inquiry

**SYNOPSIS**

```
#include <locale.h>
struct lconv *localeconv(void);
```

**DESCRIPTION**

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

The members of the structure with type **char \*** are pointers to strings, any of which (except *decimal\_point*) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are non-negative numbers, any of which can be **CHAR\_MAX** to indicate that the value is not available in the current locale. The members include the following:

**char \*decimal\_point**

The decimal-point character used to format non-monetary quantities.

**char \*thousands\_sep**

The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

**char \*grouping**

A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.

**char \*int\_curr\_symbol**

The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

**char \*currency\_symbol**

The local currency symbol applicable to the current locale.

**char \*mon\_decimal\_point**

The decimal-point used to format monetary quantities.

**char \*mon\_thousands\_sep**

The separator for groups of digits before the decimal-point in formatted monetary quantities.

**char \*mon\_grouping**

A string whose elements indicate the size of each group of digits in formatted monetary quantities.

**char \*positive\_sign**

The string used to indicate a nonnegative-valued formatted monetary quantity.

**char** *\*negative\_sign*

The string used to indicate a negative-valued formatted monetary quantity.

**char** *int\_frac\_digits*

The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

**char** *frac\_digits*

The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

**char** *p\_cs\_precedes*

Set to 1 or 0 if the *currency\_symbol* respectively precedes or succeeds the value for a non-negative formatted monetary quantity.

**char** *p\_sep\_by\_space*

Set to 1 or 0 if the *currency\_symbol* respectively is or is not separated by a space from the value for a non-negative formatted monetary quantity.

**char** *n\_cs\_precedes*

Set to 1 or 0 if the *currency\_symbol* respectively precedes or succeeds the value for a negative formatted monetary quantity.

**char** *n\_sep\_by\_space*

Set to 1 or 0 if the *currency\_symbol* respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

**char** *p\_sign\_posn*

Set to a value indicating the positioning of the *positive\_sign* for a non-negative formatted monetary quantity.

**char** *n\_sign\_posn*

Set to a value indicating the positioning of the *negative\_sign* for a negative formatted monetary quantity.

The elements of *grouping* and *mon\_grouping* are interpreted according to the following:

**CHAR\_MAX**

No further grouping is to be performed.

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

*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.

The value of *p\_sign\_posn* and *n\_sign\_posn* is interpreted according to the following:



- 0 Parentheses surround the quantity and *currency\_symbol*.
- 1 The sign string precedes the quantity and *currency\_symbol*.
- 2 The sign string succeeds the quantity and *currency\_symbol*.
- 3 The sign string immediately precedes the *currency\_symbol*.
- 4 The sign string immediately succeeds the *currency\_symbol*.

**RETURN VALUES**

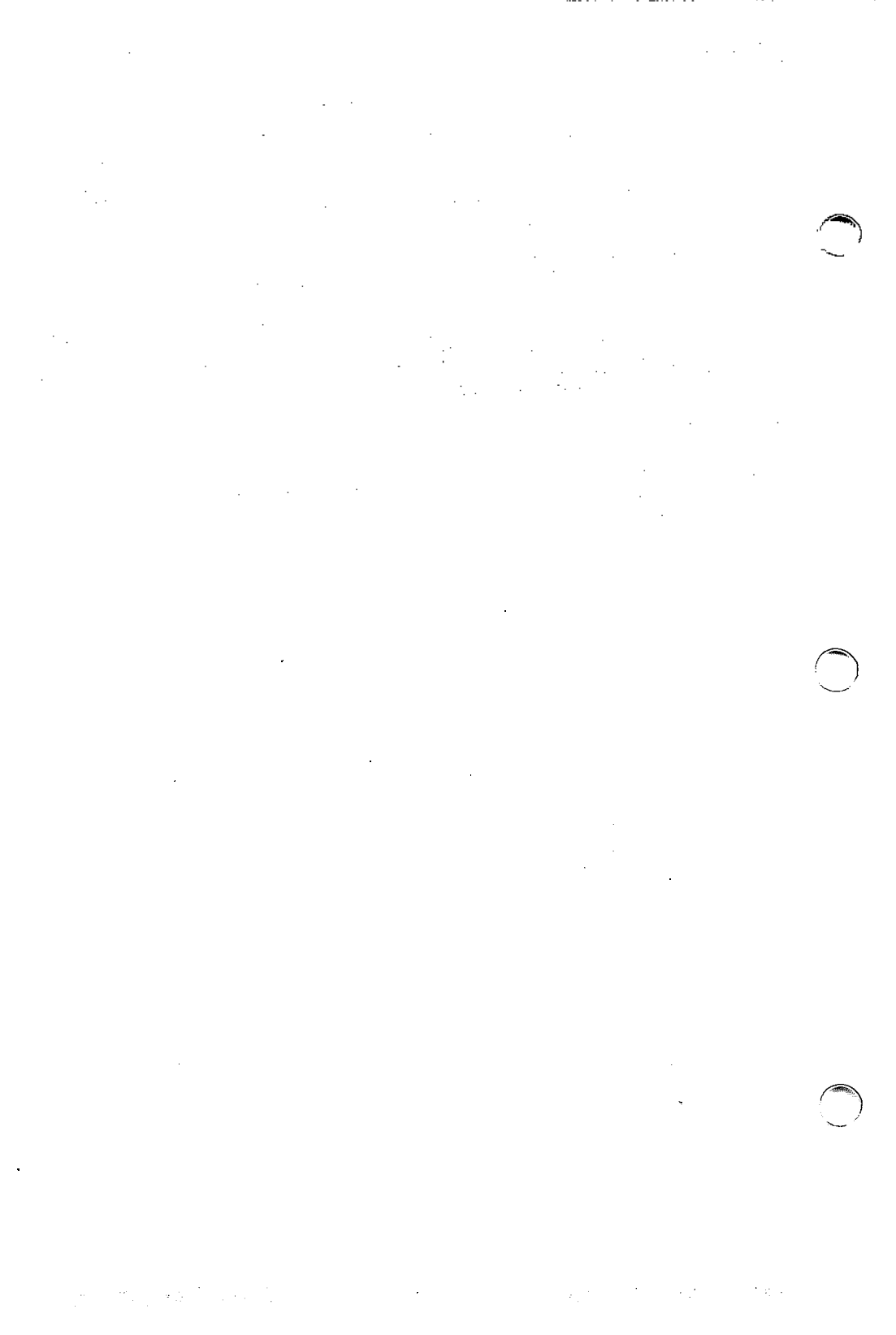
The *localeconv* function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the *localeconv* function. In addition, calls to the *setlocale* function with categories **LC\_ALL**, **LC\_MONETARY**, or **LC\_NUMERIC** may overwrite the contents of the structure.

**SEE ALSO**

*locale*(5P).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.



