Chapter 1

Introduction

1.1 Overview

This specification provides the common definitions for its companion specifications, CAE Specification, **Commands and Utilities, Issue 5** and CAE Specification, **System Interfaces and Headers**, **Issue 5** (see **Referenced Documents** on page xiv). It defines general terms, concepts and interfaces used by both other volumes. Thus, this volume is a prerequisite for understanding either of the other two.

1.2 Terminology

The following terms are used in this specification:

can
This describes a permissible optional feature or behaviour available to the user or application; all systems support such features or behaviour as mandatory requirements.

implementation-dependent
The value or behaviour is not consistent across all implementations. The provider of an implementation normally documents the requirements for correct program construction and correct data in the use of that value or behaviour. When the value or behaviour in the implementation is designed to be variable or customisable on each instantiation of the system, the provider of the implementation normally documents the nature and permissible ranges of this variation. Applications that are intended to be portable must not rely on implementation-dependent values or behaviour.

legacy
Certain features are *legacy*, which means that they are being retained for compatibility with older applications, but have limitations which make them inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality. Legacy features are marked **LEGACY**.

may
With respect to implementations, the feature or behaviour is optional. Applications should not rely on the existence of the feature. To avoid ambiguity, the reverse sense of *may* is expressed as *need not*, instead of *may not*.

must
This describes a requirement on the application or user.

should
With respect to implementations, the feature is recommended, but it is not mandatory. Applications should not rely on the existence of the feature.

With respect to users or applications, the word means recommended programming practice that is necessary for maximum portability.

undefined
A value or behaviour is undefined if this document imposes no portability requirements on applications for erroneous program constructs or erroneous data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application using such behaviour is not fully portable to all systems.

unspecified
A value or behaviour is unspecified if this document imposes no portability requirements on applications for correct program construct or correct data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application requiring a specific behaviour, rather than tolerating any behaviour when using that functionality, is not fully portable to all systems.

will
This means that the behaviour described is a requirement on the implementation and applications can rely on its existence.

1.3 Portability

Some of the utilities in CAE Specification, **Commands and Utilities, Issue 5** and functions in CAE Specification, **System Interfaces and Headers**, **Issue 5** describe functionality that might not be fully portable to systems based on the ISO POSIX-1 or ISO POSIX-2 standards. Where enhanced or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the extension or warning (see **Codes**). For maximum portability, an application should avoid such functionality.

Unless the primary task of a utility is to produce textual material on its standard output, application developers should not rely on the format or content of any such material that may be produced. Where the primary task *is* to provide such material, but the output format is incompletely specified, the description is marked. Application developers are warned not to expect that the output of such an interface on one system will be any guide to its behaviour on another system.

Codes

The codes and their meanings are as follows:

EX Extension.
The functionality described is an extension to the standards referenced above. Application writers may confidently make use of an extension as it will be supported on all XSI-conformant systems. These extensions are designed not to conflict with the published standards.

If an entire **SYNOPSIS** section is shaded and marked with one EX, all the functionality described in that entry is an extension.

Some behaviour which is allowed to be optional in the formal standards is mandated on XSI-conformant systems. Such behaviours (for example, those dependent on the availability of job control) might not be individually marked as extensions, but the mandatory nature of the feature is marked as an extension where the option is described, typically in the header where the corresponding symbolic constant is defined.

FIPS FIPS Requirements.
The **Federal Information Processing Standards (FIPS)** are a series of U.S. government procurement standards managed and maintained on behalf of the U.S. Department of Commerce by the National Institute of Standards and Technology (NIST). Where restrictions have been made in order to align with the FIPS requirements, they have the special mark shown here, and appear in the index under FIPS alignment (as well as under EX).

The following restrictions are required by FIPS 151-2:

- The implementation will support {_POSIX_CHOWN_RESTRICTED}.
- The limit {NGROUPS_MAX} will be greater than or equal to 8.
- The implementation will support the setting of the group ID of a file (when it is created) to that of the parent directory.
- The implementation will support {_POSIX_SAVED_IDS}.
- The implementation will support {_POSIX_VDISABLE}.
- The implementation will support {_POSIX_JOB_CONTROL}.
- The implementation will support {_POSIX_NO_TRUNC}.
- The *read*( ) call returns the number of bytes read when interrupted by a signal and will not return −1.
- The *write*( ) call returns the number of bytes written when interrupted by a signal and will not return −1.
- In the environment for the login shell, the environment variables *LOGNAME* and *HOME* will be defined and have the properties described in Chapter 5 of this document.
- The value of {CHILD_MAX} will be greater than or equal to 25.
- The value of {OPEN_MAX} will be greater than or equal to 20.
- The implementation will support the functionality associated with the symbols CS7, CS8, CSTOPB, PARODD and PARENB defined in **<termios.h>**.

JC Job Control Extension.
Job control is an optional feature in the operating system described by the ISO POSIX-1 standard, but it is supported by all XSI-conformant systems. When interfaces rely on this extension, they have the special mark shown here and appear in the index under JC (in addition to being under EX).

OB Obsolescent.
Some of the interfaces describe functionality that is obsolescent. Although these are fully portable to all current XSI-conformant systems they may be withdrawn in future issues.

OF Output format incompletely specified.
The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

OH Optional header.
In the **SYNOPSIS** section of some interfaces in CAE Specification, **System Interfaces and Headers**, **Issue 5** an included header is marked as in the following example:

OH
```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
```

This indicates that the marked header is not required on XSI-conformant systems. This is an extension to certain formal standards where the full synopsis is required.

OP Dependent on optional service in XSI.
Typical implementations depend on an optional service and the functionality affected need not be present if the optional service is not supported.

PI The behaviour cannot be guaranteed to be consistent.
It is not possible to guarantee that the interface behaves in the same way on all XSI-conformant systems. This is the case if it provides functionality that is system-defined or system-specific. Options that are used to *select* alternative forms of system-specific behaviour are not marked, as it is clear from their descriptions that their use is inherently non-portable.

RT Realtime.
This identifies the interfaces and additional semantics in the Realtime Feature Group.

RTT Realtime Threads.
This identifies the interfaces and additional semantics in the Realtime Threads Feature Group.

UN Possibly unsupportable feature.
It need not be possible to implement the required functionality (as defined) on all XSI-conformant systems and the functionality need not be present. This may, for example, be the case where the XSI-conformant system is hosted and the underlying system provides the service in an alternative way.

Chapter 2

Glossary

absolute pathname
See **pathname resolution** on page 22.

access mode
A particular form of access permitted to a file.

additional file access control mechanism
See **file access permissions** on page 14.

address space
The memory locations that can be referenced by a process or the threads of a process.

affirmative response
An input string that matches one of the responses acceptable to the LC_MESSAGES category keyword **yesexpr**, matching an extended regular expression in the current locale; see Section 5.3.6 on page 80.

alert
To cause the user’s terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case).

alert character
A character that in the output stream should cause a terminal to alert its user via a visual or audible notification. The alert character is the character designated by ’\a’ in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function.

alias name
A word consisting solely of underscores, digits and alphabetics from the portable character set (see Section 4.1 on page 43) and any of the following characters:

```
! % , @
```

Implementations may allow other characters within alias names as an extension.

alternate file access control mechanism
See **file access permissions** on page 14.

alternate signal stack
EX Memory associated with a thread, established upon request by the implementation for a thread, separate from the thread signal stack, in which signal handlers responding to signals sent to that thread may be executed.

angle brackets
The characters "<" (left-angle-bracket) and ">" (right-angle-bracket). When used in the phrase ‘‘enclosed in angle brackets’’, the symbol "<" immediately precedes the object to be enclosed, and ">" immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used.

appropriate privileges
An implementation-dependent means of associating privileges with a process with regard to the function calls and function call options defined in the **XSH** specification, and the commands in the **XCU** specification, that need special privileges. There may be zero or more such means.

argument
In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. See Section 10.1 on page 133 and the **XCU** specification, **Command Search and Execution** in **Section 2.9.1**. An argument is one of the options, option-arguments or operands following the command name.

In the C language, an expression in a function call expression or a sequence of preprocessing tokens in a function-like macro invocation.

arm (a timer)
To start a timer measuring the passage of time, enabling notifying a process when the specified time or time interval has passed.

assignment
See **variable assignment** on page 35.

asterisk
The character "*".

async-cancel safe function
A function that may be safely invoked by an application while the asynchronous form of cancellation is enabled. No function is async-cancel-safe unless explicitly described as such.

async-signal safe function
A function that may be invoked, without restriction, from signal-catching functions. No function is async-signal safe unless explicitly described as such.

asynchronously generated signal
A signal that is not attributable to a specific thread. Examples are: signals sent via *kill*( ), signals sent from the keyboard, and signals delivered to process groups. Being asynchronous is a property of how the signal was generated and not a property of the signal number. All signals may be generated asynchronously.

asynchronous I/O operation
An I/O operation that does not of itself cause the thread requesting the I/O to be blocked from further use of the processor.

This implies that the process and the I/O operation may be running concurrently.

asynchronous I/O completion
For an asynchronous read or write operation, when a corresponding synchronous read or write would have completed and when any associated status fields have been updated.

background job
See **background process group**.

background process
A process that is a member of a background process group.

background process group
(Or **background job**.) Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal.

backquote
The character `, also known as a *grave accent*.

backslash
The character "\", also known as a *reverse solidus*.

backspace character
A character that, in the output stream, should cause printing (or displaying) to occur one column position previous to the position about to be printed. If the position about to be printed is at the beginning of the current line, the behaviour is unspecified. The backspace is the character designated by '\b' in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the backspace function. The backspace character defined here is not necessarily the ERASE special character defined in Section 9.1.9 on page 123.

base character
One of the set of characters defined in the Latin alphabet. In Western European languages other than English, these characters are commonly used with diacritical marks (accents, cedilla, and so on) to extend the range of characters in an alphabet.

basename
The final, or only, filename in a pathname.

basic regular expression
A pattern constructed according to the rules defined in Section 7.3 on page 104.

blank character
One of the characters that belong to the **blank** character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character.

blank line
A line consisting solely of zero or more blank characters terminated by a newline character. See also **empty line** on page 12.

blocked process (or thread)
A process (or thread) that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution.

block-mode terminal
A terminal device operating in a mode incapable of the character-at-a-time input and output operations described by some of the standard utilities. See Section 8.2 on page 118.

block special file
A file that refers to a device. A block special file is normally distinguished from a character special file by providing access to the device in a manner such that the hardware characteristics of the device are not visible.

braces
The characters "{" (left brace) and "}" (right brace), also known as *curly braces*. When used in the phrase ‘‘enclosed in (curly) braces’’ the symbol "{" immediately precedes the object to be enclosed, and "}" immediately follows it. When describing these characters in the portable character set, the names <left-brace> and <right-brace> are used.

brackets
The characters "[" (left-bracket) and "]" (right-bracket), also known as *square brackets*. When used in the phrase ‘‘enclosed in (square) brackets’’ the symbol "[" immediately precedes the object to be enclosed, and "]" immediately follows it. When describing these characters in the portable

character set, the names <left-square-bracket> and <right-square-bracket> are used.

break value

EX The address at which dynamic memory allocation starts.

built-in utility

(Or **built-in**.) A utility implemented within a shell. The utilities referred to as *special built-ins* have special qualities, described in the **XCU** specification, **Section 2.14**, **Special Built-in Utilities**. Unless qualified, the term *built-in* includes the special built-in utilities. The utilities referred to as *regular built-ins* are those named in the **XCU** specification, **Command Search and Execution** in **Section 2.9.1**. There is no requirement that these utilities be actually built into the shell on the implementation, but they do have special command-search qualities.

byte

An individually addressable unit of data storage that is equal to or larger than an octet, used to store a character or a portion of a character; see **character**. A byte is composed of a contiguous sequence of bits, the number of which is implementation-dependent. The least significant bit is called the *low-order* bit; the most significant is called the *high-order* bit. Note that this definition of *byte* deviates intentionally from the usage of *byte* in some international standards, where it is used as a synonym for *octet* (always eight bits). On a system based on the ISO / IEC 9945-2: 1993 standard, a byte may be larger than eight bits so that it can be an integral portion of larger data objects that are not evenly divisible by eight bits (such as a 36-bit word that contains four 9-bit bytes).

carriage-return character

A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by ’\r’ in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line.

character

A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of a multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed.

See Section 4.1 on page 43 for a further explanation of the graphical representations of characters, or *glyphs*, as opposed to character encodings.

character array

An array of type **char**.

character class

A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale; see Section 5.3.1 on page 52.

character set

A finite set of different characters used for the representation, organisation or control of data.

character special file

A file that refers to a device. One specific type of character special file is a terminal device file, whose access is defined in Chapter 9 on page 119.

character string
A contiguous sequence of characters terminated by and including the first null byte.

child process
See **process** on page 25.

circumflex
The character "^".

clock
An object that measures the passage of time.

The current value of the time measured by a clock can be queried and, possibly, set to a value within the legal range of the clock.

clock tick
An interval of time; an implementation-dependent number of these occur each second.

coded character set
A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation.

codeset
The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information; see Section 4.2 on page 44. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned.

collating element
The smallest entity used to determine the logical ordering of character or wide-character strings. See **collation sequence**. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements.

collation
The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements.

collation sequence
The relative order of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

Multi-level sorting is accomplished by assigning elements one or more collation weights, up to the limit {COLL_WEIGHTS_MAX}; see **<limits.h>**. On each level, elements may be given the same weight (at the primary level, called an equivalence class; see **equivalence class** on page 13) or be omitted from the sequence. Strings that collate equal using the first assigned weight (primary ordering) are then compared using the next assigned weight (secondary ordering), and so on.

column position
A unit of horizontal measure related to characters in a line.

It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this specification set, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1.

command
A directive to the shell to perform a particular task; see the **XCU** specification, **Section 2.9**, **Shell Commands**.

command language interpreter
An interface that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. It is possible for applications to invoke utilities through a number of interfaces, which are collectively considered to act as command interpreters. The most obvious of these are the *sh* utility and the *system*( ) function, although *popen*( ) and the various forms of *exec* may also be considered to behave as interpreters.

composite graphic symbol
A graphic symbol consisting of a combination of two or more other graphic symbols in a single character position, such as a diacritical mark and a basic letter.

condition variable
A synchronization object which allows a thread to suspend execution, repeatedly, until some associated predicate becomes true.

control character
A character, other than a graphic character, that affects the recording, processing, transmission or interpretation of text.

control operator
In the shell, a token that performs a control function. It is one of the following symbols:

```
&    &&    (    )    ;    ;;    newline    |    ||
```

The end-of-input indicator used internally by the shell is also considered a control operator. See the **XCU** specification, **Section 2.3**, **Token Recognition**.

On some systems, the symbol **((** is a control operator; its use produces unspecified results. Applications that wish to have nested subshells, such as:

```
((echo Hello);(echo World))
```

must separate the **((** characters into two tokens by including white space between them. Some systems may treat these as invalid arithmetic expressions instead of subshells.

The **((** and **))** symbols are control operators in the KornShell, used for an alternative syntax of an arithmetic expression command. A portable application cannot use **((** as a single token (with the exception of the **$((** form for shell arithmetic).

controlling process
The session leader that established the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process.

controlling terminal
A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal (see Chapter 9 on page 119) cause signals to be sent to all processes in the process group associated with the controlling terminal.

conversion descriptor
EX A per-process unique value used to identify an open codeset conversion.

core file
EX A file of unspecified format that may be generated when a process terminates abnormally.

current working directory
See **working directory** on page 36.

cursor position
The line and column position on the screen denoted by the terminal's cursor.

data segment
EX Memory associated with a process, that may be used to contain dynamically allocated data.

device
A computer peripheral or an object that appears to the application as such.

device ID
A non-negative integer used to identify a device.

direct I/O
An operation that attempts to circumvent a system performance optimization for the optimization of the individual I/O operation.

directory
A file that contains directory entries. No two directory entries in the same directory have the same name.

directory entry
(Or **link**.) An object that associates a filename with a file. Several directory entries can associate names with the same file.

directory stream
A sequence of all the directory entries in a particular directory. An open directory stream may be implemented using a file descriptor.

disarm (a timer)
To stop a timer from measuring the passage of time, disabling any future process notifications (until the timer is armed again).

display
To output to the user's terminal. If the output is not directed to a terminal, the results are undefined.

The **XCU** specification assigns precise requirements for the terms *display* and *write*. Some historical systems have chosen to implement certain utilities without using the traditional UNIX system file descriptor model. For example, the *vi* editor might employ direct screen memory updates on a personal computer, rather than a *write*( ) system call. An instance of user prompting might appear in a dialogue box, rather than with standard error. When the **XCU** specification uses the term *display*, the method of outputting to the terminal is unspecified; many historical implementations use *termcap* or *terminfo*, but this is not a requirement. The term *write* is used when the **XCU** specification mandates that a file descriptor be used and that the output can be redirected. However, it is assumed that when the writing is directly to the terminal (it has not been redirected elsewhere), there is no practical way for a user or test suite to determine whether a file descriptor is being used or not. Therefore, the use of a file descriptor is mandated only for the redirection case and the implementation is free to use any method when the output is not redirected. The verb *write* is used almost exclusively, with the very few exceptions of those utilities where output redirection need not be supported: *tabs*, *talk*, *tput* and *vi*.

dollar sign
The character "$".

dot
The filename consisting of a single dot character (.). See **pathname resolution** on page 22. In the context of shell special built-in utilities, see *dot* in the **XCU** specification, **Section 2.14**, **Special Built-in Utilities**.

dot-dot
The filename consisting solely of two dot characters (..). See **pathname resolution** on page 22.

double-quote
The character """, also known as *quotation-mark*.

downshifting
The conversion of an upper-case character to its lower-case representation.

(clock) drift rate
The rate at which the time measured by a clock deviates from the actual passage of real time.