**NAME**

nl\_langinfo – language information

**SYNOPSIS**

```
#include <nl_types.h>
```

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

```
char *nl_langinfo (item)
```

```
nl_item item;
```

**DESCRIPTION**

The *nl\_langinfo* function returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale. The manifest constant names and values of *item* are defined in the file `/usr/include/langinfo.h`. For example:

```
nl_langinfo (ABDAY_1)
```

would return a pointer to the string *Dom* if the identified language was Portuguese, and *Sun* if the identified language was English.

The array pointed to by the return value should not be modified by the program, but may be modified by further calls to *nl\_langinfo*. In addition, calls to the *setlocale*(3P) function with a category corresponding to the category of *item* or to the category LC\_ALL may overwrite the array.

**RETURN VALUES**

In a locale where *langinfo* data is not defined, *nl\_langinfo* returns a pointer to the corresponding string in the C locale. In all locales, *nl\_langinfo* returns a pointer to an empty string if *item* contains an invalid setting.

**ERRORS**

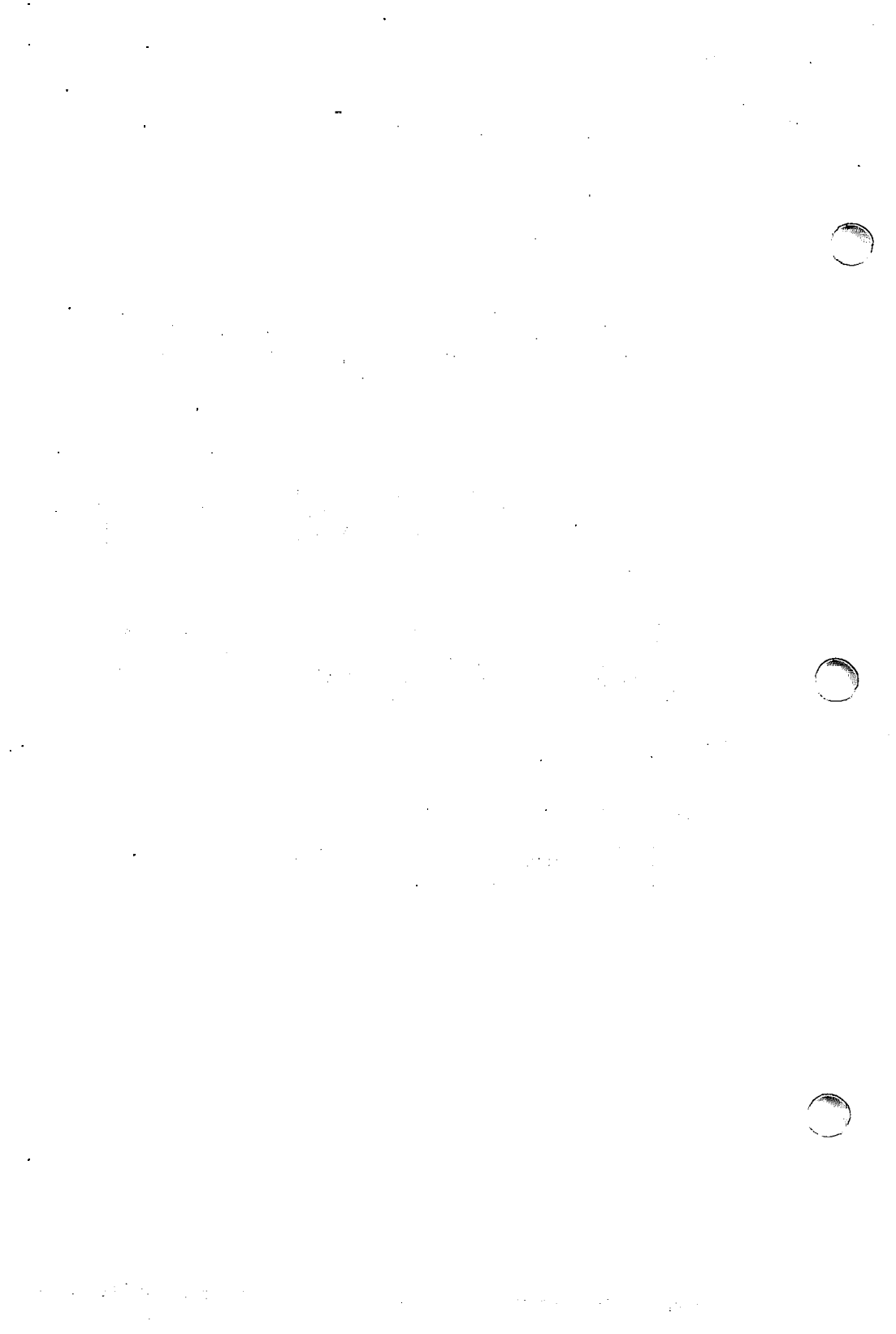
No errors are defined.

**SEE ALSO**

setlocale(3P), langinfo(5P), locale(5P).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.



## NAME

setlocale – locale control

## SYNOPSIS

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

## DESCRIPTION

The *setlocale* function sets, changes, or queries the program's locale according to the values of the *category* and *locale* arguments. The possible values for *category* are:

<b>LC_ALL</b>	Names the entire <i>locale</i> .
<b>LC_COLLATE</b>	Affects the behaviour of the string collation functions.
<b>LC_CTYPE</b>	Affects the behaviour of the character handling functions. The functions <i>isdigit</i> and <i>isxdigit</i> are not affected by the current locale.
<b>LC_MESSAGES</b>	Affects the interpretation of the strings associated with affirmative ( <i>y</i> ) and negative ( <i>n</i> ) responses.
<b>LC_MONETARY</b>	Affects the monetary formatting information returned by the <i>localeconv</i> function.
<b>LC_NUMERIC</b>	Affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the non-monetary formatting information returned by the <i>localeconv</i> function.
<b>LC_TIME</b>	Affects the behaviour of the <i>strftime</i> function.

The value **LC\_ALL** for *category* names all of the categories of the program's locale; **LC\_ALL** is a special constant, not a category.

The *locale* argument is a pointer to a character string that can be an explicit string, a NULL pointer, or a null string.

When *locale* is an explicit string, the contents of the string determines the locale. The values **POSIX** or **C** for *locale* are reserved for the default locale, which is the environment required for C translation, and also corresponds with the System V default behaviour. If *setlocale* is not invoked, the program's locale is the default locale.

When the *locale* is a NULL pointer, the program's locale is queried according to the value of *category*. The returned string contains the *locale* identifiers; if the category is **LC\_ALL**, the string contains semicolon-separated locale identifiers. Portable programs cannot rely on either the content or format of the returned string.

When the *locale* is a null string, the *setlocale* function takes the name of the new locale for the specified category from the environment as defined by the first condition met below:

1. If **LC\_ALL** is defined in the environment and is not null, the value of **LC\_ALL** is used.
2. If there is a variable defined in the environment with the same name as the category and that is not null, the value specified by that environment variable is used.