A positive drift rate causes a clock to gain time with respect to real time; a negative drift rate causes a clock to lose time with respect to real time.

driver
EX A module that controls data transferred to and received from peripheral devices. Drivers are traditionally written to be a part of the system implementation, although they are frequently written separately from the writing of the implementation. A driver may contain processor-specific code, and therefore be non-portable.

effective group ID
An attribute of a process that is used in determining various permissions, including file access permissions, described in **file access permissions** on page 14. See **group ID** on page 17. This value is subject to change during the process lifetime, as described in the *exec* family of functions and *setgid*( ).

effective user ID
An attribute of a process that is used in determining various permissions, including file access permissions. See **user ID** on page 35. This value is subject to change during the process lifetime, as described in *exec* and *setuid*( ).

eight-bit transparency
The ability of a software component to process 8-bit characters without modifying or utilising any part of the character in a way that is inconsistent with the rules of the current coded character set.

empty directory
A directory that contains, at most, directory entries for dot and dot-dot.

empty line
A line consisting of only a newline character. See also **blank line** on page 7.

empty string
(Or **null string**.) A string whose first byte is a null byte.

empty wide-character string
A wide-character string whose first element is a null wide-character code.

epoch
The time zero hours, zero minutes, zero seconds, on January 1, 1970 Coordinated Universal Time. See **seconds since the epoch** on page 28.

equivalence class
A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight.

era
An alternative method for counting and displaying years. See Section 5.3.5 on page 73.

executable file
A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file.

execute
To perform the actions described in the **XCU** specification, **Command Search and Execution** in **Section 2.9.1**. See also **invoke** on page 17.

expand
In the shell, when not qualified, the act of applying all the expansions described in the **XCU** specification, **Section 2.6**, **Word Expansions**.

extended regular expression
A pattern constructed according to the rules defined in Section 7.4 on page 109.

extended signed integral type
EX A signed integral type or an implementation-dependent type with similar properties.

extended security controls
The access control (see **file access permissions** on page 14) and privilege (see **appropriate privileges** on page 6) mechanisms have been defined to allow implementation-dependent extended security controls. These permit an implementation to provide security mechanisms to support different security policies from those described in this specification set. These mechanisms do not alter or override the defined semantics of any of the functions or utilities in this specification set.

extended unsigned integral type
EX An unsigned integral type or an implementation-dependent type with similar properties.

feature test macro
A macro used to determine whether a particular set of features will be included from a header. See the **XSH** specification, **Section 2.2**, **The Compilation Environment**.

field

In the shell, a unit of text that is the result of parameter expansion (see the **XCU** specification, **Section 2.6.2**, **Parameter Expansion**), arithmetic expansion (see the **XCU** specification, **Section 2.6.4**, **Arithmetic Expansion**), command substitution (see the **XCU** specification, **Section 2.6.3**, **Command Substitution**), or field splitting (see the **XCU** specification, **Section 2.6.5**, **Field Splitting**). During command processing (see the **XCU** specification, **Section 2.9.1**, **Simple Commands**), the resulting fields are used as the command name and its arguments.

FIFO special file

(Or **FIFO**.) A type of file with the property that data written to such a file is read on a first-in-first-out basis. Other characteristics of FIFOs are described in *open*( ), *read*( ), *write*( ) and *lseek*( ).

file

An object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file and directory. Other types of files may be supported by the implementation.

file access permissions

The standard file access control mechanism uses the file permission bits, as described below. These bits are set at the time of file creation by functions such as *open*( ), *creat*( ), *mkdir*( ) and *mkfifo*( ) and are changed by *chmod*( ). These bits are read by *stat*( ) or *fstat*( ).

Implementations may provide *additional* or *alternate* file access control mechanisms, or both. An additional access control mechanism will only further restrict the access permissions defined by the file permission bits. An alternate file access control mechanism will:

- specify file permission bits for the file owner class, file group class, and file other class of that file, corresponding to the access permissions, to be returned by *stat*( ) or *fstat*( )
- be enabled only by explicit user action, on a per-file basis by the file owner or a user with the appropriate privilege
- be disabled for a file after the file permission bits are changed for that file with *chmod*( ). The disabling of the alternate mechanism need not disable any additional mechanisms supported by an implementation.

Whenever a process requests file access permission for read, write or execute/search, if no additional mechanism denies access, access is determined as follows:

- If a process has the appropriate privilege:
  - If read, write or directory search permission is requested, access is granted.
  - If execute permission is requested, access is granted if execute permission is granted to at least one user by the file permission bits or by an alternate access control mechanism; otherwise, access is denied.
- Otherwise:
  - The file permission bits of a file contain read, write and execute/search permissions for the file owner class, file group class and file other class.
  - Access is granted if an alternate access control mechanism is not enabled and the requested access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs, or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access is denied.

file description
See **open file description** on page 21.

file descriptor
A per-process unique, non-negative integer used to identify an open file for the purpose of file access. The value of a file descriptor is from zero to {OPEN_MAX}. A process can have no more than {OPEN_MAX} file descriptors open simultaneously. File descriptors may also be used to implement message catalogue descriptors and directory streams. See **open file description** on page 21 and {OPEN_MAX} in **<limits.h>**. EX

file group class
The property of a file indicating access permissions for a process related to the group identification of a process. A process is in the file group class of a file if the process is not in the file owner class and if the effective group ID or one of the supplementary group IDs of the process matches the group ID associated with the file. Other members of the class may be implementation-dependent.

file hierarchy
Files in the system are organised in a hierarchical structure in which all of the non-terminal nodes are directories and all of the terminal nodes are any other type of file. Because multiple directory entries may refer to the same file, the hierarchy is properly described as a *directed graph*.

file mode
An object containing the *file mode bits* and file type of a file, as described in **<sys/stat.h>**.

file mode bits
A file's file permission bits, set-user-ID-on-execution bit (S_ISUID) and set-group-ID-on-execution bit (S_ISGID); see **<sys/stat.h>**.

filename
A name consisting of 1 to {NAME_MAX} bytes used to name a file. The characters composing the name may be selected from the set of all character values excluding the slash character and the null byte. The filenames dot and dot-dot have special meaning; see **pathname resolution** on page 22. A filename is sometimes referred to as a *pathname component*.

Filenames should be constructed from the portable filename character set because the use of other characters can be confusing or ambiguous in certain contexts. (For instance, the use of a colon (:) in a pathname could cause ambiguity if that pathname were included in a *PATH* definition.)

file offset
The byte position in the file where the next I/O operation begins. Each open file description associated with a regular file, block special file or directory has a file offset. A character special file that does not refer to a terminal device may have a file offset. There is no file offset specified for a pipe or FIFO.

file other class
The property of a file indicating access permissions for a process related to the user and group identification of a process. A process is in the file other class of a file if the process is not in the file owner class or file group class.

file owner class
The property of a file indicating access permissions for a process related to the user identification of a process. A process is in the file owner class of a file if the effective user ID of the process matches the user ID of the file.

file permission bits
Information about a file that is used, along with other information, to determine if a process has read, write or execute/search permission to a file. The bits are divided into three parts: owner, group and other. Each part is used with the corresponding file class of processes. These bits are contained in the file mode, as described in **<sys/stat.h>**. The detailed usage of the file permission bits in access decisions is described in **file access permissions** on page 14.

file serial number
A per-file-system unique identifier for a file.

file system
A collection of files and certain of their attributes. It provides a name space for file serial numbers referring to those files.

file times update
Each file has three associated time values that are updated when file data has been accessed, file data has been modified, or file status has been changed, respectively. These values are returned in the file characteristics structure, as described in **<sys/stat.h>**.

For each function or utility in this specification set that reads or writes file data or changes the file status, the appropriate time-related fields are noted as ‘‘marked for update’’. At an update point in time, any marked fields are set to the current time and the update marks cleared. Two such update points are when the file is no longer open by any process and when *stat*( ) or *fstat*( ) is performed on the file. Additional update points are unspecified. Marks for update, and updates themselves, are not done for files on read-only file systems.

file type
See **file** on page 14.

filter
A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream.

first open (of a file)
When a process opens a file that is not currently an open file within any process.

foreground job
See **foreground process group**.

foreground process
A process that is a member of a foreground process group.

foreground process group
(Or **foreground job**.) A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. See Chapter 9.

foreground process group ID
The process group ID of the foreground process group.

form-feed character
A character that in the output stream indicates that printing should start on the next page of an output device. The form-feed is the character designated by '\f' in the C language. If the form-feed is not the first character of an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page.

graphic character
A character, other than a control character, that has a visual representation when handwritten, printed or displayed.

group database
A system database of implementation-dependent format that contains at least the following information for each group ID:

- Group Name
- Numerical Group ID
- List of users allowed in the group.

The list of users allowed in the group is used by the *newgrp* utility.

group ID
A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group FIPS ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs or a saved set-group-ID.

group name
A string that is used to identify a group, as described in **group database**. To be portable across XSI-conformant systems, the value must be composed of characters from the portable filename character set. The hyphen should not be used as the first character of a portable group name.

hard limit
EX A system resource limitation that may be reset to a lesser or greater limit by a privileged process. A non-privileged process is restricted to only lowering its hard limit.

hard link
The relationship between two directory entries that represent the same file; see **directory entry** on page 11. This term is contrasted against **symbolic link**; see **symbolic link** on page 31.

home directory
The current directory associated with a user at the time of login.

incomplete line
A sequence of one or more non-newline characters at the end of the file.

Inf
A value representing infinity that can be stored in a floating type. Not all systems support the Inf value.

interactive shell
A processing mode of the shell that is suitable for direct user interaction.

internationalisation
The provision within a computer program of the capability of making itself adaptable to the requirements of different native languages, local customs and coded character sets.

invoke
To perform the actions described in the **XCU** specification, **Command Search and Execution** in **Section 2.9.1**, except that searching for shell functions and special built-in utilities is suppressed. See also **execute** on page 13.

ISO/IEC 646:1983
ISO 7-bit coded character set for information interchange. The reference version of the standard contains 95 graphic characters, which are identical to the graphic characters defined in the ASCII

coded character set.

ISO 6937: 1983
ISO 7-bit or 8-bit coded character set for text communication using public communication networks, private communication networks, or interchange media such as magnetic tapes and discs.

ISO 8859-1: 1987
ISO 8-bit single-byte coded character set Part 1, Latin Alphabet No 1. This standard character set comprises 191 graphic characters covering the requirements of most of Western Europe.

job
A set of processes, comprising a shell pipeline, and any processes descended from it, that are all in the same process group. See the definition of **pipeline** in the **XCU** specification, **Section 2.9.2**, **Pipelines**.

job control
A facility that allows users selectively to stop (suspend) the execution of processes and continue (resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter.

job control job ID
A handle that is used to refer to a job. The job control job ID can be any of the forms shown in the following table:

Job Control Job ID	Meaning
%%	Current job
%+	Current job
%–	Previous job
%*n*	Job number *n*
%*string*	Job whose command begins with *string*
%?*string*	Job whose command contains *string*

Table 2-1 Job Control Job ID Formats

last close (of a file)
When a process closes a file, resulting in the file not being an open file within any process.

line
A sequence of zero or more non-newline characters plus a terminating newline character.

link
See **directory entry** on page 11.

link count
The number of directory entries that refer to a particular file.

local customs
The conventions of a geographical area or territory for such things as date, time and currency formats.

locale
The definition of the subset of a user's environment that depends on language and cultural conventions; see Chapter 5 on page 49.

localisation
The process of establishing information within a computer system specific to the operation of particular native languages, local customs and coded character sets.

login
The unspecified activity by which a user gains access to the system. Each login is associated with exactly one login name.

login name
A user name that is associated with a login.

map
To create an association between a page-aligned range of the address-space of a process and a range of physical memory or some memory object, such that a reference to an address in that range of the address-space results in a reference to the associated physical memory or memory object. The mapped memory or memory object is not necessarily memory-resident.

marked message
EX A STREAMs message on which a certain flag is set. Marking a message gives the application protocol-specific information. An application can use *ioctl*( ) to determine whether a given message is marked.

memory object
RT Either a file or shared memory object.

When used in conjunction with *mmap*( ), a memory object will appear in the address-space of the calling process.

message
Information that can be transferred between processes or threads by being added to and removed from a message queue. A message consists of a fixed-size message buffer.

message catalogue
EX A file or storage area containing program messages, command prompts and responses to prompts for a particular native language, territory and codeset.

message catalogue descriptor
EX A per-process unique value used to identify an open message catalogue. A message catalogue descriptor may be implemented using a file descriptor.

message queue
An object to which messages can be added and removed. Messages may be removed in the order in which they were added or in priority order.

mode
A collection of attributes that specifies a file's type and its access permissions. See **file access permissions** on page 14.

mount point
Either the system root directory or a directory for which the *st_dev* field of structure **stat** (see **<sys/stat.h>**) differs from that of its parent directory.

multi-character collating element
A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements ch and ll.

mutex
A synchronization object used to allow multiple threads to serialize their access to shared data. The name derives from the capability it provides; namely, mutual exclusion. The thread that has locked a mutex becomes its owner and remains the owner until that same thread unlocks the mutex.

name
In the shell, a word consisting solely of underscores, digits and alphabetics from the portable character set (see Section 4.1 on page 43). The first character of a name must not be a digit.

There are no explicit limits in this specification set on the sizes of names, words (see **word** on page 36), lines or other objects. However, other implicit limits do apply: shell script lines produced by many of the standard utilities cannot exceed {LINE_MAX} and the sum of exported variables comes under the {ARG_MAX} limit. Historical shells dynamically allocate memory for names and words and parse incoming lines a byte at a time. Lines cannot have an arbitrary {LINE_MAX} limit because of historical practice such as makefiles, where *make* removes the newline characters associated with the commands for a target and presents the shell with one very long line. The text on **INPUT FILES** in the **XCU** specification, **Section 1.9**, **Utility Description Defaults** does allow a shell to run out of memory, but it cannot have arbitrary programming limits.

named STREAM
EX A STREAMS-based file descriptor that is attached to a name in the file-system namespace. All subsequent operations on the named STREAM act on the STREAM that was associated with the file descriptor until the name is disassociated from the STREAM.

NaN (not a number)
A value that can be stored in a floating type but that is not a valid floating point number. Not all systems support the NaN value.

native language
A computer user's spoken or written language, such as American English, British English, Danish, Dutch, French, German, Italian, Japanese, Norwegian or Swedish.

negative response
An input string that matches one of the responses acceptable to the LC_MESSAGES category keyword **noexpr**, matching an extended regular expression in the current locale. See Section 5.3.6 on page 80.

newline character
A character that in the output stream indicates that printing should start at the beginning of the next line. The newline is the character designated by '\n' in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line.

non-spacing characters
A character, such as a character representing a diacritical mark in the ISO 6937: 1983 standard coded character set, which is used in combination with other characters to form composite graphic symbols.

NUL
A character with all bits set to zero.

null byte
A byte with all bits set to zero.

null pointer
The value that is obtained by converting the number 0 into a pointer; for example, (**void** *) 0. The C language guarantees that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

null string
See **empty string** on page 13.

null wide-character code
A wide-character code with all bits set to zero.

number sign
The character #, also known as *hash sign*.

object file
A regular file containing the output of a compiler, formatted as input to a linkage editor for linking with other object files into an executable form. The methods of linking are unspecified and may involve the dynamic linking of objects at run time. The internal format of an object file is unspecified, but a conforming application cannot assume an object file is a text file.

offset maximum
EX An attribute of an open file description representing the largest value that can be used as a file offset.

open file
A file that is currently associated with a file descriptor.

open file description
A record of how a process or group of processes are accessing a file. Each file descriptor refers to exactly one open file description, but an open file description can be referred to by more than one file descriptor. A file offset, file status and file access modes are attributes of an open file description.

operand
An argument to a command that is generally used as an object supplying information to a utility necessary to complete its processing. Operands generally follow the options in a command line. See Section 10.1 on page 133.

operator
In the shell, either a control operator or a redirection operator.

option
An argument to a command that is generally used to specify changes in the utility's default behaviour; see Section 10.1 on page 133.

option-argument
A parameter that follows certain options. In some cases an option-argument is included within the same argument string as the option; in most cases it is the next argument. See Section 10.1 on page 133.

orphaned process group
A process group in which the parent of every member is either itself a member of the group or is not a member of the group's session.

page
The granularity of process memory mapping or locking.

Physical memory and memory objects can be mapped into the address-space of a process on page boundaries and in integral multiples of pages. Process address-space can be locked into memory (made memory-resident) on page boundaries and in integral multiples of pages.

page size
EX The size, in bytes, of the system unit of memory allocation, protection and mapping. On systems that have segment- rather than page-based memory architectures, the term ‘‘page’’ means a segment.

parameter
In the shell, an entity that stores values. There are three types of parameters: variables (named parameters), positional parameters and special parameters. Parameter expansion is accomplished by introducing a parameter with the "$" character. See the **XCU** specification, **Section 2.5**, **Parameters and Variables**.

In the C language, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition.

parent directory
When discussing a given directory, the directory that both contains a directory entry for the given directory and is represented by the pathname dot-dot in the given directory.

When discussing other types of files, a directory containing a directory entry for the file under discussion.

This concept does not apply to dot and dot-dot.

parent process
See **process** on page 25.

parent process ID
An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process.

pathname
A character string that is used to identify a file. A pathname consists of, at most, {PATH_MAX} bytes, including the terminating null byte. It has an optional beginning slash, followed by zero or more filenames separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A pathname that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes are treated as a single slash. The interpretation of the pathname is described in **pathname resolution**.

pathname component
See **filename** on page 15.

pathname resolution
Pathname resolution is performed for a process to resolve a pathname to a particular file in a file hierarchy. There may be multiple pathnames that resolve to the same file.

Each filename in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment **a/b**, file **b** is located in directory **a**). Pathname resolution fails if this cannot be accomplished. If the pathname begins with a slash, the predecessor of the first filename in the pathname is taken to be the root directory of the process (such pathnames are referred to as absolute pathnames). If the pathname does not begin with a slash, the predecessor of the first filename of the pathname is taken to be the current working directory of the process (such pathnames are referred to as relative pathnames).

FIPS

The interpretation of a pathname component is dependent on the values of {NAME_MAX} and {_POSIX_NO_TRUNC} associated with the path prefix of that component. If any pathname component is longer than {NAME_MAX}, because {_POSIX_NO_TRUNC} is in effect on all XSI-conformant systems for the path prefix of that component (see *pathconf*( )), the implementation will consider this an error condition.

EX If a symbolic link (see **symbolic link** on page 31) is encountered during pathname resolution, then pathname resolution is complete if all of the following are true:

- This is the last component of the pathname.
- The pathname has no trailing slash.
- The function is required to act on the symbolic link itself, or certain arguments direct that the function act on the symbolic link itself.

In all other cases, the system prefixes the remaining pathname, if any, with the contents of the symbolic link. The function may fail, setting *errno* to [ENAMETOOLONG], if the combined length exceeds {PATH_MAX}. Otherwise, the resolved pathname is the resolution of the pathname just created. The result is either an absolute pathname that is resolved from the root directory of the process or a relative pathname that is resolved from the directory containing the symbolic link.

The special filename dot refers to the directory specified by its predecessor. The special filename dot-dot refers to the parent directory of its predecessor directory. As a special case, in the root directory, dot-dot may refer to the root directory itself.

A pathname consisting of a single slash resolves to the root directory of the process. A null pathname is invalid.

path prefix
A pathname, with an optional ending slash, that refers to a directory.

pattern
A sequence of characters used either with regular expression notation (see Chapter 7 on page 101) or for pathname expansion (see the **XCU** specification, **Section 2.6.6**, **Pathname Expansion**), as a means of selecting various character strings or pathnames, respectively.

The syntaxes of the two patterns are similar, but not identical; this specification set always indicates the type of pattern being referred to in the immediate context of the use of the term.

period
The character (.). The term *period* is contrasted against **dot**, which is used to describe a specific directory entry.

permissions
See **file access permissions** on page 14.

persistence
A mode for semaphores, shared memory and message queues requiring that the object and its state (including data, if any) are preserved after the object is no longer referenced by any process.