3. If LANG is defined in the environment and is not null, the value of LANG is used.

If the resulting value is a supported locale, *setlocale* sets the specified category of the program's locale to that value and returns the value specified below. If the value does not name a supported locale (and is not null), *setlocale* returns a NULL pointer and the program's locale is not changed by this function call. If no non-null environment variable is present to supply a value, *setlocale* sets the specified category of the program's locale to the default locale (see above).

Setting all of the categories of the program's locale is similar to successively setting each individual category of the program's locale, except that all error checking is done before any actions are performed. To set all categories of the program's locale, *setlocale* is invoked as:

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

In this case, *setlocale* first verifies that the values of all environment variables it needs according to the precedence above indicate supported locales. If the value of any of these environment variable searches yields a locale that is not supported (and non-null), the *setlocale* function returns a NULL pointer and the program's locale is not changed. If all environment variables name supported locales, *setlocale* then proceeds as if it had been called for each category, using the appropriate value from the associated environment variable or from the default locale if there is no such value.

## RETURN VALUES

A successful call to *setlocale* returns a string that corresponds to the locale set. The string is such that a subsequent call with that string and its associated category will restore that part of the program's locale. The string returned shall not be modified by the program, and may be overwritten by a subsequent call to the *setlocale* function.

## RESTRICTIONS

The LC\_ALL environment variable is an extension to the X/Open specification; it is derived from the 1990 C language standard.

The LC\_MESSAGES category (and environment variable) is also an extension to the X/Open specification; it is added in anticipation of the POSIX.2 standard.

Portable programs should avoid using or depending on these environment variables and on the LC\_MESSAGES category.

## NOTES

For information on how a locale is defined, see *locale(5P)*.

## SEE ALSO

*localeconv(3P)*.

## NOTE TO USERS

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**NAME**

strcoll – string comparison using collating information

**SYNOPSIS**

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

```
int strcoll (s1, s2)
```

```
char *s1, *s2;
```

**DESCRIPTION**

The *strcoll* function compares the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the LC\_COLLATE category of the current locale (see *locale(5P)*).

The sign of a nonzero value returned by *strcoll* is determined by the relative ordering within the current collating sequence of the first pair of characters that differ in the objects being compared.

**RETURN VALUE**

Upon successful completion, the *strcoll* function returns an integer greater than, equal to, or less than zero, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale. On error, *strcoll* sets *errno*, but no return value is reserved to indicate an error.

**ERRORS**

The *strcoll* function may fail if:

[EINVAL]

The *s1* or *s2* argument contains characters outside the domain of the collating sequence.

**NOTE**

The *strxfrm(3P)* and *strcmp* (see *string(3P)*) functions should be used for sorting large lists.

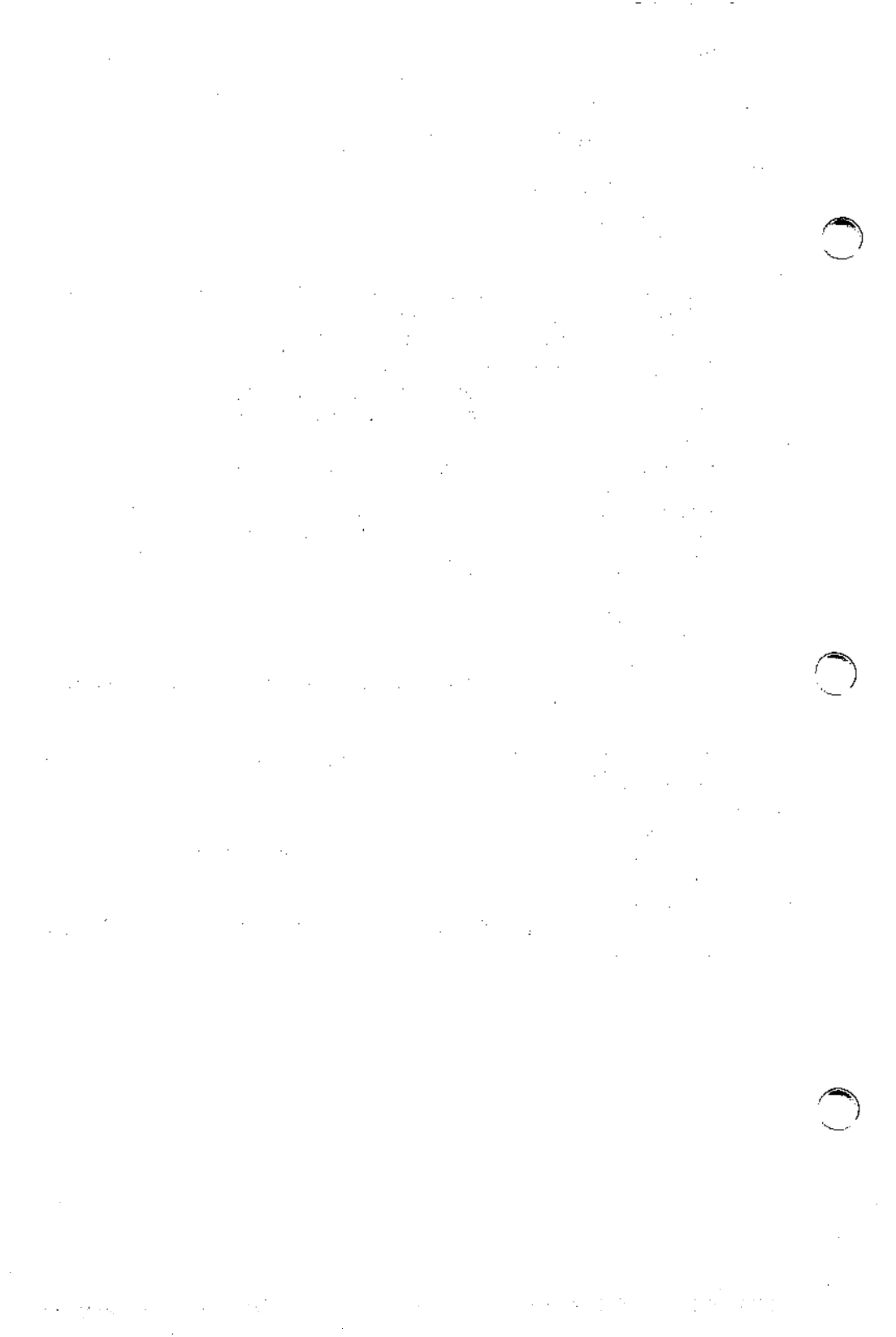
**SEE ALSO**

*strxfrm(3P)*.

*string(3P)* in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.





**NAME**

strerror – error message strings

**SYNOPSIS**

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

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

**DESCRIPTION**

The *strerror* function maps the error number in *errnum* to a language-dependent error message string and returns a pointer to it. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to the *strerror* function.