Persistence of an object does not imply that the state of the object is maintained across a system crash or a system reboot.

pipe
An object accessed by one of the pair of file descriptors created by the *pipe*( ) function. Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy.

positional parameter
In the shell, a parameter denoted by a single digit or one or more digits in curly braces. See the **XCU** specification, **Section 2.5.1**, **Positional Parameters**.

portable character set
The collection of characters that are required to be present in all locales supported by XSI-conformant systems:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ! # % ˆ & * ( ) _ + - = { } [ ]
: " ˜ ; ’ ‘ < > ? , . | \ / @ $
```

Also included are the alert, backspace, tab, newline, vertical-tab, form-feed, carriage-return and space characters and the null character, NUL.

This term is contrasted against the smaller **portable filename character set**. See Table 4-1 on page 43.

portable filename character set
The set of characters from which portable filenames are constructed. For a filename to be portable across implementations conforming to this specification set and the ISO POSIX-1 standard, it must consist only of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore and hyphen characters, respectively. The hyphen must not be used as the first character of a portable filename. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable pathname, the slash character may also be used.

preallocation
The reservation of resources in a system for a particular use.

Preallocation does not imply that the resources are immediately allocated to that use, but merely indicates that they are guaranteed to be available in bounded time when needed.

preempted process (or thread)
A running thread whose execution is suspended due to another thread becoming runnable at a higher priority.

printable character
One of the characters included in the **print** character classification of the LC_CTYPE category in the current locale; see Section 5.3.1 on page 52.

printable file
A text file consisting only of the characters included in the **print** and **space** character classifications of the LC_CTYPE category and the backspace character, all in the current locale; see Section 5.3.1 on page 52.

priority
A non-negative integer associated with processes or threads whose value is constrained to a range defined by the applicable scheduling policy. Numerically higher values represent higher priorities.

priority band
EX The queueing order applied to normal priority STREAMS messages. High priority STREAMS messages are not grouped by priority bands. The only differentiation made by the STREAMS mechanism is between zero and non-zero bands, but specific protocol modules may differentiate between priority bands.

priority-based scheduling
Scheduling in which the selection of a running thread is determined by the priorities of the runnable threads.

privilege
See **appropriate privileges** on page 6.

process
An address space with one or more threads executing within that address space, and the required system resources for those threads.

Many of the system resources defined by this specification are shared among all of the threads within a process. These include: the process ID, the parent process ID, process group ID, session membership, real, effective and saved-set user ID, real, effective and saved-set group ID, supplementary group IDs, current working directory, root directory, file mode creation mask and file descriptors.

A process is created by another process issuing the *fork*( ) function. The process that issues *fork*( ) is known as the parent process, and the new process created by the *fork*( ) is known as the child process.

process group
A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group that is identified by a process group ID. A newly created process joins the process group of its creator.

process group ID
The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. A process group ID will not be reused by the system until the process group lifetime ends.

process group leader
A process whose process ID is the same as its process group ID.

process group lifetime
A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, due either to the end of the last process' lifetime or to the last remaining process calling the *setsid*( ) or *setpgid*( ) functions.

process ID
The unique identifier representing a process. A process ID is a positive integer. A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1.

process lifetime
The period of time that begins when a process is created and ends when its process ID is returned to the system. After a process is created with a *fork*( ) function, it is considered active. At least one thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources,
EX such as the process ID, are still in use. When another process executes a *wait*( ), *wait3*( ), *waitid*( ) or *waitpid*( ) function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

process list

EX See **thread list** on page 34.

process virtual time

EX The measurement of time in units elapsed by the system clock while a process is executing.

program

A prepared sequence of instructions to the system to accomplish a defined task. The term *program* in this specification set encompasses applications written in the XSI Shell Command Language, complex utility input languages (for example, *awk*, *lex*, *sed*, and so on), and high-level languages.

pseudo-terminal

EX A pseudo-terminal provides the process with an interface that is identical to the terminal subsystem. A pseudo-terminal is composed of 2 devices, the master device and a slave device. The slave device provides processes with an interface that is identical to the terminal interface, although there need not be hardware behind that interface. Anything written on the master device is presented to the slave as an input and anything written on the slave device is presented as an input on the master side.

This specification requires a STREAMS-based implementation of pseudo-terminals to be available, but does not preclude others also being available.

radix character

The character that separates the integer part of a number from the fractional part.

read-only file system

A file system that has implementation-dependent characteristics restricting modifications.

read-write lock

EX Multiple readers, single writer (read-write) locks allow many threads to have simultaneous read-only access to data while allowing only one thread to have write access at any given time. They are typically used to protect data that is read-only more frequently than it is changed.

Read-write locks can be used to synchronise threads in the current process and other processes if they are allocated in memory that is writable and shared among the cooperating processes and have been initialised for this behaviour.

real group ID

The attribute of a process that, at the time of process creation, identifies the group of the user who created the process. See **group ID** on page 17. This value is subject to change during the process lifetime, as described in *setgid*( ).

real time

EX Time measured as total units elapsed by the system clock without regard to which thread is executing.

real user ID

The attribute of a process that, at the time of process creation, identifies the user who created the process. See **user ID** on page 35. This value is subject to change during the process lifetime, as described in *setuid*( ).

redirection

In the shell, a method of associating files with the input or output of commands. See the **XCU** specification, **Section 2.7**, **Redirection**.

redirection operator
In the shell, a token that performs a redirection function. It is one of the following symbols:

```
<    >    >|    <<    >>    <&    >&    <<–    <>
```

reentrant function
A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved.

referenced shared memory object
A shared memory object that is open or has one or more mappings defined on it.

refresh
To ensure that the information on the user’s terminal screen is up-to-date.

regular expression
A pattern constructed according to the rules defined in Chapter 7 on page 101.

region
In the context of the address space of a process, a sequence of addresses.

In the context of a file, a sequence of offsets.

regular file
A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system.

relative pathname
See **pathname resolution** on page 22.

(time) resolution
The minimum time interval that a clock can measure or whose passage a timer can detect.

root directory
A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash.

runnable process (or thread)
A thread that is capable of being a running thread, but for which no processor is available.

running process (or thread)
A thread currently executing on a processor. On multi-processor systems there may be more than one such thread in a system at a time.

saved resource limits
EX An attribute of a process that provides some flexibility in the handling of unrepresentable resource limits, as described in the *exec* family of functions and *setrlimit*( ).

saved set-group-ID
An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the *exec* family of functions and *setgid*( ).

saved set-user-ID
An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in *exec* and *setuid*( ).

scheduling
The application of a policy to select a runnable process or thread to become a running process or thread, or to alter one or more of the thread lists.

scheduling allocation domain
The set of processors on which an individual thread can be scheduled at any given time.

scheduling contention scope
A property of a thread that defines the set of threads against which that thread competes for resources.

For example, in a scheduling decision, threads sharing scheduling contention scope compete for processor resources. In this specification, a thread has scheduling contention scope of either PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS.

scheduling policy
A set of rules that is used to determine the order of execution of threads to achieve some goal.

In the context of XSI, a scheduling policy affects thread ordering:

- when a thread is a running thread and it becomes a blocked thread
- when a thread is a running thread and it becomes a preempted thread
- when a thread is a blocked thread and it becomes a runnable thread
- when a running thread calls a function that can change the priority or scheduling policy of a thread
- in other scheduling policy-defined circumstances.

Conforming implementations are required to define the manner in which each of the scheduling policies may modify the priorities or otherwise affect the ordering of threads at each of the occurrences listed above. Additionally, conforming implementations will define at what other circumstances and in what manner each scheduling policy may modify the priorities or affect the ordering of threads.

screen
A rectangular region of columns and lines on a terminal display. A screen may be a portion of a physical display device or may occupy the entire physical area of the display device.

scroll
To move the representation of data vertically or horizontally relative to the terminal screen. There are two types of scrolling:

1. The cursor moves with the data.
2. The cursor remains stationary while the data moves.

seconds since the epoch
A value to be interpreted as the number of seconds between a specified time and the epoch. A Coordinated Universal Time name (specified in terms of seconds (*tm_sec*), minutes (*tm_min*), hours (*tm_hour*), days since January 1 of the year (*tm_yday*), and calendar year minus 1900 (*tm_year*)) is related to a time represented as seconds since the Epoch, according to the expression below.

If the year < 1970 or the value is negative, the relationship is undefined. If the year ≥ 1970 and the value is non-negative, the value is related to a Coordinated Universal Time name according to the expression:

$$tm\_sec + tm\_min*60 + tm\_hour*3\,600 + tm\_yday*86\,400 + \\ (tm\_year-70)*31\,536\,000 + ((tm\_year-69)\,/\,4)*86\,400$$

semaphore
A shareable resource that has a non-negative integral value. When the value is zero, there is a (possibly empty) set of threads awaiting the availability of the semaphore.

semaphore lock operation
An operation that is applied to a semaphore. If, prior to the operation, the value of the semaphore is zero, the semaphore lock operation causes the calling thread to be blocked and added to the set of threads awaiting the semaphore. Otherwise, the value is decremented.

semaphore unlock operation
An operation that is applied to a semaphore. If, prior to the operation, there are any threads in the set of threads awaiting the semaphore, then some thread from that set will be removed from the set and become unblocked. Otherwise, the semaphore value is incremented.

session
A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see *setsid*( ). There can be multiple process groups in the same session.

session leader
A process that has created a session; see *setsid*( ).

session lifetime
The period between when a session is created and the end of the lifetime of all the process groups that remain as members of the session.

shared memory object
An object that represents memory that can be mapped concurrently into the address space of more than one process.

shell
A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal.

shell, the
The XSI Shell Command Language Interpreter (see *sh*), a specific instance of a shell.

shell script
A file containing shell commands. If the file is made executable, it can be executed by specifying its name as a simple command (see the **XCU** specification, **Section 2.9.1**, **Simple Commands**). Execution of a shell script causes a shell to execute the commands within the script. Alternatively, a shell can be requested to execute the commands in a shell script by specifying the name of the shell script as the operand to the *sh* utility.

signal
A mechanism by which a process or thread may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself.

signal stack
EX Memory established for a thread, in which signal handlers catching signals sent to that thread are executed.

single-quote
The character "’", also known as *apostrophe*.

slash
The character "/", also known as *solidus*.

socket
EX A communications endpoint associated with a file descriptor that provides communications services using a specified communications protocol. See the **Networking Services** specification.

soft limit
EX A resource limitation established for each process that the process may set to any value less than or equal to the hard limit.

source code
When dealing with the XSI Shell Command Language, input to the command language interpreter. The term *shell script* is synonymous with this meaning.

When dealing with the C language, input to a C compiler conforming to the ISO C standard.

When dealing with another XSI-compliant language, input to a compiler conforming to that language standard.

Source code also refers to the input statements prepared for the following standard utilities: *awk*, *bc*, *ed*, *lex*, *localedef*, *make*, *sed* and *yacc*.

Source code can also refer to a collection of sources meeting any or all of these meanings.

special parameter
In the shell, a parameter named by a single character from the following list:

```
   *   @   #   ?   !   -   $   0
```

See the **XCU** specification, **Section 2.5.2**, **Special Parameters**.

space character
The character defined in the portable character set as <space>. The space character is a member of the **space** character class of the current locale, but represents the single character, and not all of the possible members of the class. (See **white space** on page 36.)

standard error
An output stream usually intended to be used for diagnostic messages.

standard input
An input stream usually intended to be used for primary data input.

standard output
An output stream usually intended to be used for primary data output.

standard utilities
The utilities described in the **XCU** specification.

stream
Appearing in lower case, a stream is a file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the *fdopen*( ), *fopen*( ) or *popen*( ) functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. See the **XSH** specification, **Section 2.4**, **Standard I/O Streams**.

STREAM
EX Appearing in upper case, STREAM refers to a full duplex connection between a process and an open device or pseudo-device. It optionally includes one or more intermediate processing modules that are interposed between the process end of the STREAM and the device driver (or pseudo-device driver) end of the STREAM. See the **XSH** specification, **Section 2.5**, STREAMS**.**

STREAM end

EX The STREAM end is the driver end of the STREAM and is also known as the downstream end of the STREAM.

STREAM head

EX The STREAM head is the beginning of the STREAM and is at the boundary between the system and the application process. This is also known as the upstream end of the STREAM.

STREAMS multiplexor

EX A driver with multiple STREAMS connected to it. Multiplexing with STREAMS connected above is referred to as N-to-1, or upper multiplexing. Multiplexing with STREAMS connected below is referred to as 1-to-N or lower multiplexing.

string

A contiguous sequence of bytes terminated by and including the first null byte.

subshell

A shell execution environment, distinguished from the main or current shell execution environment by the attributes described in the **XCU** specification, **Section 2.12**, **Shell Execution Environment**.

successfully transferred

For a write operation to a regular file, when the system ensures that all data written is readable on any subsequent open of the file (even one that follows a system or power failure) in the absence of a failure of the physical storage medium.

For a read operation, when an image of the data on the physical storage medium is available to the requesting process.

supplementary group ID

An attribute of a process used in determining file access permissions. A process has up to {NGROUPS_MAX} supplementary group IDs in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created. Whether a process' effective group ID is included in or omitted from its list of supplementary group IDs is unspecified.

suspended job

A job that has received a SIGSTOP, SIGTSTP, SIGTTIN or SIGTTOU signal that caused the process group to stop. A suspended job is a background job, but a background job is not necessarily a suspended job.

symbolic link

EX A type of file that contains a pathname. The pathname is interpolated into a pathname being resolved, during pathname resolution, to create a new pathname when it is encountered.

synchronised I/O completion

The state of an I/O operation that has either been successfully transferred or diagnosed as unsuccessful.

synchronised I/O data integrity completion

- For read, when the operation has been completed or diagnosed if unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronised read operation was requested, these write requests shall be successfully transferred prior to reading the data.

- For write, when the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred and all file system information required to retrieve the data is successfully transferred.

File attributes that are not necessary for data retrieval (access time, modification time, status change time) need not be successfully transferred prior to returning to the calling process.

synchronised I/O file integrity completion
Identical to a synchronised I/O data integrity completion with the addition that all file attributes relative to the I/O operation (including access time, modification time, status change time) will be successfully transferred prior to returning to the calling process.

synchronised I/O operation
An I/O operation performed on a file that provides the application assurance of the integrity of its data and files.

synchronous I/O operation
An I/O operation that causes the thread requesting the I/O to be blocked from further use of the processor until that I/O operation completes.

Note that a synchronous I/O operation does not imply synchronised I/O data integrity completion or synchronised I/O file integrity completion.

synchronously generated signal
A signal that is attributable to a specific thread.

For example, a thread executing an illegal instruction or touching invalid memory causes a synchronously generated signal. Being synchronous is a property of how the signal was generated and not a property of the signal number.

system
An implementation of the XSI.

system crash
An interval initiated by an unspecified circumstance that causes all processes (possibly other than special system processes) to be terminated in an undefined manner, after which any changes to the state and contents of files created or written to by an application prior to the
EX interval are undefined, except as required elsewhere in this specification set.

system console
EX An optional file that receives messages sent by *fmtmsg*( ) when the MM_CONSOLE flag is set.

system documentation
All documentation provided with an XSI-conformant implementation except for the Conformance Statement Questionnaire (CSQ). Electronically distributed documents for an XSI-conformant implementation are considered part of the system documentation.

system process
An implementation-dependent object, other than a process executing an application, that has a process ID.

system scheduling priority
A number used as advice to the system to alter process scheduling priorities. Raising the value should give a process additional preference when scheduling a process to run. Lowering the value should reduce the preference and make a process less likely to run. Typically, a process with higher system scheduling priority will run to completion more quickly than an equivalent process with lower system scheduling priority. A scheduling priority of zero specifies the default policy of the system.

This definition is not intended to suggest that all processes in a system have priorities that are comparable. Scheduling policy extensions such as adding real-time priorities make the notion of a single underlying priority for all scheduling policies problematic. Some systems may implement the features related to *nice* to affect all processes on the system, others to affect just the general time-sharing activities implied by this specification set, and others may have no effect at all. Because of the use of ‘‘implementation-dependent’’ in *nice* and *renice*, a wide range of implementation strategies is possible.

system reboot
An implementation-dependent sequence of events that may result in the loss of transitory data; that is, data that is not saved in permanent storage. For example, message queues, shared memory, semaphores and processes.

tab character
A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behaviour is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation.

terminal
(Or **terminal device**.) A character special file that obeys the specifications of the general terminal interface as described in Chapter 9 on page 119.

text column
A roughly rectangular block of characters capable of being laid out side-by-side next to other text columns on an output page or terminal screen. The widths of text columns are measured in column positions.

text file
A file that contains characters organised into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX} bytes in length, including the newline character. Although the XSI does not distinguish between text files and binary files (see the ISO C standard), many utilities only produce predictable or meaningful output when operating on text files. The standard utilities that have such restrictions always specify *text files* in their **STDIN** or **INPUT FILES** sections.

The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters gracefully or they explicitly describe their limitations within their individual sections. The only difference between text and binary files is that text files have lines of less than {LINE_MAX} bytes, with no NUL characters, each terminated by a newline character. The definition allows a file with a single newline character, but not a totally empty file, to be called a text file. If a file ends with an incomplete line it is not strictly a text file by this definition. The newline character referred to in this specification set is not some generic line separator, but a single character; files created on systems where they use multiple characters for ends of lines are not portable to all XSI-conformant systems without some translation process.

thread
A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, *errno* value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via *malloc*( ), directly addressable storage obtained through implementation-supplied functions and automatic variables, are accessible to all threads in the same process.

thread ID
Each thread in a process is uniquely identified during its lifetime by a value of type **pthread_t** called a thread ID.

thread list
An ordered set of runnable processes that all have the same ordinal value for their priority.

The ordering of processes on the list is determined by a scheduling policy or policies. The set of thread lists includes all runnable processes in the system.

thread-safe
A function that may be safely invoked concurrently by multiple threads. Examples are any ‘‘pure’’ function, a function which holds a mutex locked while it is accessing static storage, or objects shared among threads.

thread-specific data key
A process global handle of type **pthread_key_t** which is used for naming thread-specific data.

Although the same key value may be used by different threads, the values bound to the key by *pthread_setspecific*( ) and accessed by *pthread_getspecific*( ) are maintained on a per-thread basis and persist for the life of the calling thread.

tilde
The character ~.

timer
A mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed.

timer overrun
A condition that occurs each time a timer, for which there is already an expiration signal queued to the process, expires.

token
A sequence of characters that the shell considers as a single unit when reading input, according to the rules in the **XCU** specification, **Section 2.3**, **Token Recognition**. A token is either an operator or a word.

upshifting
The conversion of a lower-case character to its upper-case representation.

user database
A system database of implementation-dependent format that contains at least the following information for each user ID:

- User name
- Numerical user ID
- Initial numerical group ID
- Initial working directory
- Initial user program.

The initial numerical group ID is used by the *newgrp* utility. Any other circumstances under which the initial values are operative are implementation-dependent.

If the initial user program field is null, an implementation-dependent program is used.

If the initial working directory field is null, the interpretation of that field is implementation-dependent.

user ID

A non-negative integer that is used to identify a system user. When the identity of a user is FIPS associated with a process, a user ID value is referred to as a real user ID, an effective user ID or a saved set-user-ID.

user name

A string that is used to identify a user, as described in **user database** on page 34. To be portable across XSI-conformant systems, the value must be composed of characters from the portable filename character set. The hyphen should not be used as the first character of a portable user name.

utility

A program that can be called by name from a shell to perform a specific task, or related set of tasks. This program is either an executable file, such as might be produced by a compiler or linker system from computer source code, or a file of shell source code, directly interpreted by the shell. The program may have been produced by the user, provided by the system implementor, or acquired from an independent distributor. The term *utility* does not apply to the special built-in utilities provided as part of the XSI Shell Command Language; see the **XCU** specification, **Section 2.14**, **Special Built-in Utilities**. The system may implement certain utilities as shell functions (see the **XCU** specification, **Section 2.9.5**, **Function Definition Command**) or built-in utilities, but only an application that is aware of the command search order described in the **XCU** specification, **Command Search and Execution** in **Section 2.9.1** or of performance characteristics can discern differences between the behaviour of such a function or built-in utility and that of a true executable file.

variable

In the shell, a named parameter. See the **XCU** specification, **Section 2.5**, **Parameters and Variables**.

variable assignment

In the shell, a word consisting of the following parts:

```
varname=value
```

When used in a context where assignment is defined to occur (see the **XCU** specification, **Section 2.9.1**, **Simple Commands**) and at no other time, the *value* (representing a word or field) will be assigned as the value of the variable denoted by *varname*. The *varname* and *value* parts meet the requirements for a name and a word, respectively, except that they are delimited by the embedded unquoted equals-sign in addition to the delimiting described in the **XCU** specification, **Section 2.3**, **Token Recognition**. In all cases, the variable will be created if it did not already exist. If *value* is not specified, the variable will be given a null value.

An alternative form of variable assignment:

```
symbol=value
```

(where *symbol* is a valid word delimited by an equals-sign, but not a valid name) produces unspecified results. This form is used by the KornShell *name*[*expression*]=*value* syntax.

vertical-tab character

A character that in the output stream indicates that printing should start at the next vertical tabulation position. The vertical-tab is the character designated by '\v' in the C language. If the current position is at or past the last defined vertical tabulation position, the behaviour is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation.

white space
A sequence of one or more characters that belong to the **space** character class as defined via the LC_CTYPE category in the current locale.

In the POSIX locale, white space consists of one or more blank characters (space and tab characters), newline characters, carriage-return characters, form-feed characters and vertical-tab characters.

wide-character code (C language)
An integer value corresponding to a single graphic symbol or control code. See Section 4.3 on page 45.

wide-character string
A contiguous sequence of wide-character codes terminated by and including the first null wide-character code.

word
In the shell, a token other than an operator. In some cases a word is also a portion of a word token: in the various forms of parameter expansion (see the **XCU** specification, **Section 2.6.2**, **Parameter Expansion**), such as ${*name*–*word*}, and variable assignment, such as *name*=*word*, the word is the portion of the token depicted by *word*. The concept of a word is no longer applicable following word expansions only fields remain; see the **XCU** specification, **Section 2.6**, **Word Expansions**.

working directory
(Or **current working directory**.) A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash.

world-wide portability interface
Functions for handling characters in a codeset-independent manner.

write
To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. See the distinction between *display* and *write* in **display** on page 11.

XSI-conformant
A system which allows an application to be built using a set of services that are consistent across all systems that conform to this specification set.

zombie process
An inactive process that will be deleted at some later time when its parent process executes *wait*( ) or *waitpid*( ).

[n, m] and [n, m)
Notations denoting mathematical ranges. The square brackets [and] include the limit; the parentheses (and) exclude the limit; that is, if *x* is in [0, 1], it can be from 0 to 1 inclusive, but if *x* is in [0, 1), it can be from 0 up to but not including 1.

±0
The algebraic sign provides additional information about any variable that has the value zero. Although all precisions have distinct representations for +0, –0, +Inf and –Inf, the signs are significant in some circumstances, such as division by zero, and not in others.

CHANGE HISTORY

Issue 4

Numerous changes and additions are made for alignment with the ISO C standard and the ISO POSIX-1 standard.

Issue 4, Version 2

The following terms are added to support the adoption of additional traditional UNIX interfaces: *alternate signal stack*, *break value*, *data segment*, *driver*, *hard limit*, *host byte order*, *named STREAM*, *network byte order*, *network host database*, *network net database*, *network protocol database*, *network service database*, *pad*, *parent window*, *priority band*, *process virtual time*, *pseudo-terminal*, *real time*, *signal stack*, *socket*, *soft limit*, *STREAM* (second definition), *STREAM end*, *STREAM head*, *STREAMS multiplexor*, *symbolic link*, *system console* and *timer*.

Issue 5

Numerous terms are added to support adoption of the ISO POSIX Threads Extension and the ISO POSIX Realtime Extension.

Chapter 3

File Format Notation

The **STDIN**, **STDOUT**, **STDERR**, **INPUT FILES** and **OUTPUT FILES** sections of the utility descriptions use a syntax to describe the data organisation within the files, when that organisation is not otherwise obvious. The syntax is similar to that used by the **XSH** specification *printf*( ) function, as described in this chapter. When used in **STDIN** or **INPUT FILES** sections of the utility descriptions, this syntax describes the format that could have been used to write the text to be read, not a format that could be used by the *scanf*( ) function to read the input file.

The description of an individual record is as follows:

```
"<format>", [<arg1>, <arg2>, . . . , <argn>]
```

The *format* is a character string that contains three types of objects defined below:

characters
Characters that are not *escape sequences* or *conversion specifications*, as described below, are copied to the output.

escape sequences
Represent non-graphic characters.

conversion specifications
Specifies the output format of each argument. (See below.)

The following characters have the following special meaning in the format string:

" " (An empty character position.) One or more blank characters.

∆ Exactly one space character.

The notation for spaces allows some flexibility for application output. Note that an empty character position in *format* represents one or more blank characters on the output (not *white space*, which can include newline characters). Therefore, another utility that reads that output as its input must be prepared to parse the data using *scanf*( ), *awk*, and so forth. The ∆ character is used when exactly one space character is output.

The following table lists escape sequences and associated actions on display devices capable of the action.

Escape Sequence	**Represents Character**	**Terminal Action**
\\	backslash	None.
\a	alert	Attempts to alert the user through audible or visible notification.
\b	backspace	Moves the printing position to one column before the current position, unless the current position is the start of a line.
\f	form-feed	Moves the printing position to the initial printing position of the next logical page.
\n	newline	Moves the printing position to the start of the next line.
\r	carriage-return	Moves the printing position to the start of the current line.
\t	tab	Moves the printing position to the next tab position on the current line. If there are no more tab positions left on the line, the behaviour is undefined.
\v	vertical-tab	Moves the printing position to the start of the next vertical tab position. If there are no more vertical tab positions left on the page, the behaviour is undefined.

Table 3-1 Escape Sequences and Associated Actions

Each conversion specification is introduced by the percent-sign character (%). After the character %, the following appear in sequence:

flags Zero or more *flags*, in any order, that modify the meaning of the conversion specification.

field width An optional string of decimal digits to specify a minimum *field width*. For an output field, if the converted value has fewer bytes than the field width, it is padded on the left (or right, if the left-adjustment flag (–), described below, has been given to the field width).

precision Gives the minimum number of digits to appear for the d, o, i, u, x or X conversions (the field is padded with leading zeros), the number of digits to appear after the radix character for the e and f conversions, the maximum number of significant digits for the g conversion; or the maximum number of bytes to be written from a string in s conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

conversion characters
A conversion character (see below) that indicates the type of conversion to be applied.

The *flag* characters and their meanings are:

– The result of the conversion is left-justified within the field.

+ The result of a signed conversion always begins with a sign (+ or –).

<space> If the first character of a signed conversion is not a sign, a space character is prefixed to the result. This means that if the space character and + flags both appear, the space character flag is ignored.

# The value is to be converted to an alternative form. For c, d, i, u and s conversions, the behaviour is undefined. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x or X conversion, a non-zero result has 0x or 0X prefixed to it, respectively. For e, E, f, g and G conversions, the result always contains a radix character, even if no digits follow the radix character. For g and G conversions, trailing zeros are not removed from the result as they usually are.

0 For d, i, o, u, x, X, e, E, f, g and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and – flags both appear, the 0 flag is ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behaviour is undefined.

Each conversion character results in fetching zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are ignored.

The *conversion characters* and their meanings are:

d,i,o,u,x,X The integer argument is written as signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x and X). The d and i specifiers convert to signed decimal in the style **[–]***dddd*. The x conversion uses the numbers and letters 0123456789abcdef and the X conversion uses the numbers and letters 0123456789ABCDEF. The *precision* component of the argument specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of 0 is no characters. If both the field width and precision are omitted, the implementation may precede, follow or precede and follow numeric arguments of types d, i and u with blank characters; arguments of type o (octal) may be preceded with leading zeros.

The treatment of integers and spaces is different from the *printf*( ) function in that they can be surrounded with blank characters. This was done so that, given a format such as:

```
"%d\n", <foo>
```

the implementation could use a *printf*( ) call such as:

```
printf("%6d\n", foo);
```

and still conform. This notation is thus somewhat like *scanf*( ) in addition to *printf*( ).

f The floating point number argument is written in decimal notation in the style **[–]***ddd.ddd*, where the number of digits after the radix character (shown here as a decimal point) is equal to the *precision* specification. The LC_NUMERIC locale category determines the radix character to use in this format. If the *precision* is omitted from the argument, six digits are written after the radix character; if the *precision* is explicitly 0, no radix character appears.

e,E The floating point number argument is written in the style **[–]***d.ddd*e±*dd* (the symbol ± indicates either a plus or minus sign), where there is one digit before the radix character (shown here as a decimal point) and the number of digits after it is equal to the precision. The LC_NUMERIC locale category determines the radix character to use in this format. When the precision is missing, six digits are written after the radix character; if the precision is 0, no radix character appears. The E conversion character produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits. However, if the value to be written requires an exponent greater than two digits, additional exponent digits are written as necessary.

g,G — The floating point number argument is written in style f or e (or in style E in the case of a G conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted: style g is used only if the exponent resulting from the conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.

c — The integer argument is converted to an **unsigned char** and the resulting byte is written.

s — The argument is taken to be a string and bytes from the string are written until the end of the string or the number of bytes indicated by the *precision* specification of the argument is reached. If the precision is omitted from the argument, it is taken to be infinite, so all bytes up to the end of the string are written.

% — Write a % character; no argument is converted.

In no case does a non-existent or insufficient *field width* cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. The term *field width* should not be confused with the term *precision* used in the description of %s.

One difference from the C function *printf*( ) is that the l and h conversion characters are not used. As expressed by the **XCU** specification, there is no differentiation between decimal values for type **int**, type **long** or type **short**. The specifications %d or %i should be interpreted as an arbitrary length sequence of digits. Also, no distinction is made between single precision and double precision numbers (**float** or **double** in C). These are simply referred to as floating point numbers.

Many of the output descriptions in the **XCU** specification use the term *line*, such as:

`"%s"`, <*`input line`*>

Since the definition of *line* includes the trailing newline character already, there is no need to include a \n in the format; a double newline character would otherwise result.

Examples

To represent the output of a program that prints a date and time in the form Sunday, July 3, 10:02, where <*weekday*> and <*month*> are strings:

`"%s,△%s△%d,△%d:%.2d\n"`, <*`weekday`*>, <*`month`*>, <*`day`*>, <*`hour`*>, <*`min`*>

To show π written to 5 decimal places:

`"pi△=△%.5f\n"`, <*`value of`* π>

To show an input file format consisting of five colon-separated fields:

`"%s:%s:%s:%s:%s\n"`, <*`arg1`*>, <*`arg2`*>, <*`arg3`*>, <*`arg4`*>, <*`arg5`*>

Chapter 4

Character Set

4.1 Portable Character Set

Conforming implementations support one or more coded character sets. Each supported locale includes the *portable character set* specified in the following table.

Symbolic Name	Glyph	Symbolic Name	Glyph	Symbolic Name	Glyph
				<circumflex>	^
<NUL>		<colon>	:	<circumflex-accent>	^
<alert>		<semicolon>	;	<underscore>	_
<backspace>		<less-than-sign>	<	<underline>	_
<tab>		<equals-sign>	=	<low-line>	_
<newline>		<greater-than-sign>	>	<grave-accent>	`
<vertical-tab>		<question-mark>	?	<a>	a
<form-feed>		<commercial-at>	@	<b>	b
<carriage-return>		<A>	A	<c>	c
<space>		<B>	B	<d>	d
<exclamation-mark>	!	<C>	C	<e>	e
<quotation-mark>	"	<D>	D	<f>	f
<number-sign>	#	<E>	E	<g>	g
<dollar-sign>	$	<F>	F	<h>	h
<percent-sign>	%	<G>	G	<i>	i
<ampersand>	&	<H>	H	<j>	j
<apostrophe>	'	<I>	I	<k>	k
<left-parenthesis>	(	<J>	J	<l>	l
<right-parenthesis>	)	<K>	K	<m>	m
<asterisk>	*	<L>	L	<n>	n
<plus-sign>	+	<M>	M	<o>	o
<comma>	,	<N>	N	<p>	p
<hyphen>	-	<O>	O	<q>	q
<hyphen-minus>	-	<P>	P	<r>	r
<period>	.	<Q>	Q	<s>	s
<full-stop>	.	<R>	R	<t>	t
<slash>	/	<S>	S	<u>	u
<solidus>	/	<T>	T	<v>	v
<zero>	0	<U>	U	<w>	w
<one>	1	<V>	V	<x>	x
<two>	2	<W>	W	<y>	y
<three>	3	<X>	X	<z>	z
<four>	4	<Y>	Y	<left-brace>	{
<five>	5	<Z>	Z	<left-curly-bracket>	{
<six>	6	<left-square-bracket>	[	<vertical-line>	|
<seven>	7	<backslash>	\	<right-brace>	}
<eight>	8	<reverse-solidus>	\	<right-curly-bracket>	}
<nine>	9	<right-square-bracket>	]	<tilde>	~

Table 4-1 Portable Character Set

Table 4-1 on page 43 defines the characters in the portable character set and the corresponding symbolic character names used to identify each character in a character set description file. The table contains more than one symbolic character name for characters whose traditional name differs from the chosen name.

This specification set places only the following requirements on the encoded values of the characters in the portable character set:

- If the encoded values associated with each member of the portable character set are not invariant across all locales supported by the implementation, the results achieved by an application accessing those locales are unspecified.
- The encoded values associated with the digits 0 to 9 will be such that the value of each character after 0 will be one greater than the value of the previous character.
- A null character, NUL, which has all bits set to zero, will be in the set of characters.
- The encoded values associated with the members of the portable character set are each represented in a single byte. Moreover, if the value is stored in an object of C-language type **char**, it is guaranteed to be positive (except the NUL, which is always zero).

4.2 Character Encoding

The POSIX locale contains the characters in Table 4-1 on page 43, which have the properties listed in Section 5.3.1 on page 52. Implementations may also add other characters. In other locales, the presence, meaning and representation of any additional characters is locale-specific.

In locales other than the POSIX locale, a character may have a state-dependent encoding. There are two types of these encodings:

- A single-shift encoding (where each character not in the initial shift state is preceded by a shift code) can be defined if each shift-code and character sequence is considered a multi-byte character. This is done using the concatenated-constant format in a character set description file, as described in Section 4.4 on page 45. If the implementation supports a character encoding of this type, all of the standard utilities in the **XCU** specification will support it. Use of a single-shift encoding with any of the functions in the **XSH** specification that do not specifically mention the effects of state-dependent encoding is implementation-dependent.
- A locking-shift encoding (where the state of the character is determined by a shift code that may affect more than the single character following it) cannot be defined with the current character set description file format. Use of a locking-shift encoding with any of the standard utilities in the **XCU** specification or with any of the functions in the **XSH** specification that do not specifically mention the effects of state-dependent encoding is implementation-dependent.

While in the initial shift state, all characters in the portable character set retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state. A byte with all bits zero is interpreted as the null character independent of shift state. Thus a byte with all bits zero must never occur in the second or subsequent bytes of a character.

The maximum allowable number of bytes in a character in the current locale is indicated by MB_CUR_MAX, defined in the **XSH** specification **<stdlib.h>**, and by the **<mb_cur_max>** value in a character set description file; see Section 4.4 on page 45. The implementation's maximum number of bytes in a character is defined by the C-language macro {MB_LEN_MAX}.

4.3 C Language Wide-character Codes

In the shell, the standard utilities are written so that the encodings of characters are described by the locale's LC_CTYPE definition (see Section 5.3.1 on page 52) and there is no differentiation between characters consisting of single octets (8-bit bytes), larger bytes, or multiple bytes. However, in the C language, a differentiation is made. To ease the handling of variable length characters, the C language has introduced the concept of wide character codes.

All wide-character codes in a given process consist of an equal number of bits. This is in contrast to characters, which can consist of a variable number of bytes. The byte or byte sequence that represents a character can also be represented as a wide-character code. Wide-character codes thus provide a uniform size for manipulating text data. A wide-character code having all bits zero is the null wide-character code (see **null wide-character code** on page 21), and terminates wide-character strings (see Section 4.3). The wide-character value for each member of the Portable Character Set will equal its value when used as the lone character in an integer character constant. Wide-character codes for other characters are locale- and implementation-dependent. State shift bytes do not have a wide-character code representation.

4.4 Character Set Description File

Implementations provide a character set description file for at least one coded character set supported by the implementation. These files are referred to elsewhere in this specification set as *charmap* files. It is implementation-dependent whether or not users or applications can provide additional character set description files.

This specification set does not require that multiple character sets or codesets be supported. Although multiple charmap files are supported, it is the responsibility of the implementation to provide the file or files; if only one is provided, only that one will be accessible using the *localedef* utility's −**f** option (although in the case of just one file on the system, −**f** is not useful).

Each character set description file defines characteristics for the coded character set and the encoding for the characters specified in Table 4-1 on page 43 and may define encoding for additional characters supported by the implementation. Other information about the coded character set may also be in the file. Coded character set character values are defined using symbolic character names followed by character encoding values.

The character set description file provides:

- The capability to describe character set attributes (such as collation order or character classes) independent of character set encoding, and using only the characters in the portable character set. This makes it possible to create generic *localedef* source files for all codesets that share the portable character set (such as the ISO 8859 family or IBM Extended ASCII).
- Standardised symbolic names for all characters in the portable character set, making it possible to refer to any such character regardless of encoding.

The charmap file was introduced to resolve problems with the portability of, especially, *localedef* sources. This specification set assumes that the portable character set is constant across all locales, but does not prohibit implementations from supporting two incompatible codings, such as both ASCII and EBCDIC. Such dual-support implementations should have all charmaps and *localedef* sources encoded using one portable character set, in effect cross-compiling for the other environment. Naturally, charmaps (and *localedef* sources) are only portable without transformation between systems using the same encodings for the portable character set. They can, however, be transformed between two sets using only a subset of the actual characters (the portable set). However, the particular coded character set used for an application or an implementation does not necessarily imply different characteristics or collation; on the contrary, these attributes should in many cases be identical, regardless of codeset. The charmap provides the capability to define a common locale definition for multiple codesets (the same *localedef* source can be used for codesets with different extended characters; the ability in the charmap to define empty names allows for characters missing in certain codesets).

Each symbolic name specified in Table 4-1 on page 43 is included in the file and is mapped to a unique encoding value (except for those symbolic names that are shown with identical glyphs). If the control characters commonly associated with the symbolic names in the following table are supported by the implementation, the symbolic names and their corresponding encoding values are included in the file. Some of the encodings associated with the symbolic names in this table may be the same as characters in the portable character set table.

<ACK>	<DC2>	<ENQ>	<FS>	<IS4>	<SOH>
<BEL>	<DC3>	<EOT>	<GS>	<LF>	<STX>
<BS>	<DC4>	<ESC>	<HT>	<NAK>	<SUB>
<CAN>	<DEL>	<ETB>	<IS1>	<RS>	<SYN>
<CR>	<DLE>	<ETX>	<IS2>	<SI>	<US>
<DC1>	<EM>	<FF>	<IS3>	<SO>	<VT>

Table 4-2 Control Character Set

The following declarations can precede the character definitions. Each must consist of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more blank characters, 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. The characters of the name must be taken from the set of characters with visible glyphs defined in Table 4-1 on page 43.

<mb_cur_max> The maximum number of bytes in a multi-byte character. This defaults to 1.

<mb_cur_min> An unsigned positive integer value that defines the minimum number of bytes in a character for the encoded character set. EX On XSI-conformant systems, **<mb_cur_min>** is always 1.

<escape_char> The escape character used to indicate that the characters following will be interpreted in a special way, as defined later in this section. This defaults to backslash (\), which is the character glyph used in all the following text and examples, unless otherwise noted.

<comment_char> The character that when placed in column 1 of a charmap line, is used to indicate that the line is to be ignored. The default character is the number sign (#).

The character set mapping definitions will be all the lines immediately following an identifier line containing the string **CHARMAP** starting in column 1, and preceding a trailer line containing the string **END CHARMAP** starting in column 1. Empty lines and lines containing a **<comment_char>** in the first column will be ignored. Each non-comment line of the character set mapping definition (that is, between the **CHARMAP** and **END CHARMAP** lines of the file) must be in either of two forms:

`"%s %s %s\n"`, <*symbolic-name*>, <*encoding*>, <*comments*>

or:

`"%s...%s %s %s\n"`, <*symbolic-name*>, <*symbolic-name*>, <*encoding*>, <*comments*>

In the first format, the line in the character set mapping definition defines a single symbolic name and a corresponding encoding. A symbolic name is one or more characters from the set shown with visible glyphs in Table 4-1 on page 43, 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.

In the second format, the line in the character set mapping definition defines a range of one or more symbolic names. In this form, the symbolic names must consist of zero or more non-numeric characters from the set shown with visible glyphs in Table 4-1 on page 43, followed by an integer formed by one or more decimal digits. The characters preceding the integer must be identical in the two symbolic names, and the integer formed by the digits in the second symbolic name must be equal to or greater than the integer formed by the digits in the first name. This is interpreted as a series of symbolic names formed from the common part and each of the integers between the first and the second integer, inclusive. As an example, <j0101>...<j0104> is interpreted as the symbolic names <j0101>, <j0102>, <j0103> and <j0104>, in that order.

A character set mapping definition line must exist for all symbolic names specified in Table 4-1 on page 43, and must define the coded character value that corresponds to the character glyph indicated in the table, or the coded character value that corresponds with the control character symbolic name. If the control characters commonly associated with the symbolic names in Table 4-2 on page 46 are supported by the implementation, the symbolic name and the corresponding encoding value must be included in the file. Additional unique symbolic names may be included. A coded character value can be represented by more than one symbolic name.

The encoding part is expressed as one (for single-byte character values) or more concatenated decimal, octal or hexadecimal constants in the following formats:

`"%cd%d"`, <*escape_char*>, <*decimal byte value*>

`"%cx%x"`, <*escape_char*>, <*hexadecimal byte value*>

`"%c%o"`, <*escape_char*>, <*octal byte value*>

Decimal constants must be represented by two or three decimal digits, preceded by the escape character and the lower-case letter d; for example, \d05, \d97 or \d143. Hexadecimal constants must be represented by two hexadecimal digits, preceded by the escape character and the lower-case letter x; for example, \x05, \x61 or \x8f. Octal constants must be represented by two or three octal digits, preceded by the escape character; for example, \05, \141 or \217. In a portable charmap file, each constant must represent an 8-bit byte. Implementations supporting other byte sizes may allow constants to represent values larger than those that can be represented in 8-bit bytes, and to allow additional digits in constants. When constants are concatenated for multi-byte character values, they must be of the same type, and interpreted in byte order from first to last with the least significant byte of the multi-byte character specified by the last constant. The manner in which these constants are represented in the character stored in the system is implementation-dependent. (This big endian notation was chosen for reasons of portability. There is no requirement that the internal representation in the computer memory be in this same order.) Omitting bytes from a multi-byte character definition produces undefined results.

In lines defining ranges of symbolic names, the encoded value is the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent symbolic names defined by the range will have encoding values in increasing order. For example, the line:

```
<j0101>...<j0104>    \d129\d254
```

will be interpreted as:

```
<j0101>              \d129\d254
<j0102>              \d129\d255
<j0103>              \d130\d0
<j0104>              \d130\d1
```

Note that this line will be interpreted as the example even on systems with bytes larger than 8 bits.

The comment is optional.

For the interpretation of the dollar sign and the number sign, see **dollar sign** on page 12 and **number sign** on page 21.

Chapter 5

Locale

5.1 General

A *locale* is the definition of the subset of a user's environment that depends on language and cultural conventions. It 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_CTYPE Character classification and case conversion.

LC_COLLATE Collation order.

LC_TIME Date and time formats.

LC_NUMERIC Numeric, non-monetary formatting.

LC_MONETARY Monetary formatting.

LC_MESSAGES Formats of informative and diagnostic messages and interactive responses.

The standard utilities in the **XCU** specification base their behaviour on the current locale, as defined in the **ENVIRONMENT VARIABLES** section for each utility. The behaviour of some of the C-language functions defined in the **XSH** specification will also be modified based on the current locale, as defined by the last call to *setlocale*( ).

EX Locales other than those supplied by the implementation can be created by the application via the *localedef* utility, if it is provided; see the **XCU** specification. This capability is supported on all X/Open systems where the {POSIX2_LOCALEDEF} or {XOPEN_XCU_VERSION} options are supported; see the **XSH** specification **<unistd.h>**. Even if *localedef* is not provided, all implementations conforming to the **XSH** specification provide one or more locales that behave as described in this chapter. The input to the utility is described in Section 5.3 on page 50. The value that is used to specify a locale when using environment variables will be the string specified as the *name* operand to the *localedef* utility when the locale was created. The strings "C" and "POSIX" are reserved as identifiers for the POSIX locale (see Section 5.2 on page 50). When the value of a locale environment variable begins with a slash (/), it is interpreted as the pathname of the locale definition; the type of file (regular, directory, and so forth) used to store the locale definition is implementation-dependent. If the value does not begin with a slash, the mechanism used to locate the locale is implementation-dependent.

If different character sets are used by the locale categories, the results achieved by an application utilising these categories are undefined. Likewise, if different codesets are used for the data being processed by interfaces whose behaviour is dependent on the current locale, or the codeset is different from the codeset assumed when the locale was created, the result is also undefined.

Applications can select the desired locale by invoking the *setlocale*( ) function (or equivalent) with the appropriate value. If the function is invoked with an empty string, such as:

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

the value of the corresponding environment variable is used. If the environment variable is unset or is set to the empty string, the implementation sets the appropriate environment as defined in Chapter 6 on page 93.

5.2 POSIX Locale

All systems provide a *POSIX locale*, also known as the C locale. The behaviour of standard utilities and functions in the POSIX locale is as if the locale was defined via the *localedef* utility with input data from the POSIX locale tables in Section 5.3.

The tables in Section 5.3 describe the characteristics and behaviour of the POSIX locale for data consisting entirely of characters from the portable character set and the control character set. For other characters, the behaviour is unspecified. For C-language programs, the POSIX locale is the default locale when the *setlocale*( ) function is not called.

The POSIX locale can be specified by assigning to the appropriate environment variables the values "C" or "POSIX".

All implementations define a locale as the default locale, to be invoked when no environment variables are set, or set to the empty string. This default locale can be the POSIX locale or any other, implementation-dependent locale. Some implementations may provide facilities for local installation administrators to set the default locale, customising it for each location. This specification set does not require such a facility.

5.3 Locale Definition

Locales can be described with the file format presented in this section. The file format is that accepted by the *localedef* utility. For the purposes of this section, the file is referred to as the *locale definition file*, but no locales are affected by this file unless it is processed by *localedef* or some similar mechanism. Any requirements in this section imposed upon the utility apply to *localedef* or to any other similar utility used to install locale information using the locale definition file format described here.

The locale definition file must contain one or more locale category source definitions, and must not contain more than one definition for the same locale category. If the file contains source definitions for more than one category, implementation-dependent categories, if present, must appear after the categories defined by Section 5.1 on page 49. A category source definition must contain either the definition of a category or a **copy** directive. For a description of the **copy** directive, see *localedef*. In the event that some of the information for a locale category, as specified in this specification, is missing from the locale source definition, the behaviour of that category, if it is referenced, is unspecified.

A category source definition consists of a category header, a category body and a category trailer. A category header consists of the character string naming of the category, beginning with the characters LC_. The category trailer consists of the string END, followed by one or more blank characters and the string used in the corresponding category header.

The category body consists of one or more lines of text. Each line contains an identifier, optionally followed by one or more operands. Identifiers are either keywords, identifying a particular locale element, or collating elements. In addition to the keywords defined in this specification, the source can contain implementation-dependent keywords. Each keyword within a locale must have a unique name (that is, two categories cannot have a commonly-named keyword); no keyword can start with the characters LC_. Identifiers must be separated from the operands by one or more blank characters.

Operands must be characters, collating elements or strings of characters. Strings must be enclosed in double-quotes. Literal double-quotes within strings must be preceded by the <*escape character*>, described below. When a keyword is followed by more than one operand, the operands must be separated by semicolons; blank characters are allowed both before and after a semicolon.

The first category header in the file can be preceded by a line modifying the comment character. It has the following format, starting in column 1:

`"comment_char %c\n"`, <*`comment character`*>

The comment character defaults to the number sign (#). Blank lines and lines containing the <*comment character*> in the first position are ignored.

The first category header in the file can be preceded by a line modifying the escape character to be used in the file. It has the following format, starting in column 1:

`"escape_char %c\n"`, <*`escape character`*>

The escape character defaults to backslash, which is the character used in all examples shown in this specification.

A line can be continued by placing an escape character as the last character on the line; this continuation character will be discarded from the input. Although the implementation need not accept any one portion of a continued line with a length exceeding {LINE_MAX} bytes, it places no limits on the accumulated length of the continued line. Comment lines cannot be continued on a subsequent line using an escaped newline character.

Individual characters, characters in strings, and collating elements must be represented using symbolic names, as defined below. In addition, characters can be represented using the characters themselves or as octal, hexadecimal or decimal constants. When non-symbolic notation is used, the resultant locale definitions will in many cases not be portable between systems. The left angle bracket (<) is a reserved symbol, denoting the start of a symbolic name; when used to represent itself it must be preceded by the escape character. The following rules apply to character representation:

1. A character can be represented via a symbolic name, enclosed within angle brackets "<" and ">". The symbolic name, including the angle brackets, must exactly match a symbolic name defined in the charmap file specified via the *localedef* **–f** option, and will be replaced by a character value determined from the value associated with the symbolic name in the charmap file. The use of a symbolic name not found in the charmap file constitutes an error, unless the category is LC_CTYPE or LC_COLLATE, in which case it constitutes a warning condition (see *localedef* for a description of action resulting from errors and warnings). The specification of a symbolic name in a **collating–element** or **collating–symbol** section that duplicates a symbolic name in the charmap file (if present) is an error. Use of the escape character or a right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character.

   Example:

   ```
   <c>;<c–cedilla>  "<M><a><y>"
   ```

2. A character can be represented by the character itself, in which case the value of the character is implementation-dependent. Within a string, the double-quote character, the escape character and the right angle bracket character must be escaped (preceded by the escape character) to be interpreted as the character itself. Outside strings, the characters:

   ```
   ,      ;      <      >      escape_char
   ```

   must be escaped to be interpreted as the character itself.

   Example:

   ```
   c    β   "May"
   ```

3. A character can be represented as an octal constant. An octal constant is specified as the escape character followed by two or more octal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   ```
   \143;\347;\143\150   "\115\141\171"
   ```

4. A character can be represented as a hexadecimal constant. A hexadecimal constant is specified as the escape character followed by an x followed by two or more hexadecimal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   ```
   \x63;\xe7;\x63\x68   "\x4d\x61\x79"
   ```

5. A character can be represented as a decimal constant. A decimal constant is specified as the escape character followed by a d followed by two or more decimal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   ```
   \d99;\d231;\d99\d104  "\d77\d97\d121"
   ```

Implementations may accept single-digit octal, decimal or hexadecimal constants following the escape character. Only characters existing in the character set for which the locale definition is created can be specified, whether using symbolic names, the characters themselves, or octal, decimal or hexadecimal constants. If a charmap file is present, only characters defined in the charmap can be specified using octal, decimal or hexadecimal constants. Symbolic names not present in the charmap file can be specified and will be ignored, as specified under item 1 above.

5.3.1 LC_CTYPE

The LC_CTYPE category defines character classification, case conversion and other character attributes. In addition, a series of characters can be represented by three adjacent periods representing an ellipsis symbol (...). The ellipsis specification is interpreted as meaning that all values between the values preceding and following it represent valid characters. The ellipsis specification is valid only within a single encoded character set; that is, within a group of characters of the same size. An ellipsis is interpreted as including in the list all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value of the character following the ellipsis.

Example:

```
\x30;...;\x39;
```

includes in the character class all characters with encoded values between the endpoints.

The following keywords are recognised. In the descriptions, the term ''automatically included'' means that it is not an error either to include or omit any of the referenced characters; the implementation will provide them if missing (even if the entire keyword is missing) and accept them silently if present. When the implementation automatically includes a missing character, it will have an encoded value dependent on the charmap file in effect (see the description of the *localedef* −**f** option); otherwise, it will have a value derived from an implementation-dependent character mapping.

The character classes **digit**, **xdigit**, **lower**, **upper** and **space** have a set of automatically included characters. These only need to be specified if the character values (that is, encoding) differ from the implementation default values. It is not possible to define a locale without these automatically included characters unless some implementation extension is used to prevent their inclusion. Such a definition would not be a proper superset of the C or POSIX locale and thus, it might not be possible for applications conforming to the XSI to work properly.

upper Define characters to be classified as upper-case letters.

In the POSIX locale, the 26 upper-case letters are included:

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

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct** or **space** can be specified. The upper-case letters A to Z, as defined in Section 4.4 on page 45 (the portable character set), are automatically included in this class.

lower Define characters to be classified as lower-case letters.

In the POSIX locale, the 26 lower-case letters are included:

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

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct** or **space** can be specified. The lower-case letters a to z of the portable character set are automatically included in this class.

alpha Define characters to be classified as letters.

In the POSIX locale, all characters in the classes **upper** and **lower** are included.

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct** or **space** can be specified. Characters classified as either **upper** or **lower** are automatically included in this class.

digit Define the characters to be classified as numeric digits.

In the POSIX locale, only:

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

are included.

In a locale definition file, only the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 can be specified, and in contiguous ascending sequence by numerical value. The digits 0 to 9 of the portable character set are automatically included in this class.

The definition of character class **digit** requires that only ten characters the ones defining digits can be specified; alternative digits (for example, Hindi or Kanji) cannot be specified here. However, the encoding may vary if an implementation supports more than one encoding.

space Define characters to be classified as white-space characters.

In the POSIX locale, at a minimum, the characters space, form-feed, newline, carriage-return, tab and vertical-tab are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **graph** or **xdigit** can be specified. The characters space, form-feed, newline, carriage-return, tab and vertical-tab of the portable character set, and any characters included in the class **blank** are automatically included in this class.

cntrl Define characters to be classified as control characters.

In the POSIX locale, no characters in classes **alpha** or **print** are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **punct**, **graph**, **print** or **xdigit** can be specified.

punct Define characters to be classified as punctuation characters.

In the POSIX locale, neither the space character nor any characters in classes **alpha**, **digit** or **cntrl** are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **cntrl**, **xdigit** or as the space character can be specified.

graph Define characters to be classified as printable characters, not including the space character.

In the POSIX locale, all characters in classes **alpha**, **digit** and **punct** are included; no characters in class **cntrl** are included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit** and **punct** are automatically included in this class. No character specified for the keyword **cntrl** can be specified.

print Define characters to be classified as printable characters, including the space character.

In the POSIX locale, all characters in class **graph** are included; no characters in class **cntrl** are included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit**, **punct** and the space character are automatically included in this class. No character specified for the keyword **cntrl** can be specified.

xdigit Define the characters to be classified as hexadecimal digits.

In the POSIX locale, only:

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

are included.

In a locale definition file, only the characters defined for the class **digit** can be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits 10 to 15 inclusive, with each set in ascending order (for example A, B, C, D, E, F, a, b, c, d, e, f). The digits 0 to 9, the upper-case letters A to F and the lower-case letters a to f of the portable character set are automatically included in this class.

The definition of character class **xdigit** requires that the characters included in character class **digit** be included here also.

blank Define characters to be classified as blank characters.

In the POSIX locale, only the space and tab characters are included.

In a locale definition file, the characters space and tab are automatically included in this class.

EX **charclass** Define one or more locale-specific character class names as strings separated by semicolons. Each named character class can then be defined subsequently in the LC_CTYPE definition. A character class name consists of at least one and at most {CHARCLASS_NAME_MAX} bytes of alphanumeric characters from the portable filename character set. The first character of a character class name cannot be a digit. The name cannot match any of the LC_CTYPE keywords defined in this specification.

charclass-name Define characters to be classified as belonging to the named locale-specific character class. In the POSIX locale, the locale-specific named character classes need not exist.

If a class name is defined by a **charclass** keyword, but no characters are subsequently assigned to it, this is not an error; it represents a class without any characters belonging to it.

The *charclass-name* can be used as the *property* argument to the *wctype*( ) function, in regular expression and shell pattern-matching bracket expressions, and by the *tr* command.

toupper Define the mapping of lower-case letters to upper-case letters.

In the POSIX locale, at a minimum, the 26 lower-case characters:

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

are mapped to the corresponding 26 upper-case characters:

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

In a locale definition file, the operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair enclosed by parentheses. The first character in each pair is the lower-case letter, the second the corresponding upper-case letter. Only characters specified for the keywords **lower** and **upper** can be specified. The lower-case letters a to z, and their corresponding upper-case letters A to Z, of the portable character set are automatically included in this mapping, but only when the **toupper** keyword is omitted from the locale definition.

tolower Define the mapping of upper-case letters to lower-case letters.