In this implementation, *strerror* obtains the error message strings from a message catalogue named **libc.cat**. If such a message catalogue is not found in NLS\_PATH (see *environ(5P)*), then the system default catalogue, **/lib/locale/ISC/msgcat/libc.cat**, which contains the English version of the error messages, will be used.

**RETURN VALUE**

Upon successful completion, *strerror* returns a pointer to the generated message string. No return value is reserved to indicate an error.

**ERRORS**

The *strerror* function may fail if:

[EINVAL]

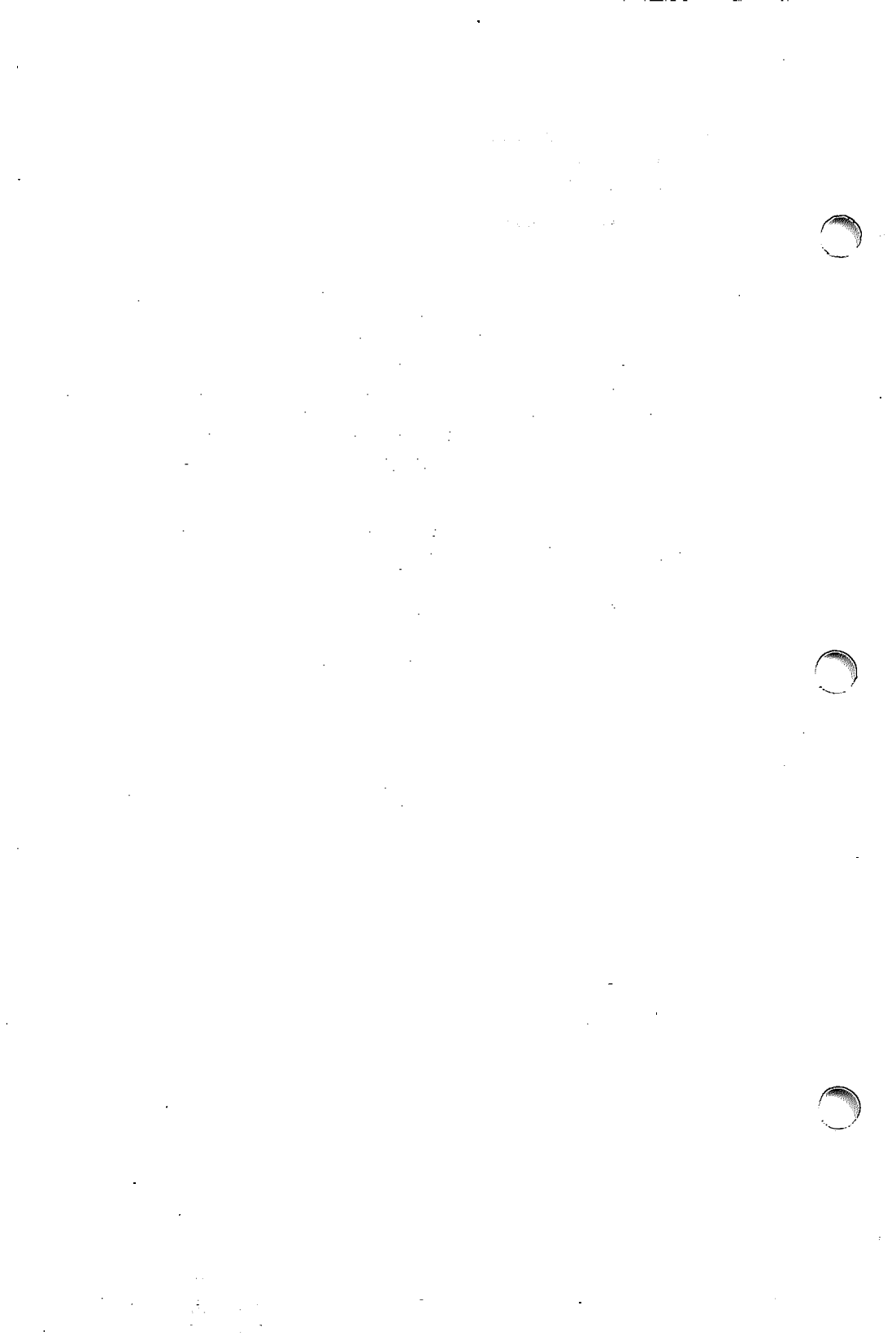
The value of *errnum* is not a valid error message number.

**SEE ALSO**

*perror(3P)*, *environ(5P)* in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.



**NAME**

strxfrm – string transformation

**SYNOPSIS**

#include &lt;string.h&gt;

size\_t strxfrm (s1, s2, n)

char \*s1, \*s2;

size\_t n;

**DESCRIPTION**

The *strxfrm* function transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the *strcmp* (see *string*(3P)) or *memcmp* (see *memory*(3C)) functions are applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the *strcoll*(3P) function applied to the same two original strings, based on the collating sequence information in the program's locale (category LC\_COLLATE); see *locale*(5P). No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behaviour is undefined.

**RETURN VALUE**

The *strxfrm* function returns the length of the transformed string (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

The *strxfrm* function returns (*size\_t*) -1 on error and sets *errno* to indicate the error.

**ERRORS**

The *strxfrm* function may fail if:

[EINVAL]

The *s1* or *s2* argument contains characters outside the domain of the collating sequence.

**SEE ALSO**

strcoll(3P), locale(5P).

memory(3C), string(3P) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.



**NAME**

**gencat** – format of message text source file used as input to **gencat(1P)**

**DESCRIPTION**

This entry supplies the format of a message text source file as defined by the *X/Open Portability Guide, Volume 3, XSI Supplementary Definitions*, Section 5.2.1, “Message Text Source Files.” The following symbolic constant values are found in `/usr/include/sys/limits.h` and `/usr/include/nl_types.h`, respectively:

<i>Symbolic Constant</i>	<i>Value</i>
NL_SETMAX	255
NL_MSGMAX	32767
NL_TEXTMAX	1023
NL_SETD	1

The format of a message text source file is defined as follows. Note that the fields of a message text source line are separated by a single ASCII space or tab character. Any other ASCII spaces or tabs are considered as being part of the subsequent field.

**\$set *n comment***

This line specifies the set identifier of the messages that follow until the next **\$set**, **\$delset**, or end-of-file appears. The *n* denotes the set identifier, which is defined as a number in the range [1, {NL\_SETMAX}]. Set identifiers must be presented in ascending order within a single source file but need not be contiguous. Any string following the set identifier is treated as a comment. If no **\$set** directive is specified in a message text source file, all messages will be located in an implementation-defined default message set NL\_SETD.

**\$delset *n comment***

This line deletes message set *n* from an existing message catalogue. The *n* denotes the set number [1, {NL\_SETMAX}]. Any string following the set number is treated as a comment.

**\$ *comment***

A line beginning with **\$** followed by an ASCII space or tab character is treated as a comment.

***m message-text***

The *m* denotes the message identifier, which is defined as a number in the range [1, {NL\_MSGMAX}]. The *message-text* is stored in the message catalogue with the set identifier specified by the last **\$set** directive, and with message identifier *m*. If the *message-text* is empty and an ASCII space or tab field separator is present, an empty string is stored in the message catalogue. If a message source line has a message number but neither a field separator nor *message-text*, the existing message with that number (if any) is deleted from the catalogue. Message identifiers must be in ascending order within a single set but need not be contiguous. The length of *message-text* must be in the range [0, {NL-TEXTMAX}].

**\$quote *c***

This specifies an optional quote character *c*, which can be used to surround *message-text* so that trailing spaces or null (empty) messages are visible in a message source line. By default, or if an empty **\$quote** directive is supplied, no quoting of *message-text* will be recognised.

Empty lines in a message text file are ignored. The effects of lines starting with any character other than those defined above are implementation defined.

Text strings can contain the special characters and escape sequences defined in the following table:

<i>Description</i>	<i>Symbol</i>	<i>Sequence</i>
new-line character	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 a 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.

A backslash followed by an ASCII new-line character is also used to continue a string on the following line. Thus, the following two lines describe a single message string:

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

which is equivalent to:

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

**SEE ALSO**

gencat(1P).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

**NAME**

loadfont – format of a loadfont input file

**DESCRIPTION**

This section describes the format of files that can be used to change the font used by the console when using the *loadfont* utility with the *-f* option.

The format is compatible with the Binary Distribution Format version 2.1 as developed by Adobe Systems, Inc., however, certain restrictions apply. Video cards, when used with the INTERACTIVE UNIX Operating System in textmode, only accept constant width, constant height fonts of certain sizes. The *loadfont* utility also requires that there is a description of all 256 characters of the codeset used specified in the fontfile. Certain attributes are not used by *loadfont* but are maintained for compatibility purposes.

As a consequence, fontfiles used with *loadfont* can also be used for other purposes, such as with the INTERACTIVE X11 Windowing System, but not always the other way around.

**File Format**

A *loadfont* input file is a plain ASCII file containing only printable characters (octal 40 through 176) and a carriage return at the end of each line.

The information about a particular font should be contained in a single file. The file begins with information on the font in general, followed by the information and bitmaps for the individual characters. The file should contain bitmaps for all 256 characters, and each character should be of the same size.

A font bitmap description file has the following general form, where each item is contained on a separate line of text in the file. Items on a line are separated by spaces:

The word **STARTFONT** followed by the version number 2.1.

One or more lines beginning with the word **COMMENT** These lines can be used to add comments to the file and will be ignored by the *loadfont* program.

The word **FONT** followed by the full name of the font. The name continues all the way to the end of the line, and may contain spaces.

The word **SIZE** followed by the point size of the characters, the x resolution, and the y resolution of the font. The sizes are not verified by *loadfont* but the line containing this keyword needs to be there for compatibility purposes.

The word **FONTBOUNDINGBOX** followed by the width in x, height in y, and the x and y displacement of the lower left-hand corner from the origin. Again, the sizes are not verified by *loadfont* but this line containing the keyword needs to be there for compatibility purposes.

Optionally, the word **STARTPROPERTIES** followed by the number of properties that follow. If present, the number needs to match the number of lines following this one before the

occurrence of a line beginning with **ENDPROPERTIES**. These lines consist of a word for the property name followed by either an integer or string surrounded by double quotes. Properties named **FONT\_ASCENT**, **FONT\_DESCENT** and **DEFAULT\_CHAR** are typically present in BDF files to define the logical font-ascent and font-descent and the default-char for the font.

As mentioned above, this section, if it exists, is terminated by **ENDPROPERTIES**.

The word **CHARS** followed by the number of characters that follow. This number should always be **256**.

This terminates the part of the *loadfont* input file describing features of the font in general. The rest of the file contains descriptions of the individual characters. They consist of the following parts:

The word **STARTCHAR** followed by up to 14 characters (no blanks) describing the character. This can either be something like **C0041**, which indicates the hex value of the character or **uppercaseA**, which describes the character.

The word **ENCODING** followed by a positive integer representing value by which this character is represented internally in the codeset for which this font is used. The integer needs to be specified in decimal.

The word **SWIDTH** followed by the scalable width in x and y of character. Scalable widths are in units of 1/1000th of the size of the character. The y value should always be 0; the x value is typically 666 for the type of characters used with *loadfont*. The values are not checked by the *loadfont* utility, but this line needs to be there for compatibility purposes.

The word **DWIDTH** followed by two numbers, which in a BDF file would mean the width in x and y of the character in device units. The y value is always zero. The x value is typically 8. *loadfont* checks only for the presence of the **DWIDTH** keyword.

The word **BBX** followed by the width in x, height in y and x and y displacement of the lower left-hand corner from the origin of the character.

Most fonts used by video cards will not use the bottom 4 rows of pixels, which basically means a vertical (y) displacement of -4. The only width allowed by *loadfont* is 8; heights supported are 8, 14, and 16. All **BBX** lines of the subsequent characters should list the same height and width as the first one (because only fixed size fonts are supported).

The optional word **ATTRIBUTES** followed by the attributes as 4 hex-encoded characters. The *loadfont* utility will accept this line, if present, but there is no meaning attached to it.

The word **BITMAP**, which indicates the beginning of the bit-map representation of the character. This line should be followed by **height** lines (height as specified in the **BBX** line)



representing a hex-encoded bitmap of the character, one byte per line.

The word **ENDCHAR** indicating the end of the bitmap for this character.

After all the bitmaps, the end of the file is indicated by the **ENDFONT** keyword.

### Example

The following example lists the beginning of the *loadfont* input file for an 8 by 16 font, supporting the IBM 437 codeset, as well as the bitmap representation of the character uppercase A.

```
STARTFONT 2.1
FONT 8x16
SIZE 16 75 75
FONTBOUNDINGBOX 8 16 0 -4
STARTPROPERTIES 3
FONT_DESCENT 4
FONT_ASCENT 12
DEFAULT_CHAR 0
ENDPROPERTIES
CHARS 256
STARTCHAR C0000
ENCODING 0
...
```

Bitmap for uppercase A character:

```
STARTCHAR C0041
ENCODING 65
SWIDTH 666 0
DWIDTH 8 0
BBX 8 16 0 -4
BITMAP
00
00
10
38
6c
c6
c6
fe
c6
c6
c6
c6
00
00
00
00
ENDCHAR
```

**loadfont(4)**

**loadfont(4)**

**FILES**

/usr/lib/loadfont/vga437.bdf

**SEE ALSO**

loadfont(1).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

## NAME

charmap – character set description file

**/lib/charmap/\***

## DESCRIPTION

The INTERACTIVE UNIX System supports single-byte coded character sets that are supersets of the ASCII coded character set. Examples of such coded character sets are:

**IBM codepage 437** This is the familiar “IBM PC” codeset, which is the default codeset in the INTERACTIVE UNIX System.

**IBM codepage 850** This is the IBM “International” codepage.

**ISO/IEC 8859-1** This is an international standard coded character set, also known as “Latin Alphabet No. 1,” which covers Western European languages.