In the POSIX locale, at a minimum, the 26 upper-case characters:

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

are mapped to the corresponding 26 lower-case characters:

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

In a locale definition file, the operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair enclosed by parentheses. The first character in each pair is the upper-case letter, the second the corresponding lower-case letter. Only characters specified for the keywords **lower** and **upper** can be specified. If the **tolower** keyword is omitted from the locale definition, the mapping will be the reverse mapping of the one specified for **toupper**.

copy Specify the name of an existing locale to be used as the definition of this category. If this keyword is specified, no other keyword can be specified.

The following table shows the character class combinations allowed.

In	**Can Also Belong To**										
Class	upper	lower	alpha	digit	space	cntrl	punct	graph	print	xdigit	blank
upper		-	A	x	x	x	x	A	A	-	x
lower	-		A	x	x	x	x	A	A	-	x
alpha	-	-		x	x	x	x	A	A	-	x
digit	x	x	x		x	x	x	A	A	A	x
space	x	x	x	x		-	*	*	*	x	-
cntrl	x	x	x	x	-		x	x	x	x	-
punct	x	x	x	x	-	x		A	A	x	-
graph	-	-	-	-	-	x	-		A	-	-
print	-	-	-	-	-	x	-	-		-	-
xdigit	-	-	-	-	x	x	x	A	A		x
blank	x	x	x	x	A	-	*	*	*	x	

Table 5-1 Valid Character Class Combinations

Notes:

1. Explanation of codes:

   A Automatically included; see text.

   - Permitted.

   x Mutually exclusive.

   * See note 2.

2. The space character, which is part of the **space** and **blank** classes, cannot belong to **punct** or **graph**, but automatically belongs to the **print** class. Other **space** or **blank** characters can be classified as any of **punct**, **graph** or **print**.

The character classifications for the POSIX locale follow; the code listing depicting the *localedef* input, the table representing the same information, sorted by character.

```
LC_CTYPE
# The following is the POSIX locale LC_CTYPE.
# "alpha" is by default "upper" and "lower"
# "alnum" is by definition "alpha" and "digit"
# "print" is by default "alnum", "punct" and the <space> character
# "graph" is by default "alnum" and "punct"
#
upper    <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
         <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>
#
lower    <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
         <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>
#
digit    <zero>;<one>;<two>;<three>;<four>;<five>;<six>;\
         <seven>;<eight>;<nine>
#
space    <tab>;<newline>;<vertical-tab>;<form-feed>;\
         <carriage-return>;<space>
#
cntrl    <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
         <form-feed>;<carriage-return>;\
         <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;\
         <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
         <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
         <IS1>;<DEL>
#
punct    <exclamation-mark>;<quotation-mark>;<number-sign>;\
         <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
         <left-parenthesis>;<right-parenthesis>;<asterisk>;\
         <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
         <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
         <greater-than-sign>;<question-mark>;<commercial-at>;\
         <left-square-bracket>;<backslash>;<right-square-bracket>;\
         <circumflex>;<underscore>;<grave-accent>;<left-curly-bracket>;\
         <vertical-line>;<right-curly-bracket>;<tilde>
#
xdigit   <zero>;<one>;<two>;<three>;<four>;<five>;<six>;<seven>;\
         <eight>;<nine>;<A>;<B>;<C>;<D>;<E>;<F>;<a>;<b>;<c>;<d>;<e>;<f>
#
blank    <space>;<tab>
#
toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);(<z>,<Z>)
#
tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);\
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);\
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);\
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);(<Z>,<z>)
END LC_CTYPE
```

Symbolic Name	Other Case	Character Classes
<NUL>		**cntrl**
<SOH>		**cntrl**
<STX>		**cntrl**
<ETX>		**cntrl**
<EOT>		**cntrl**
<ENQ>		**cntrl**
<ACK>		**cntrl**
<alert>		**cntrl**
<backspace>		**cntrl**
<tab>		**cntrl, space, blank**
<newline>		**cntrl, space**
<vertical-tab>		**cntrl, space**
<form-feed>		**cntrl, space**
<carriage-return>		**cntrl, space**
<SO>		**cntrl**
<SI>		**cntrl**
<DLE>		**cntrl**
<DC1>		**cntrl**
<DC2>		**cntrl**
<DC3>		**cntrl**
<DC4>		**cntrl**
<NAK>		**cntrl**
<SYN>		**cntrl**
<ETB>		**cntrl**
<CAN>		**cntrl**
<EM>		**cntrl**
<SUB>		**cntrl**
<ESC>		**cntrl**
<IS4>		**cntrl**
<IS3>		**cntrl**
<IS2>		**cntrl**
<IS1>		**cntrl**
<space>		**space, print, blank**
<exclamation-mark>		**punct, print, graph**
<quotation-mark>		**punct, print, graph**
<number-sign>		**punct, print, graph**
<dollar-sign>		**punct, print, graph**
<percent-sign>		**punct, print, graph**
<ampersand>		**punct, print, graph**
<apostrophe>		**punct, print, graph**
<left-parenthesis>		**punct, print, graph**
<right-parenthesis>		**punct, print, graph**

Symbolic Name	Other Case	Character Classes
<asterisk>		**punct, print, graph**
<plus-sign>		**punct, print, graph**
<comma>		**punct, print, graph**
<hyphen>		**punct, print, graph**
<period>		**punct, print, graph**
<slash>		**punct, print, graph**
<zero>		**digit, xdigit, print, graph**
<one>		**digit, xdigit, print, graph**
<two>		**digit, xdigit, print, graph**
<three>		**digit, xdigit, print, graph**
<four>		**digit, xdigit, print, graph**
<five>		**digit, xdigit, print, graph**
<six>		**digit, xdigit, print, graph**
<seven>		**digit, xdigit, print, graph**
<eight>		**digit, xdigit, print, graph**
<nine>		**digit, xdigit, print, graph**
<colon>		**punct, print, graph**
<semicolon>		**punct, print, graph**
<less-than-sign>		**punct, print, graph**
<equals-sign>		**punct, print, graph**
<greater-than-sign>		**punct, print, graph**
<question-mark>		**punct, print, graph**
<commercial-at>		**punct, print, graph**
<A>	<a>	**upper, xdigit, alpha, print, graph**
<B>	<b>	**upper, xdigit, alpha, print, graph**
<C>	<c>	**upper, xdigit, alpha, print, graph**
<D>	<d>	**upper, xdigit, alpha, print, graph**
<E>	<e>	**upper, xdigit, alpha, print, graph**
<F>	<f>	**upper, xdigit, alpha, print, graph**
<G>	<g>	**upper, alpha, print, graph**
<H>	<h>	**upper, alpha, print, graph**
<I>	<i>	**upper, alpha, print, graph**
<J>	<j>	**upper, alpha, print, graph**
<K>	<k>	**upper, alpha, print, graph**
<L>	<l>	**upper, alpha, print, graph**
<M>	<m>	**upper, alpha, print, graph**
<N>	<n>	**upper, alpha, print, graph**
<O>	<o>	**upper, alpha, print, graph**
<P>	<p>	**upper, alpha, print, graph**
<Q>	<q>	**upper, alpha, print, graph**
<R>	<r>	**upper, alpha, print, graph**
<S>	<s>	**upper, alpha, print, graph**
<T>	<t>	**upper, alpha, print, graph**
<U>	<u>	**upper, alpha, print, graph**
<V>	<v>	**upper, alpha, print, graph**
<W>	<w>	**upper, alpha, print, graph**

Symbolic Name	Other Case	Character Classes
<X>	<x>	**upper, alpha, print, graph**
<Y>	<y>	**upper, alpha, print, graph**
<Z>	<z>	**upper, alpha, print, graph**
<left-square-bracket>		**punct, print, graph**
<backslash>		**punct, print, graph**
<right-square-bracket>		**punct, print, graph**
<circumflex>		**punct, print, graph**
<underscore>		**punct, print, graph**
<grave-accent>		**punct, print, graph**
<a>	<A>	**lower, xdigit, alpha, print, graph**
<b>	<B>	**lower, xdigit, alpha, print, graph**
<c>	<C>	**lower, xdigit, alpha, print, graph**
<d>	<D>	**lower, xdigit, alpha, print, graph**
<e>	<E>	**lower, xdigit, alpha, print, graph**
<f>	<F>	**lower, xdigit, alpha, print, graph**
<g>	<G>	**lower, alpha, print, graph**
<h>	<H>	**lower, alpha, print, graph**
<i>	<I>	**lower, alpha, print, graph**
<j>	<J>	**lower, alpha, print, graph**
<k>	<K>	**lower, alpha, print, graph**
<l>	<L>	**lower, alpha, print, graph**
<m>	<M>	**lower, alpha, print, graph**
<n>	<N>	**lower, alpha, print, graph**
<o>	<O>	**lower, alpha, print, graph**
<p>	<P>	**lower, alpha, print, graph**
<q>	<Q>	**lower, alpha, print, graph**
<r>	<R>	**lower, alpha, print, graph**
<s>	<S>	**lower, alpha, print, graph**
<t>	<T>	**lower, alpha, print, graph**
<u>	<U>	**lower, alpha, print, graph**
<v>	<V>	**lower, alpha, print, graph**
<w>	<W>	**lower, alpha, print, graph**
<x>	<X>	**lower, alpha, print, graph**
<y>	<Y>	**lower, alpha, print, graph**
<z>	<Z>	**lower, alpha, print, graph**
<left-curly-bracket>		**punct, print, graph**
<vertical-line>		**punct, print, graph**
<right-curly-bracket>		**punct, print, graph**
<tilde>		**punct, print, graph**
<DEL>		**cntrl**

5.3.2 LC_COLLATE

The LC_COLLATE category provides a collation sequence definition for numerous utilities in the **XCU** specification (*sort*, *uniq*, and so forth), regular expression matching (see Chapter 7 on page 101) and the *strcoll*( ), *strxfrm*( ), *wcscoll*( ) and *wcsxfrm*( ) functions in the **XSH** specification.

A collation sequence definition defines the relative order between collating elements (characters and multi-character collating elements) in the locale. This order is expressed in terms of collation values; that is, by assigning each element one or more collation values (also known as collation weights). This does not imply that implementations assign such values, but that ordering of strings using the resultant collation definition in the locale will behave as if such assignment is done and used in the collation process. At least the following capabilities are provided:

1. **Multi-character collating elements**. Specification of multi-character collating elements (that is, sequences of two or more characters to be collated as an entity).
2. **User-defined ordering of collating elements**. Each collating element is assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching and, unless collation weights are explicitly specified, also as the collation weight to be used in sorting.
3. **Multiple weights and equivalence classes**. Collating elements can be assigned one or more (up to the limit {COLL_WEIGHTS_MAX}) collating weights for use in sorting. The first weight is hereafter referred to as the primary weight.
4. **One-to-Many mapping**. A single character is mapped into a string of collating elements.
5. **Equivalence class definition**. Two or more collating elements have the same collation value (primary weight).
6. **Ordering by weights**. When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements; the elements in each successive pair of elements are then compared according to the relative primary weights for the elements. If equal, and more than one weight has been assigned, then the pairs of collating elements are recompared according to the relative subsequent weights, until either a pair of collating elements compare unequal or the weights are exhausted.

The following keywords are recognised in a collation sequence definition. They are described in detail in the following sections.

collating-element Define a collating-element symbol representing a multi-character collating element. This keyword is optional.

collating-symbol Define a collating symbol for use in collation order statements. This keyword is optional.

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

order_end Specify the end of the collation-order statements.

copy Specify the name of an existing locale to be used as the definition of this category. If this keyword is specified, no other keyword can be specified.

The collating-element Keyword

In addition to the collating elements in the character set, the **collating–element** keyword is used to define multi-character collating elements. The syntax is:

`"collating-element %s from \"%s\"\n"`, <*collating-symbol*>, <*string*>

The <*collating-symbol*> operand is a symbolic name, enclosed between angle brackets (< and >), and must not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collates as an entity. A <*collating-element*> defined via this keyword is only recognised with the LC_COLLATE category.

Example:

```
collating-element <ch> from "<c><h>"
collating-element <e-acute> from "<acute><e>"
collating-element <ll> from "ll"
```

The collating-symbol Keyword

This keyword will be used to define symbols for use in collation sequence statements; that is, between the **order_start** and the **order_end** keywords. The syntax is:

`"collating-symbol %s\n"`, <*collating-symbol*>

The <*collating-symbol*> is a symbolic name, enclosed between angle brackets (< and >), and must not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. A <*collating-symbol*> defined via this keyword is only recognised with the LC_COLLATE category.

Example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

The **collating–symbol** keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight.

The order_start Keyword

The **order_start** keyword must precede 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 %s;%s;...;%s\n"`, <*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 define 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 will issue a warning message. The following directives will be supported:

forward Specifies that comparison operations for the weight level proceed from start of string towards the end of string.

backward Specifies that comparison operations for the weight level proceed from end of string towards the beginning of string.

position Specifies that comparison operations for the weight level will consider the relative position of elements in the strings not subject to **IGNORE**. The string containing an element not subject to **IGNORE** after the fewest collating elements subject to **IGNORE** from the start of the compare will collate first. If both strings contain a character not subject to **IGNORE** in the same relative position, the collating values assigned to the elements will determine the ordering. In case of equality, subsequent characters not subject to **IGNORE** are considered in the same manner.

The directives **forward** and **backward** are mutually exclusive.

Example:

```
order_start    forward;backward
```

If no operands are specified, a single **forward** operand is assumed.

The character (and collating element) order is defined by the order in which characters and elements are specified between the **order_start** and **order_end** keywords. This character order is used in range expressions in regular expressions (see Chapter 7). Weights assigned to the characters and elements define the collation sequence; in the absence of weights, the character order is also the collation sequence.

The **position** keyword provides the capability to consider, in a compare, the relative position of characters not subject to **IGNORE**. As an example, consider the two strings ‘‘o-ring’’ and ‘‘or-ing’’. Assuming the hyphen is subject to **IGNORE** on the first pass, the two strings will compare equal, and the position of the hyphen is immaterial. On second pass, all characters except the hyphen are subject to **IGNORE**, and in the normal case the two strings would again compare equal. By taking position into account, the first collates before the second.

Collation Order

The **order_start** keyword is followed by collating identifier entries. The syntax for the collating element entries is:

`"%s %s;%s;...;%s\n"`, <*`collating-identifier`*>, <*`weight`*>, <*`weight`*>, . . .

Each *collating-identifier* consists of either a character (in any of the forms defined in Section 5.3 on page 50), a <*collating-element*>, a <*collating-symbol*>, an ellipsis or the special symbol **UNDEFINED**. The order in which collating elements are specified determines the character order sequence, such that each collating element compares less than the elements following it. The NUL character compares lower than any other character.

A <*collating-element*> is used to specify multi-character collating elements, and indicates that the character sequence specified via the <*collating-element*> is to be collated as a unit and in the relative order specified by its place.

A <*collating-symbol*> is used to define a position in the relative order for use in weights. No weights are specified with a <*collating-symbol*>.

The ellipsis symbol specifies that a sequence of characters will collate according to their encoded character values. It is interpreted as indicating that 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, in the current coded character set, will be 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 preceding line specified the NUL character, and a trailing ellipsis as if the following line specified the highest coded character set value in the current coded character set. An ellipsis is treated as invalid if the preceding or following lines do not specify characters in the current coded character set. The use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable between implementations.

The symbol **UNDEFINED** is interpreted as including all coded character set values not specified explicitly or via the ellipsis symbol. Such characters are inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their coded character set values. If no **UNDEFINED** symbol is specified, and the current coded character set contains characters not specified in this section, the utility will issue a warning message and place such characters at the end of the character collation order.

The optional operands for each collation-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 collation-elements can be assigned the same weight; they belong to the same *equivalence class* if they have the same primary weight. Collation behaves as if, for each weight level, elements subject to **IGNORE** are removed, unless the **position** collation directive is specified for the corresponding level with the **order_start** keyword. Then each successive pair of elements is compared according to the relative weights for the elements. If the two strings compare equal, the process is repeated for the next weight level, up to the limit {COLL_WEIGHTS_MAX}.

Weights are expressed as characters (in any of the forms specified in Section 5.3 on page 50), <*collating-symbol*>s, <*collating-element*>s, an ellipsis, or the special symbol **IGNORE**. A single character, a <*collating-symbol*> or a <*collating-element*> represent the relative position in the character collating sequence of the character or symbol, rather than the character or characters themselves. Thus, rather than assigning absolute values to weights, a particular weight is expressed using the relative order value assigned to a collating element based on its order in the character collation sequence.

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the character <eszet> is given the string "<s><s>" as a weight, comparisons are performed as if all occurrences of the character <eszet> are replaced by <s><s> (assuming that <s> has the collating weight <s>). If it is necessary to define <eszet> and <s><s> as an equivalence class, then a collating element must be defined for the string ss.

All characters specified via an ellipsis will by default be assigned unique weights, equal to the relative order of characters. Characters specified via an explicit or implicit **UNDEFINED** special symbol will by default be assigned the same primary weight (that is, belong to the same equivalence class). An ellipsis symbol as a weight is interpreted to mean that each character in the sequence has unique weights, equal to the relative order of their character in the character collation sequence. 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**.

The special keyword **IGNORE** as a weight indicates that when strings are compared using the weights at the level where **IGNORE** is specified, the collating element is ignored; that is, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are subject to **IGNORE** in their primary weight form an equivalence class.

An empty operand is interpreted as the collating element itself.

For example, the order statement:

```
<a>     <a>;<a>
```

is equal to:

```
<a>
```

An ellipsis can be used as an operand if the collating element was an ellipsis, and is interpreted as the value of each character defined by the ellipsis.

The collation order as defined in this section defines the interpretation of bracket expressions in regular expressions (see Section 7.3.5 on page 105).

Example:

```
order_start   forward;backward
UNDEFINED     IGNORE;IGNORE
<LOW>
<space>       <LOW>;<space>
...           <LOW>;...
<a>           <a>;<a>
<a-acute>     <a>;<a-acute>
<a-grave>     <a>;<a-grave>
<A>           <a>;<A>
<A-acute>     <a>;<A-acute>
<A-grave>     <a>;<A-grave>
<ch>          <ch>;<ch>
<Ch>          <ch>;<Ch>
<s>           <s>;<s>
<eszet>       "<s><s>";"<eszet><eszet>"
order_end
```

This example is interpreted as follows:

1. The **UNDEFINED** means that all characters not specified in this definition (explicitly or via the ellipsis) are ignored for collation purposes; for regular expression purposes they are ordered first.

2. All characters between <space> and <a> have the same primary equivalence class and individual secondary weights based on their ordinal encoded values.

3. All characters based on the upper- or lower-case character a belong to the same primary equivalence class.

4. The multi-character collating element <ch> is represented by the collating symbol <ch> and belongs to the same primary equivalence class as the multi-character collating element <Ch>.

The order_end Keyword

The collating order entries must be terminated with an **order_end** keyword.