Note that the 7-bit ASCII codeset must be contained within each of these codesets.

The *charmap* files are used to define and document the supported coded character sets, primarily for use in the *colldf*(1P) and *iconv*(1P) utilities. Each character in the coded character set is described with a symbolic name and the character encoding. The INTERACTIVE UNIX System provides charmap files for the above coded character sets, as well as a charmap for ASCII. Users may add charmap files provided that the following rules are followed:

1. The new charmap must contain the symbolic names and values used in the ASCII charmap.
2. The charmap can only contain entries describing single-byte characters between the CHARMAP and END CHARMAP statements.

The default location for charmap files used by *colldf* and *iconv* is **/lib/charmap**; a charmap file in any other directory must be specified by a path name containing a slash (/).

The format of a charmap file is as follows:

declarations

**CHARMAP** This is the charmap header.

**regular\_entries** These are the regular single-byte coded character set descriptions.

**END CHARMAP** Defines the end of the charmap.

**EXTENDED\_CHARMAP**

Starts optional section defining sequences of one or more bytes to be treated as characters by the *iconv* command.

**extended\_entries** These are the extended charmap entries.

**END EXTENDED\_CHARMAP**

Defines the end of the extended charmap section.

The following is a description of the permissible entries in each section and their format.

## DECLARATIONS

The following optional declarations can precede the character definitions. Each declaration consists of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more spaces or tabs, followed by the value to be assigned to the symbol.

### <code\_set\_name>

The name of the coded character set for which the character set description file is defined. Only characters defined in the ASCII charmap can be used in the name.

### <escape\_char>

The escape character is used to indicate that the characters following will be interpreted in a special way, as defined later. The default is the backslash (\) character.

### <comment\_char>

The comment character is used to indicate that the characters following on the line constitute a comment and will be ignored. The default is the # character.

### <mb\_cur\_max>

The maximum number of bytes in a character in the regular charmap. The default value (which is the only value permitted in the INTERACTIVE UNIX System) is 1.

### <mb\_cur\_min>

The minimum number of bytes in a character in the regular charmap. The value cannot exceed the value of <mb\_cur\_max>.

## CHARMAP

The charmap starts with an identifier line containing the string CHARMAP starting in column 1, and ends with a trailer line containing the string END CHARMAP starting in column 1. Empty lines and lines containing a # in the first column are ignored. Each noncomment line of the character set mapping definition (i.e., between the CHARMAP and END CHARMAP lines of the file) is in the form:

<symbolic-name> encoding

A symbolic name is one or more characters from the set defined in the ASCII charmap enclosed between angle brackets. A character following an escape character is interpreted as itself; for example, the sequence "<\\|>" represents the symbolic name "<\\>" enclosed between angle brackets.

The encoding part must be expressed as a decimal, octal, or hexadecimal constant in the following formats (the "<\\>" represents the escape character):

<code>\dnnn</code>	decimal value
<code>\xnn</code>	hexadecimal value
<code>\nnn</code>	octal value

Decimal constants are represented by two or three decimal digits, preceded by the escape character and the lowercase letter `d`; for example, `\d97` or `\d143`. Hexadecimal constants are represented by two hexadecimal digits, preceded by the escape character and the lowercase letter `x`; for example, `\x61` or `\x8f`. Octal constants are represented by two or three octal digits preceded by an escape character, for example, `\141` or `\217`.

Example of (part of) a *charmap* file:

```
CHARMAP
<NUL>          \d000
<newline>     \12
<percent-sign> \x25
<one>         \d048
<A>           \d065
<A-acute>    \d193
...
END CHARMAP
```

#### EXTENDED\_CHARMAP

The INTERACTIVE UNIX System does not support multi-byte coded character sets. However, certain common codesets (such as ISO 6937) define certain accented letters as combinations of two bytes (“dead key sequences”). As an example, the letter `<A-acute>` may be represented by a two-byte sequence, the first byte representing the accent and the second the base letter. The *iconv* utility requires that such characters be defined in the charmap. They must be defined in the optional EXTENDED\_CHARMAP section.

The format is the same as in the charmap section, except that the encoding consists of two (or more) concatenated constants, for example:

```
EXTENDED CHARMAP
<A-acute>      \d039\d065
END EXTENDED_CHARMAP
```

#### NOTES

“8859” is used as a synonym for the ISO/IEC 8859-1 codeset.

## FILES

User-defined *charmap* files must be stored in the **/lib/charmap** directory.

**/lib/charmap/\***

Default directory for charmap files (\* is the name of charmap file).

**/lib/charmap/ASCII.cmap**

Contains ASCII charmap entries.

**/lib/charmap/437.cmap**

Contains IBM codepage 437 charmap entries.

**/lib/charmap/850.cmap**

Contains IBM codepage 850 charmap entries.

**/lib/charmap\*/8859.cmap**

Contains ISO/IEC 8859-1 charmap entries.

## SEE ALSO

**colldef(1P), iconv(1P).**

**NAME**

langinfo – language information

**DESCRIPTION**

The **langinfo.h** header file defines the symbolic constants to be used in the *nl\_langinfo* function to retrieve *langinfo* data. The mode of the constants is given in **nl\_types.h**.

The following symbolic constants are recognized:

<b>D_T_FMT</b>	String for formatting date and time.
<b>D_FMT</b>	String for formatting of date.
<b>T_FMT</b>	String for formatting of time.
<b>AM_STR</b>	Ante Meridiem abbreviation.
<b>PM_STR</b>	Post Meridiem abbreviation.
<b>DAY_1</b>	Name of the first day of the week (e.g., Sunday).
<b>DAY_2</b>	Name of the second day of the week (e.g., Monday).
<b>DAY_3</b>	Name of the third day of the week (e.g., Tuesday).
<b>DAY_4</b>	Name of the fourth day of the week (e.g., Wednesday).
<b>DAY_5</b>	Name of the fifth day of the week (e.g., Thursday).
<b>DAY_6</b>	Name of the sixth day of the week (e.g., Friday).
<b>DAY_7</b>	Name of the seventh day of the week (e.g., Saturday).
<b>ABDAY_1</b>	Abbreviated name of the first day of the week.
<b>ABDAY_2</b>	Abbreviated name of the second day of the week.
<b>ABDAY_3</b>	Abbreviated name of the third day of the week.
<b>ABDAY_4</b>	Abbreviated name of the fourth day of the week.
<b>ABDAY_5</b>	Abbreviated name of the fifth day of the week.
<b>ABDAY_6</b>	Abbreviated name of the sixth day of the week.
<b>ABDAY_7</b>	Abbreviated name of the seventh day of the week.
<b>MON_1</b>	Name of the first month of the year (e.g., January).
<b>MON_2</b>	Name of the second month of the year (e.g., February).
<b>MON_3</b>	Name of the third month of the year (e.g., March).
<b>MON_4</b>	Name of the fourth month of the year (e.g., April).
<b>MON_5</b>	Name of the fifth month of the year (e.g., May).
<b>MON_6</b>	Name of the sixth month of the year (e.g., June).
<b>MON_7</b>	Name of the seventh month of the year (e.g., July).
<b>MON_8</b>	Name of the eighth month of the year (e.g., August).
<b>MON_9</b>	Name of the ninth month of the year (e.g., September).