The collation sequence definition of the POSIX locale follows; the code listing depicts the *localedef* input.

```
LC_COLLATE
# This is the POSIX locale definition for the LC_COLLATE category.
# The order is the same as in the ASCII codeset.
order_start forward
<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
<CAN>
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen>
```

```
<period>
<slash>
<zero>
<one>
<two>
<three>
<four>
<five>
<six>
<seven>
<eight>
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<A>
<B>
<C>
<D>
<E>
<F>
<G>
<H>
<I>
<J>
<K>
<L>
<M>
<N>
<O>
<P>
<Q>
<R>
<S>
<T>
<U>
<V>
<W>
<X>
<Y>
<Z>
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a>
```

```
<b>
<c>
<d>
<e>
<f>
<g>
<h>
<i>
<j>
<k>
<l>
<m>
<n>
<o>
<p>
<q>
<r>
<s>
<t>
<u>
<v>
<w>
<x>
<y>
<z>
<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
<DEL>
order_end
#
END LC_COLLATE
```

5.3.3 LC_MONETARY

EX The LC_MONETARY category defines the rules and symbols that are used to format monetary numeric information. This information is available through the *localeconv*( ) function and is used by the *strfmon*( ) function.

EX Some of the information is also available in an alternative form via the *nl_langinfo*( ) function (see CRNCYSTR in **<langinfo.h>**).

The following items are defined in this category of the locale. The item names are the keywords recognised by the *localedef* utility when defining a locale. They are also similar to the member names of the **lconv** structure defined in **<locale.h>**; see the **XSH** specification for the exact symbols in the header. The *localeconv*( ) function returns {CHAR_MAX} for unspecified integer items and the empty string ("") for unspecified or size zero string items.

In a locale definition file, the operands are strings, formatted as indicated by the grammar in Section 5.4 on page 82. For some keywords, the strings can contain only integers. Keywords that are not provided, string values set to the empty string (""), or integer keywords set to –1, are used to indicate that the value is not available in the locale.

int_curr_symbol The international currency symbol. The operand is a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in the ISO 4217: 1987 standard. The fourth character is the character used to separate the international currency symbol from the monetary quantity.

currency_symbol The string used as the local currency symbol.

mon_decimal_point The operand is a string containing the symbol that is used as the decimal delimiter (radix character) in monetary formatted quantities. In contexts where standards (such as the ISO C standard) limit the **mon_decimal_point** to a single byte, the result of specifying a multi-byte operand is unspecified.

mon_thousands_sep The operand is a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. In contexts where standards limit the **mon_thousands_sep** to a single byte, the result of specifying a multi-byte operand is unspecified.

mon_grouping 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. If the last integer is not −1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is −1, then no further grouping will be performed.

The following is an example of the interpretation of the **mon_grouping** keyword. Assuming that the value to be formatted is 123456789 and the **mon_thousands_sep** is ’, then the following table shows the result. The third column shows the equivalent string in the ISO C standard that would be used by the *localeconv*( ) function to accommodate this grouping.

mon_grouping	Formatted Value	ISO C String
3;−1	123456’789	"\3\177"
3	123’456’789	"\3"
3;2;−1	1234’56’789	"\3\2\177"
3;2	12’34’56’789	"\3\2"
−1	123456789	"\177"

In these examples, the octal value of {CHAR_MAX} is 177.

positive_sign A string used to indicate a non-negative-valued formatted monetary quantity.

negative_sign A string used to indicate a negative-valued formatted monetary quantity.

int_frac_digits An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using **int_curr_symbol**.

frac_digits
An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using **currency_symbol**.

p_cs_precedes
An integer set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a monetary quantity with a non-negative value, and set to 0 if the symbol succeeds the value.

p_sep_by_space
An integer set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a monetary quantity with a non-negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

n_cs_precedes
An integer set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a monetary quantity with a negative value, and set to 0 if the symbol succeeds the value.

n_sep_by_space
An integer set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a monetary quantity with a negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

p_sign_posn
An integer set to a value indicating the positioning of the **positive_sign** for a monetary quantity with a non-negative value. The following integer values are recognised for both **p_sign_posn** and **n_sign_posn**:

0 Parentheses enclose the quantity and the **currency_symbol** or **int_curr_symbol**.

1 The sign string precedes the quantity and the **currency_symbol** or **int_curr_symbol**.

2 The sign string succeeds the quantity and the **currency_symbol** or **int_curr_symbol**.

3 The sign string precedes the **currency_symbol** or **int_curr_symbol**.

4 The sign string succeeds the **currency_symbol** or **int_curr_symbol**.

n_sign_posn
An integer set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

copy
Note: This is a *localedef* utility keyword, unavailable through *localeconv*( ).

Specify the name of an existing locale to be used as the definition of this category. If this keyword is specified, no other keyword can be specified.

The following table shows the result of various combinations:

			p_sep_by_space	
		2	**1**	**0**
p_cs_precedes = 1	**p_sign_posn** = 0	($1.25)	($ 1.25)	($1.25)
	p_sign_posn = 1	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 2	$1.25 +	$ 1.25+	$1.25+
	p_sign_posn = 3	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 4	$ +1.25	$+ 1.25	$+1.25
p_cs_precedes = 0	**p_sign_posn** = 0	(1.25 $)	(1.25 $)	(1.25$)
	p_sign_posn = 1	+1.25 $	+1.25 $	+1.25$
	p_sign_posn = 2	1.25$ +	1.25 $+	1.25$+
	p_sign_posn = 3	1.25+ $	1.25 +$	1.25+$
	p_sign_posn = 4	1.25$ +	1.25 $+	1.25$+

The monetary formatting definitions for the POSIX locale follow; the code listing depicting the EX *localedef* input, the table representing the same information with the addition of *localeconv*( ) and *nl_langinfo*( ) formats. All values are unspecified in the POSIX locale.

```
LC_MONETARY
# This is the POSIX locale definition for
# the LC_MONETARY category.
#
int_curr_symbol      ""
currency_symbol      ""
mon_decimal_point    ""
mon_thousands_sep    ""
mon_grouping         -1
positive_sign        ""
negative_sign        ""
int_frac_digits      -1
p_cs_precedes        -1
p_sep_by_space       -1
n_cs_precedes        -1
n_sep_by_space       -1
p_sign_posn          -1
n_sign_posn          -1
#
END LC_MONETARY
```

Item	POSIX locale Value	langinfo Constant	*localeconv*( ) Value	localedef Value
currency_symbol	n/a	CRNCYSTR	""	""
frac_digits	n/a	-	CHAR_MAX	–1
int_curr_symbol	n/a	-	""	""
int_frac_digits	n/a	-	CHAR_MAX	–1
mon_decimal_point	n/a	-	""	""
mon_thousands_sep	n/a	-	""	""
mon_grouping	n/a	-	""	""
positive_sign	n/a	-	""	""
negative_sign	n/a	-	""	""
p_cs_precedes	n/a	CRNCYSTR	CHAR_MAX	–1
n_cs_precedes	n/a	CRNCYSTR	CHAR_MAX	–1
p_sep_by_space	n/a	-	CHAR_MAX	–1
n_sep_by_space	n/a	-	CHAR_MAX	–1
p_sign_posn	n/a	-	CHAR_MAX	–1
n_sign_posn	n/a	-	CHAR_MAX	–1

EX In the preceding table, the **langinfo Constant** column represents an X/Open extension. The entry **n/a** indicates that the value is not available in the POSIX locale.

5.3.4 LC_NUMERIC

The LC_NUMERIC category defines the rules and symbols that will be used to format non-monetary numeric information. This information is available through the *localeconv*( ) function. EX Some of the information is also available in an alternative form via the *nl_langinfo*( ) function.

The following items are defined in this category of the locale. The item names are the keywords recognised by the *localedef* utility when defining a locale. They are also similar to the member names of the *lconv* structure defined in **<locale.h>**; see the **XSH** specification for the exact symbols in the header. The *localeconv*( ) function returns {CHAR_MAX} for unspecified integer items and the empty string ("") for unspecified or size zero string items.

In a locale definition file, the operands are strings, formatted as indicated by the grammar in Section 5.4 on page 82. For some keywords, the strings only can contain integers. Keywords that are not provided, string values set to the empty string (""), or integer keywords set to −1, will be used to indicate that the value is not available in the locale. The following keywords are recognised:

decimal_point The operand is a string containing the symbol that is used as the decimal delimiter (radix character) in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string. In contexts where standards limit the **decimal_point** to a single byte, the result of specifying a multi-byte operand is unspecified.

thousands_sep The operand is a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary formatted monetary quantities. In contexts where standards limit the **thousands_sep** to a single byte, the result of specifying a multi-byte operand is unspecified.

grouping Define the size of each group of digits in formatted non-monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not −1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is −1, then no further grouping will be performed.

copy **Note:** This is a *localedef* utility keyword, unavailable through *localeconv*( ).

Specify the name of an existing locale to be used as the definition of this category. If this keyword is specified, no other keyword can be specified.

The non-monetary numeric formatting definitions for the POSIX locale follow; the code listing depicting the *localedef* input, the table representing the same information with the addition of EX *localeconv*( ) values and *nl_langinfo*( ) constants.

```
LC_NUMERIC
# This is the POSIX locale definition for
# the LC_NUMERIC category.
#
decimal_point    "<period>"
thousands_sep    ""
grouping         -1
#
END LC_NUMERIC
```

Item	POSIX locale Value	langinfo Constant	*localeconv*( ) Value	localedef Value
decimal_point	"."	RADIXCHAR	"."	.
thousands_sep	n/a	THOUSEP	""	""
grouping	n/a	-	""	−1

EX In the preceding table, the **langinfo Constant** column represents an X/Open extension. The entry **n/a** indicates that the value is not available in the POSIX locale.

5.3.5 LC_TIME

The LC_TIME category defines the interpretation of the field descriptors supported by the *date* EX utility and affects the behaviour of the *strftime*( ), *wcsftime*( ), *strptime*( ) and *nl_langinfo*( ) functions. Because the interfaces for C-language access and locale definition differ significantly, they are described separately.

LC_TIME Locale Definition

For locale definition, the following mandatory keywords are recognised:

abday Define the abbreviated weekday names, corresponding to the %a field descriptor (conversion specification in the *strftime*( ), *wcsftime*( ) and *strptime*( ) functions). The operand consists of seven semicolon-separated strings, each surrounded by double-quotes. The first string is the abbreviated name of the day corresponding to Sunday, the second the abbreviated name of the day corresponding to Monday, and so on.

day Define the full weekday names, corresponding to the %A field descriptor. The operand consists of seven semicolon-separated strings, each surrounded by double-quotes. The first string is the full name of the day corresponding to Sunday, the second the full name of the day corresponding to Monday, and so on.

abmon Define the abbreviated month names, corresponding to the %b field descriptor. The operand consists of twelve semicolon-separated strings, each surrounded by double-quotes. The first string is the abbreviated name of the first month of the year (January), the second the abbreviated name of the second month, and so on.

mon Define the full month names, corresponding to the %B field descriptor. The operand consists of twelve semicolon-separated strings, each surrounded by double-quotes. The first string is the full name of the first month of the year (January), the second the full name of the second month, and so on.

d_t_fmt Define the appropriate date and time representation, corresponding to the %c field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain escape sequences defined in the table in Table 3-1 on page 40 (\\, **a**, **b**, **f**, **n**, **r**, **t**, **v**).

d_fmt Define the appropriate date representation, corresponding to the %x field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain escape sequences defined in the table in Table 3-1 on page 40.

t_fmt Define the appropriate time representation, corresponding to the %X field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain escape sequences defined in the table in Table 3-1 on page 40.

am_pm Define the appropriate representation of the *ante meridiem* and *post meridiem* strings, corresponding to the %p field descriptor. The operand consists of two strings, separated by a semicolon, each surrounded by double-quotes. The first string represents the *ante meridiem* designation, the last string the *post meridiem* designation.

t_fmt_ampm Define the appropriate time representation in the 12-hour clock format with **am_pm**, corresponding to the %r field descriptor. The operand consists of a string and can contain any combination of characters and field descriptors. If the string is empty, the 12-hour format is not supported in the locale.

EX **era** Define how years are counted and displayed for each era in a locale. The operand consists of semicolon-separated strings. Each string is an era description segment with the format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

according to the definitions below. There can be as many era description segments as are necessary to describe the different eras.

Note: The start of an era might not be the earliest point in the era it may be the latest. For example, the Christian era BC starts on the day before January 1, AD 1, and increases with earlier time.

direction Either a + or a – character. The + character indicates that years closer to the *start_date* have lower numbers than those closer to the *end_date*. The – character indicates that years closer to the *start_date* have higher numbers than those closer to the *end_date*.

offset The number of the year closest to the *start_date* in the era, corresponding to the %Ey field descriptor.

start_date A date in the form *yyyy* /*mm* /*dd*, where *yyyy*, *mm* and *dd* are the year, month and day numbers respectively of the start of the era. Years prior to AD 1 are represented as negative numbers.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values –* or +*. The value –* indicates that the ending date is the beginning of time. The value +* indicates that the ending date is the end of time.

era_name A string representing the name of the era, corresponding to the %EC field descriptor.

era_format A string for formatting the year in the era, corresponding to the %EY field descriptor.

era_d_fmt Define the format of the date in alternative era notation, corresponding to the %Ex field descriptor.

era_t_fmt Define the locale's appropriate alternative time format, corresponding to the %EX field descriptor.

era_d_t_fmt Define the locale's appropriate alternative date and time format, corresponding to the %Ec field descriptor.

alt_digits Define alternative symbols for digits, corresponding to the %O field descriptor modifier. The operand consists of semicolon-separated strings, each surrounded by double-quotes. The first string is the alternative symbol corresponding with zero, the second string the symbol corresponding with one, and so on. Up to 100 alternative symbol strings can be specified. The %O modifier indicates that the string corresponding to the value specified via the field descriptor will be used instead of the value.

copy Specify the name of an existing locale to be used as the definition of this category. If this keyword is specified, no other keyword can be specified.

LC_TIME C-language Access

EX The following information can be accessed. These correspond to constants defined in **<langinfo.h>** and used as arguments to the *nl_langinfo*( ) function.

ABDAY_*x* The abbreviated weekday names (for example Sun), where *x* is a number from 1 to 7.

DAY_*x* The full weekday names (for example Sunday), where *x* is a number from 1 to 7.

ABMON_*x* The abbreviated month names (for example Jan), where *x* is a number from 1 to 12.

MON_*x* The full month names (for example January), where *x* is a number from 1 to 12.

D_T_FMT The appropriate date and time representation.

D_FMT The appropriate date representation.

T_FMT The appropriate time representation.

AM_STR The appropriate ante-meridiem affix.

PM_STR The appropriate post-meridiem affix.

T_FMT_AMPM The appropriate time representation in the 12-hour clock format with AM_STR and PM_STR.

ERA The era description segments, which describe how years are counted and displayed for each era in a locale. Each era description segment has the format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

according to the definitions below. There will be as many era description segments as are necessary to describe the different eras. Era description segments are separated by semicolons.

Note: The start of an era might not be the earliest point in the era it may be the latest. For example, the Christian era BC starts on the day before January 1, AD 1, and increases with earlier time.

direction Either a + or a – character. The + character indicates that years closer to the *start_date* have lower numbers than those closer to the *end_date*. The – character indicates that years closer to the *start_date* have higher numbers than those closer to the *end_date*.

offset The number of the year closest to the *start_date* in the era.

start_date A date in the form *yyyy* /*mm* /*dd*, where *yyyy*, *mm* and *dd* are the year, month and day numbers respectively of the start of the era. Years prior to AD 1 are represented as negative numbers.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values –* or +*. The value –* indicates that the ending date is the beginning of time. The value +* indicates that the ending date is the end of time.

era_name The era, corresponding to the %EC conversion specification.

era_format The format of the year in the era, corresponding to the %EY conversion specification.

ERA_D_FMT The era date format.

EX ERA_T_FMT The locale’s appropriate alternative time format, corresponding to the %EX field descriptor.

ERA_D_T_FMT The locale’s appropriate alternative date and time format, corresponding to the %Ec field descriptor.

ALT_DIGITS The alternative symbols for digits, corresponding to the %O conversion specification modifier. The value consists of semicolon-separated symbols. The first is the alternative symbol corresponding to zero, the second is the symbol corresponding to one, and so on. Up to 100 alternative symbols may be specified.

The following table displays the correspondence between the items described above and the conversion specifiers used by the *date* utility and the *strftime*( ), *wcsftime*( ) and *strptime*( ) functions.

	localedef Keyword	langinfo Constant	Conversion Specifier
	abday	ABDAY_*x*	%a
	day	DAY_*x*	%A
	abmon	ABMON_*x*	%b
	mon	MON	%B
	d_t_fmt	D_T_FMT	%c
	d_fmt	D_FMT	%x
	t_fmt	T_FMT	%X
	am_pm	AM_STR	%p
	am_pm	PM_STR	%p
	t_fmt_ampm	T_FMT_AMPM	%r
EX	**era**	ERA	%EC, %Ey, %EY
EX	**era_d_fmt**	ERA_D_FMT	%Ex
EX	**era_t_fmt**	ERA_T_FMT	%EX
EX	**era_d_t_fmt**	ERA_D_T_FMT	%Ec
EX	**alt_digits**	ALT_DIGITS	%O

EX In the preceding table, the **langinfo Constant** column represents an X/Open extension.

LC_TIME General Information

Although certain of the field descriptors in the POSIX locale (such as the name of the month) are shown with initial capital letters, this need not be the case in other locales. Programs using these fields may need to adjust the capitalisation if the output is going to be used at the beginning of a sentence.

The LC_TIME descriptions of **abday**, **day**, **mon** and **abmon** imply a Gregorian style calendar (7-day weeks, 12-month years, leap years, and so forth). Formatting time strings for other types of calendars is outside the scope of this specification set.

As specified under *date* in the Locale Definition and *strftime*( ), in the **XSH** specification, the field descriptors corresponding to the optional keywords consist of a modifier followed by a traditional field descriptor (for instance %Ex). If the optional keywords are not supported by the implementation or are unspecified for the current locale, these field descriptors are treated as the traditional field descriptor. For instance, assume the following keywords:

```
alt_digits    "0th";"1st";"2nd";"3rd";"4th";"5th";\
              "6th";"7th";"8th";"9th";"10th"

d_fmt        "The %Od day of %B in %Y"
```

On 7/4/1776, the %x field descriptor would result in ‘‘The 4th day of July in 1776’’, while 7/14/1789 would come out as ‘‘The 14 day of July in 1789’’. It can be noted that the above example is for illustrative purposes only; the %O modifier is primarily intended to provide for Kanji or Hindi digits in *date* formats.