MON_10	Name of the tenth month of the year (e.g., October).
MON_11	Name of the eleventh month of the year (e.g., November).
MON_12	Name of the twelfth month of the year (e.g., December).
ABMON_1	Abbreviated name of the first month of the year.
ABMON_2	Abbreviated name of the second month of the year.
ABMON_3	Abbreviated name of the third month of the year.
ABMON_4	Abbreviated name of the fourth month of the year.
ABMON_5	Abbreviated name of the fifth month of the year.
ABMON_6	Abbreviated name of the sixth month of the year.
ABMON_7	Abbreviated name of the seventh month of the year.
ABMON_8	Abbreviated name of the eighth month of the year.
ABMON_9	Abbreviated name of the ninth month of the year.
ABMON_10	Abbreviated name of the tenth month of the year.
ABMON_11	Abbreviated name of the eleventh month of the year.
ABMON_12	Abbreviated name of the twelfth month of the year.
RADIXCHAR	Decimal delimiter.
THOUSEP	Thousands separator.
YESSTR	Affirmative response for yes/no. Note that this is returned as an uncompiled regular expression.
NOSTR	Negative response for yes/no. Note that this is returned as an uncompiled regular expression.
CRNCYSTR	Currency symbol, preceded by “-” if the symbol should appear before the value, by “+” if the symbol should follow the value, or by “.” if the symbol should replace the decimal delimiter.

**SEE ALSO**

nl\_langinfo(3P).

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.



## NAME

locale – define and set international environment

## DESCRIPTION

A *locale* is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behaviour of components of the system. Category names correspond to the following environment variable names:

LC_ALL	Overrides the settings of all of the following environment variables.
LC_COLLATE	Affects the behaviour of the string collation functions.
LC_CTYPE	Affects the behaviour of the character handling functions.
LC_MESSAGES	Affects the interpretation of the strings associated with affirmative ( <i>y</i> ) and negative ( <i>n</i> ) responses.
LC_MONETARY	Affects the monetary formatting information returned by the <i>localeconv</i> (3P) function.
LC_NUMERIC	Affects the decimal-delimiter character for the formatted input/output functions and the string conversion functions, as well as the non-monetary formatting information returned by the <i>localeconv</i> function.
LC_TIME	Affects the behaviour of the <i>strftime</i> function (see <i>ctime</i> (3P)).
LANG	Provides a “fallback” value to be used if one of the above (except LC_ALL) is not set or is set to the empty string.

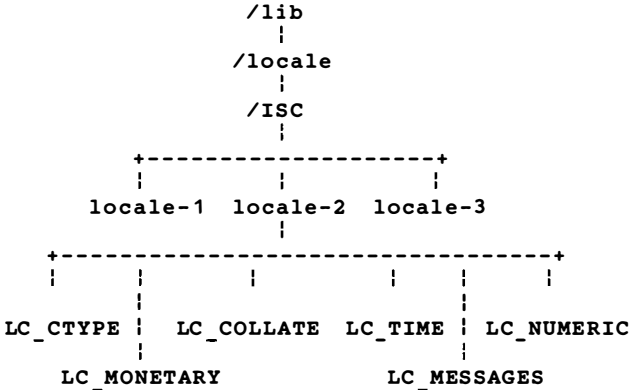
Programs compiled and linked with the *-Xp* option can use the *setlocale* function to modify the environment. When the program starts, the environment is set to the *C locale*, which corresponds to the traditional UNIX System environment. Programs can modify this environment by using the *setlocale*(3P) function.

If so directed by the program, the values of the above environment variables will be used to set the environment.

The value assigned to the environment variable LC\_ALL, if set, will be used for all locale categories. LC\_ALL is primarily intended for use when a user wishes to make sure that a particular program is executed with one locale only (i.e., no mixed locales).

The value assigned to the environment variable LANG will be used as the value for any of the above variables for which no valid value is assigned. If LANG is set to a valid value, and none of the above variables are set, then the entire environment will be set to the value indicated by LANG.

The information that defines a specific *locale* must be stored in data files on the system. The information for each category is stored in a file with a name corresponding to the environment variable name. The default location is within a directory under `/lib/locale/ISC`. The name of the directory is the name of the *locale*:



### Creating a Locale

The following steps are used to create the locale information.

Locales installed under `/lib/locale/ISC` should be viewed as “public” locales; all others should be considered private. Installation procedures are the same for both private and public locales. Only the system administrator should be able to create, modify, or delete public locales.

As a first step, create a directory with the desired name of the locale within `/lib/locale/ISC` (or, in case of a private locale, the appropriate directory). Then, the individual categories should be created as described in the following sections.

#### LC\_COLLATE

The information in the `LC_COLLATE` file is generated via the `colldf` utility. For details, see the utility description.

#### LC\_CTYPE

The information in the `LC_CTYPE` file is generated via the `chrtbl` utility. After executing the `chrtbl` utility, the generated data file must be copied or moved to the `locale` directory and given the name of `LC_CTYPE`. As an example, assuming that the name of the desired locale is `fr_FR.8859` and the `chrclass` value in the character classification table is `french`, then the following steps should be performed:

```

Schrtbl      sourcename
Sp french  /lib/locale/ISC/fr_FR.8859/LC_CTYPE
  
```

**LC\_MESSAGES**

The information in the **LC\_MESSAGES** file is in text format and defines the strings associated with the affirmative (“y”) and negative (“n”) responses used by selected utilities. Each line in the text file contains a keyword and a value, separated by space(s) or tab(s). Strings must be enclosed in quotation marks; individual characters can be so enclosed, but it is not required. Lines starting with a # are ignored. The following keywords are recognised:

**LC\_MESSAGES** This keyword must be the first in the file.

**yesexpr** The value is a regular expression used to evaluate an affirmative response. The regular expression must be enclosed in quote marks.

**noexpr** The value is a regular expression used to evaluate a negative response. The regular expression must be enclosed in quote marks.

**END LC\_MESSAGES**

This keyword must be the last in the file.

Example:

```
LC_MESSAGES
yesexpr      "[Yy][[:alpha:]]*"
noexpr       "[Nn].*"
END LC_MESSAGES
```

**LC\_MONETARY**

The information in the **LC\_MONETARY** file is in text format. Each line in the text file contains a keyword and a value, separated by space(s) or tab(s). Strings must be enclosed in quotation marks; individual characters can be so enclosed, but it is not required. Lines starting with a # are ignored. For a detailed definition of the values, see *localeconv(3P)*. The following keywords are recognised:

**LC\_MONETARY** This keyword must be the first in the file.

**int\_curr\_symbol** The value is the four-character string to be used as international currency symbol, enclosed in quote marks.

**currency\_symbol** The value is the character used as currency symbol.

**mon\_decimal\_point** The value is the decimal delimiter used to format monetary values.

**mon\_thousands\_sep** The value is the separator used to format monetary values.

**mon\_grouping** The value is a string of semicolon-separated numbers, as described in *localeconv(3P)*.

**positive\_sign** The string used to indicate a value for a non-negative formatted monetary quantity.

**negative\_sign** The string used to indicate a negative-valued formatted monetary quantity.

<b>int_frac_digits</b>	The number of fractional digits (those after the decimal delimiter) to be displayed in an internationally formatted monetary quantity.
<b>frac_digits</b>	The number of fractional digits (those after the decimal delimiter) to be displayed in a formatted monetary quantity.
<b>p_cs_precedes</b>	Set to 1 or 0 if the <b>currency_symbol</b> respectively precedes or succeeds the value for a non-negative formatted monetary quantity.
<b>p_sep_by_space</b>	Set to 1 or 0 if the <b>currency_symbol</b> respectively is or is not separated by a space from the value for a non-negative formatted monetary quantity.
<b>n_cs_precedes</b>	Set to 1 or 0 if the <b>currency_symbol</b> respectively is or is not separated by a space from the value for a negative formatted monetary quantity.
<b>n_sep_by_space</b>	Set to 1 or 0 if the <b>currency_symbol</b> respectively is or is not separated by a space from the value for a negative formatted monetary quantity.
<b>p_sign_posn</b>	Set to a value indicating the positioning of the <b>positive_sign</b> for a non-negative formatted monetary quantity.
<b>n_sign_posn</b>	Set to a value indicating the positioning of the <b>negative_sign</b> for a negative formatted monetary quantity.

**END LC\_MONETARY**

This keyword must be the last in the file.

## Example:

```
LC_MONETARY
int_curr_symbol      "USD "
currency_symbol      "$"
mon_decimal_point    "."
mon_thousands_sep   " "
mon_grouping         3
negative_sign        "CR"
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
n_sign_posn          1
END LC_MONETARY
```

**LC\_NUMERIC**

The information in the **LC\_NUMERIC** file is in text format. Each line in the text file contains a keyword and a value, separated by space(s) or tab(s). Lines starting with a **#** are ignored. The following keywords are recognized:

**LC\_NUMERIC** This keyword must be the first in the file.

- decimal\_point** The value is the character to be used as decimal delimiter; it may be enclosed in quotation marks.
- thousands\_sep** The value is the character used as the thousands separator; it may be enclosed in quotation marks.
- grouping** The value is a string of semicolon-separated numbers, as described in *localeconv(3P)*.
- END LC\_NUMERIC** This keyword must be the last in the file.

Example:

```
LC_NUMERIC
decimal_point    "."
thousands_sep   " "
grouping         3;3;0
END LC_NUMERIC
```

### LC\_TIME

The information in the **LC\_TIME** file is in text format. Each line in the text file contains a keyword and one or more values. The keyword is separated from the values by space(s) or tab(s). Values are separated by semicolons which can have spaces or tabs before or after them. Strings must be enclosed in quotation marks; individual characters can be so enclosed, but it is not required. Lines starting with a **#** are ignored. Lines can be continued by using a backslash (\) at the end of the line. The following keywords are recognised:

- LC\_TIME** This keyword must be the first in the file.
- abday** Defines the abbreviated names of the weekdays, starting with Sunday.
- day** Defines the names of the weekdays, starting with Sunday.
- abmon** Defines the abbreviated names of the months, starting with January.
- mon** Defines the names of the months, starting with January.
- t\_fmt** Defines the format of the time string, using the *strftime* conversion specifiers (see *ctime(3P)*).
- d\_fmt** Defines the format of the date string, using the *strftime* conversion specifiers (see *ctime(3P)*).
- d\_t\_fmt** Defines the format of the combined date and time string, using the *strftime* conversion specifiers (see *ctime(3P)*).
- am\_pm** Defines the strings used to represent ante meridiem and post meridiem (in that order).
- t\_fmt\_ampm** Defines the format of the time string in 12-hour format.
- END LC\_TIME** This keyword must be the last in the file.

## Example:

```

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"

t_fmt      "%H:%M:%S"
d_fmt      "%d/%m/%y"
d_t_fmt    "%a %b %d %H %M %S %Y"
am_pm      "AM"; "PM"
t_fmt_ampm "%I:%M:%S %p"
END LC_TIME

```

## Locale Naming Conventions and Usage

X/Open recommends that *locale* names follow a certain convention. The recommended format is:

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

where:

- language** Indicates the language area, e.g., fr (for French).
- territory** Indicates the geographical area, e.g., CH (for Switzerland), which controls, for example, monetary editing rules.
- codeset** Indicates the used code set, e.g., 8859.
- modifier** Can be used to distinguish between otherwise identical names (for instance between two different collation sequences).

## Example:

```

$LANG=fr_FR.8859
$LANG_COLLATE=$HOME/mylocale

```

In the above declarations, the default *locale* is French (France), using the 8859-1 codeset. (8859 is used as a synonym for the ISO/IEC 8859-1 codeset, also known as "Latin-1.") This is the *locale* chosen for all categories except `LC_COLLATE`, for which a "private" *locale* in the directory *mylocale* is chosen.

**FILES**

/lib/locale/ISC/\*      Default directory for locale directory structures (\* is the name of the locale).  
/lib/locale/ISC/\*/LC\_COLLATE      Contains LC\_COLLATE information.  
/lib/locale/ISC/\*/LC\_CTYPE      Contains LC\_CTYPE information.  
/lib/locale/ISC/\*/LC\_MESSAGES      Contains LC\_MESSAGES information.  
/lib/locale/ISC/\*/LC\_MONETARY      Contains LC\_MONETARY information.  
/lib/locale/ISC/\*/LC\_NUMERIC      Contains LC\_NUMERIC information.  
/lib/locale/ISC/\*/LC\_TIME      Contains LC\_TIME information.

**SEE ALSO**

chrtbl(1M), colldef(1P), localeconv(3P), setlocale(3P),  
ctime(3P), environ(5P) in the *INTERACTIVE SDS Guide and  
Programmer's Reference Manual*.

**NOTE TO USERS**

This entry is reprinted from the *INTERACTIVE SDS Guide and  
Programmer's Reference Manual*.