EX The following is an example for Japan that supports the current plus last three Emperors and reverts to Western style numbering for years prior to the Meiji era. The example also allows for the custom of using a special name for the first year of an era instead of using 1. (The examples substitute romaji where kanji should be used.)

```
   era_d_fmt "%EY%mgatsu%dnichi (%a)"
```

```
   era    "+:2:1990/01/01:+*:Heisei:%EC%Eynen";\
          "+:1:1989/01/08:1989/12/31:Heisei:%ECgannen";\
          "+:2:1927/01/01:1989/01/07:Shouwa:%EC%Eynen";\
          "+:1:1926/12/25:1926/12/31:Shouwa:%ECgannen";\
          "+:2:1913/01/01:1926/12/24:Taishou:%EC%Eynen";\
          "+:1:1912/07/30:1912/12/31:Taishou:%ECgannen";\
          "+:2:1869/01/01:1912/07/29:Meiji:%EC%Eynen";\
          "+:1:1868/09/08:1868/12/31:Meiji:%ECgannen";\
          "-:1868:1868/09/07:-*::%Ey"
```

Assuming that the current date is September 21, 1991, a request to *date* or *strftime*( ) would yield the following results:

```
   %Ec - Heisei3nen9gatsu21nichi (Sat) 14:39:26
   %EC - Heisei
   %Ex - Heisei3nen9gatsu21nichi (Sat)
   %Ey - 3
   %EY - Heisei3nen
```

Example era definitions for the Republic of China:

```
   era    "+:2:1913/01/01:+*:ChungHwaMingGuo:%EC%EyNen";\
          "+:1:1912/1/1:1912/12/31:ChungHwaMingGuo:%ECYuenNen";\
          "+:1:1911/12/31:-*:MingChien:%EC%EyNen"
```

Example definitions for the Christian Era:

```
   era    "+:0:0000/01/01:+*:AD:%EC %Ey";\
          "+:1:-0001/12/31:-*:BC:%Ey %EC"
```

The LC_TIME category definition of the POSIX locale follows; the code listing depicts the
EX *localedef* input; the table depicts the *langinfo* items defined in this category.

```
LC_TIME
# This is the POSIX locale definition for
# the LC_TIME category.
#
# Abbreviated weekday names (%a)
abday      "<S><u><n>";"<M><o><n>";"<T><u><e>";"<W><e><d>";\
           "<T><h><u>";"<F><r><i>";"<S><a><t>"
#
# Full weekday names (%A)
day        "<S><u><n><d><a><y>";"<M><o><n><d><a><y>";\
           "<T><u><e><s><d><a><y>";"<W><e><d><n><e><s><d><a><y>";\
           "<T><h><u><r><s><d><a><y>";"<F><r><i><d><a><y>";\
           "<S><a><t><u><r><d><a><y>"
#
```

```
# Abbreviated month names (%b)
abmon      "<J><a><n>";"<F><e><b>";"<M><a><r>";\
           "<A><p><r>";"<M><a><y>";"<J><u><n>";\
           "<J><u><l>";"<A><u><g>";"<S><e><p>";\
           "<O><c><t>";"<N><o><v>";"<D><e><c>"
#
# Full month names (%B)
mon        "<J><a><n><u><a><r><y>";"<F><e><b><r><u><a><r><y>";\
           "<M><a><r><c><h>";"<A><p><r><i><l>";\
           "<M><a><y>";"<J><u><n><e>";\
           "<J><u><l><y>";"<A><u><g><u><s><t>";\
           "<S><e><p><t><e><m><b><e><r>";"<O><c><t><o><b><e><r>";\
           "<N><o><v><e><m><b><e><r>";"<D><e><c><e><m><b><e><r>"
#
# Equivalent of AM/PM (%p)      "AM";"PM"
am_pm      "<A><M>";"<P><M>"
#
# Appropriate date and time representation (%c)
#    "%a %b %e %H:%M:%S %Y"
d_t_fmt    "<percent-sign><a><space><percent-sign><b>\
            <space><percent-sign><e><space><percent-sign><H>\
            <colon><percent-sign><M><colon><percent-sign><S>\
            <space><percent-sign><Y>"
#
# Appropriate date representation (%x)   "%m/%d/%y"
d_fmt      "<percent-sign><m><slash><percent-sign><d>\
            <slash><percent-sign><y>"
#
# Appropriate time representation (%X)   "%H:%M:%S"
t_fmt      "<percent-sign><H><colon><percent-sign><M>\
            <colon><percent-sign><S>"
#
# Appropriate 12-hour time representation (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign><M><colon>\
            <percent-sign><S> <percent_sign><p>"
#
END LC_TIME
```

EX

Item	POSIX Locale Value	Item	POSIX Locale Value
D_T_FMT	"%a %b %e %H:%M:%S %Y"	**MON_3**	"March"
D_FMT	"%m / %d / %y"	**MON_4**	"April"
T_FMT	"%H:%M:%S"	**MON_5**	"May"
AM_STR	"AM"	**MON_6**	"June"
PM_STR	"PM"	**MON_7**	"July"
T_FMT_AMPM	"%I:%M:%S %p"	**MON_8**	"August"
DAY_1	"Sunday"	**MON_9**	"September"
DAY_2	"Monday"	**MON_10**	"October"
DAY_3	"Tuesday"	**MON_11**	"November"
DAY_4	"Wednesday"	**MON_12**	"December"
DAY_5	"Thursday"	**ABMON_1**	"Jan"
DAY_6	"Friday"	**ABMON_2**	"Feb"
DAY_7	"Saturday"	**ABMON_3**	"Mar"
ABDAY_1	"Sun"	**ABMON_4**	"Apr"
ABDAY_2	"Mon"	**ABMON_5**	"May"
ABDAY_3	"Tue"	**ABMON_6**	"Jun"
ABDAY_4	"Wed"	**ABMON_7**	"Jul"
ABDAY_5	"Thu"	**ABMON_8**	"Aug"
ABDAY_6	"Fri"	**ABMON_9**	"Sep"
ABDAY_7	"Sat"	**ABMON_10**	"Oct"
MON_1	"January"	**ABMON_11**	"Nov"
MON_2	"February"	**ABMON_12**	"Dec"

5.3.6 LC_MESSAGES

The LC_MESSAGES category defines the format and values for affirmative and negative responses.

EX The message catalogue used by the standard utilities and selected by the *catopen*( ) function is determined by the setting of *NLSPATH*; see Chapter 6 on page 93. The LC_MESSAGES category can be specified as part of an *NLSPATH* substitution field.

EX The following keywords are recognised as part of the locale definition file. The *nl_langinfo*( ) function accepts upper-case versions of the first four keywords.

yesexpr The operand consists of an extended regular expression (see Section 7.4 on page 109) that describes the acceptable affirmative response to a question expecting an affirmative or negative response.

noexpr The operand consists of an extended regular expression that describes the acceptable negative response to a question expecting an affirmative or negative response.

EX **yesstr** (**LEGACY**)
The operand consists of a fixed string (not a regular expression) that can be used by an application for composition of a message that lists an acceptable affirmative response, such as in a prompt.

EX **nostr** (**LEGACY**)
The operand consists of a fixed string that can be used by an application for composition of a message that lists an acceptable negative response.

copy Specify the name of an existing locale to be used as the definition of this category. If this keyword is specified, no other keyword can be specified.

Note that the **yesstr** and **nostr** values have different uses from those in Issue 3.

EX The format and values for affirmative and negative responses of the POSIX locale follow; the code listing depicting the *localedef* input, the table representing the same information with the addition of *nl_langinfo*( ) constants.

```
LC_MESSAGES
# This is the POSIX locale definition for
# the LC_MESSAGES category.
#
yesexpr  "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
#
noexpr   "<circumflex><left-square-bracket><n><N><right-square-bracket>"
#
yesstr   "yes"
nostr    "no"
END LC_MESSAGES
```

EX

localedef Keyword	**langinfo Constant**	**POSIX Locale Value**
yesexpr	YESEXPR	"^[yY]"
noexpr	NOEXPR	"^[nN]"
EX **yesstr**	YESSTR	"yes" (**LEGACY**)
EX **nostr**	NOSTR	"no" (**LEGACY**)

LC_MESSAGES Application Usage

EX The **yesstr** and **nostr** locale keywords and the YESSTR and NOSTR *langinfo* items formerly were used to match user affirmative and negative responses. In this issue, the **yesexpr**, **noexpr**, YESEXPR and NOEXPR extended regular expressions have replaced them. However, they have been retained for backward compatibility to allow an application to include a sample desired response in a prompting message. They are marked **LEGACY**. Applications should use the general locale-based messaging facilities (see the **Internationalisation Guide**) to issue such prompting messages.

5.4 Locale Definition Grammar

The grammar and lexical conventions in this section together describe the syntax for the locale definition source. The general conventions for this style of grammar are described in the **XCU** specification, **Section 1.8**, **Grammar Conventions**. The grammar takes precedence over the text.

5.4.1 Locale Lexical Conventions

The lexical conventions for the locale definition grammar are described in this section.

The following tokens are processed (in addition to those string constants shown in the grammar):

LOC_NAME
A string of characters representing the name of a locale.

CHAR
Any single character.

NUMBER
A decimal number, represented by one or more decimal digits.

COLLSYMBOL
A symbolic name, enclosed between angle brackets. The string cannot duplicate any charmap symbol defined in the current charmap (if any), or a **COLLELEMENT** symbol.

COLLELEMENT
A symbolic name, enclosed between angle brackets, which cannot duplicate either any charmap symbol or a **COLLSYMBOL** symbol.

EX CHARCLASS
A string of alphanumeric characters from the portable character set, the first of which is not a digit, consisting of at least one and at most {CHARCLASS_NAME_MAX} bytes, and optionally surrounded by double-quotes.

CHARSYMBOL
A symbolic name, enclosed between angle brackets, from the current charmap (if any).

OCTAL_CHAR
One or more octal representations of the encoding of each byte in a single character. The octal representation consists of an escape character (normally a backslash) followed by two or more octal digits.

HEX_CHAR
One or more hexadecimal representations of the encoding of each byte in a single character. The hexadecimal representation consists of an escape character followed by the constant x and two or more hexadecimal digits.

DECIMAL_CHAR
One or more decimal representations of the encoding of each byte in a single character. The decimal representation consists of an escape character followed by a character d and two or more decimal digits.

ELLIPSIS
The string ...

EXTENDED_REG_EXP
An extended regular expression as defined in the grammar in Section 7.5 on page 112.

EOL
The line termination character newline.

5.4.2 Locale Grammar

This section presents the grammar for the locale definition.

```
%token              LOC_NAME
%token              CHAR
%token              NUMBER
%token              COLLSYMBOL COLLELEMENT
%token              CHARSYMBOL OCTAL_CHAR HEX_CHAR DECIMAL_CHAR
%token              ELLIPSIS
%token              EXTENDED_REG_EXP
%token              EOL

%start              locale_definition

%%

locale_definition   : global_statements locale_categories
                    |                   locale_categories
                    ;

global_statements   : global_statements symbol_redefine
                    | symbol_redefine
                    ;

symbol_redefine     : ’escape_char’  CHAR EOL
                    | ’comment_char’ CHAR EOL
                    ;
locale_categories   : locale_categories locale_category
                    | locale_category
                    ;

locale_category     : lc_ctype    | lc_collate | lc_messages
                    | lc_monetary | lc_numeric | lc_time
                    ;

/* The following grammar rules are common to all categories */

char_list           : char_list char_symbol
                    | char_symbol
                    ;

char_symbol         : CHAR    | CHARSYMBOL
                    | OCTAL_CHAR | HEX_CHAR | DECIMAL_CHAR
                    ;

elem_list           : elem_list char_symbol
                    | elem_list COLLSYMBOL
                    | elem_list COLLELEMENT
                    | char_symbol
                    | COLLSYMBOL
                    | COLLELEMENT
                    ;

symb_list           : symb_list COLLSYMBOL
                    | COLLSYMBOL
                    ;
```

```
       locale_name         : LOC_NAME
                           | ’"’ LOC_NAME ’"’
                           ;

       /* The following is the LC_CTYPE category grammar */

       lc_ctype            : ctype_hdr ctype_keywords         ctype_tlr
                           | ctype_hdr ’copy’ locale_name EOL ctype_tlr
                           ;

       ctype_hdr           : ’LC_CTYPE’ EOL
                           ;

       ctype_keywords      : ctype_keywords ctype_keyword
                           | ctype_keyword
                           ;

       ctype_keyword       : charclass_keyword charclass_list EOL
                           | charconv_keyword charconv_list EOL
EX                         | ’charclass’ charclass_namelist EOL
                           ;
       charclass_namelist  : charclass_namelist ’  ;’ CHARCLASS
                           | CHARCLASS
                           ;

       charclass_keyword   : ’upper’   | ’lower’ | ’alpha’ | ’digit’
                           | ’punct’ | ’xdigit’ | ’space’ | ’print’
                           | ’graph’ | ’blank’ | ’cntrl’
EX                         | CHARCLASS
                           ;
       charclass_list      : charclass_list ’  ;’ char_symbol
                           | charclass_list ’  ;’ ELLIPSIS ’;’ char_symbol
                           | char_symbol
                           ;

       charconv_keyword    : ’toupper’
                           | ’tolower’
                           ;

       charconv_list       : charconv_list ’  ;’ charconv_entry
                           | charconv_entry
                           ;
       charconv_entry      : ’(’ char_symbol ’,’ char_symbol ’)’
                           ;

       ctype_tlr           : ’END’ ’LC_CTYPE’ EOL
                           ;

       /* The following is the LC_COLLATE category grammar */

       lc_collate          : collate_hdr collate_keywords       collate_tlr
                           | collate_hdr ’copy’ locale_name EOL collate_tlr
                           ;

       collate_hdr         : ’LC_COLLATE’ EOL
                           ;
```

```
collate_keywords    :                order_statements
                    | opt_statements order_statements
                    ;
opt_statements      : opt_statements collating_symbols
                    | opt_statements collating_elements
                    | collating_symbols
                    | collating_elements
                    ;
collating_symbols   : ’collating-symbol’ COLLSYMBOL EOL
                    ;
collating_elements  : ’collating-element’ COLLELEMENT
                    ’from’ ’"’ elem_list ’"’ EOL
                    ;
order_statements    : order_start collation_order order_end
                    ;
order_start         : ’order_start’ EOL
                    | ’order_start’ order_opts EOL
                    ;
order_opts          : order_opts ’  ;’ order_opt
                    | order_opt
                    ;
order_opt           : order_opt ’,’ opt_word
                    | opt_word
                    ;
opt_word            : ’forward’   | ’backward’ | ’position’
                    ;
collation_order     : collation_order collation_entry
                    | collation_entry
                    ;
collation_entry     : COLLSYMBOL EOL
                    | collation_element weight_list EOL
                    | collation_element             EOL
                    ;
collation_element   : char_symbol
                    | COLLELEMENT
                    | ELLIPSIS
                    | ’UNDEFINED’
                    ;
weight_list         : weight_list ’  ;’ weight_symbol
                    | weight_list ’  ;’
                    | weight_symbol
                    ;
```

```
weight_symbol       : /* empty */
                    | char_symbol
                    | COLLSYMBOL
                    | ’"’ elem_list ’"’
                    | ’"’ symb_list ’"’
                    | ELLIPSIS
                    | ’IGNORE’
                    ;
order_end           : ’order_end’ EOL
                    ;
collate_tlr         : ’END’ ’LC_COLLATE’ EOL
                    ;
/* The following is the LC_MESSAGES category grammar */
lc_messages         : messages_hdr messages_keywords      messages_tlr
                    | messages_hdr ’copy’ locale_name EOL messages_tlr
                    ;
messages_hdr        : ’LC_MESSAGES’ EOL
                    ;
messages_keywords   : messages_keywords messages_keyword
                    | messages_keyword
                    ;
messages_keyword    : ’yesexpr’ ’"’ EXTENDED_REG_EXP ’"’ EOL
                    | ’noexpr’  ’"’ EXTENDED_REG_EXP ’"’ EOL
                    | ’yesstr’  ’"’ char_list ’"’ EOL
                    | ’nostr’   ’"’ char_list ’"’ EOL
                    ;
messages_tlr        : ’END’ ’LC_MESSAGES’ EOL
                    ;
/* The following is the LC_MONETARY category grammar */
lc_monetary         : monetary_hdr monetary_keywords       monetary_tlr
                    | monetary_hdr ’copy’ locale_name EOL  monetary_tlr
                    ;
monetary_hdr        : ’LC_MONETARY’ EOL
                    ;
monetary_keywords   : monetary_keywords monetary_keyword
                    | monetary_keyword
                    ;
monetary_keyword    : mon_keyword_string mon_string EOL
                    | mon_keyword_char NUMBER EOL
                    | mon_keyword_char ’-1’   EOL
                    | mon_keyword_grouping mon_group_list EOL
                    ;
```

```
mon_keyword_string  : ’int_curr_symbol’   | ’currency_symbol’
                    | ’mon_decimal_point’ | ’mon_thousands_sep’
                    | ’positive_sign’ | ’negative_sign’
                    ;
mon_string          : ’"’ char_list ’"’
                    | ’""’
                    ;
mon_keyword_char    : ’int_frac_digits’   | ’frac_digits’
                    | ’p_cs_precedes’ | ’p_sep_by_space’
                    | ’n_cs_precedes’ | ’n_sep_by_space’
                    | ’p_sign_posn’ | ’n_sign_posn’
                    ;
mon_keyword_grouping  : ’mon_grouping’
                    ;
mon_group_list      : NUMBER
                    | mon_group_list ’  ;’ NUMBER
                    ;
monetary_tlr        : ’END’ ’LC_MONETARY’ EOL
                    ;
/* The following is the LC_NUMERIC category grammar */
lc_numeric          : numeric_hdr numeric_keywords       numeric_tlr
                    | numeric_hdr ’copy’ locale_name EOL numeric_tlr
                    ;
numeric_hdr         : ’LC_NUMERIC’ EOL
                    ;
numeric_keywords    : numeric_keywords numeric_keyword
                    | numeric_keyword
                    ;
numeric_keyword     : num_keyword_string num_string EOL
                    | num_keyword_grouping num_group_list EOL
                    ;
num_keyword_string  : ’decimal_point’
                    | ’thousands_sep’
                    ;
num_string          : ’"’ char_list ’"’
                    | ’""’
                    ;
num_keyword_grouping: ’grouping’
                    ;
num_group_list      : NUMBER
                    | num_group_list ’  ;’ NUMBER
                    ;
```

```
numeric_tlr         : ’END’ ’LC_NUMERIC’ EOL
                    ;

/* The following is the LC_TIME category grammar */

lc_time             : time_hdr time_keywords          time_tlr
                    | time_hdr ’copy’ locale_name EOL time_tlr
                    ;

time_hdr            : ’LC_TIME’ EOL
                    ;

time_keywords       : time_keywords time_keyword
                    | time_keyword
                    ;

time_keyword        : time_keyword_name time_list EOL
                    | time_keyword_fmt time_string EOL
                    | time_keyword_opt time_list EOL
                    ;

time_keyword_name   : ’abday’   | ’day’ | ’abmon’ | ’mon’
                    ;

time_keyword_fmt    : ’d_t_fmt’   | ’d_fmt’ | ’t_fmt’
                    | ’am_pm’ | ’t_fmt_ampm’
                    ;

time_keyword_opt    : ’era’   | ’era_d_fmt’ | ’era_t_fmt’
                    | ’era_d_t_fmt’ | ’alt_digits’
                    ;

time_list           : time_list ’  ;’ time_string
                    | time_string
                    ;

time_string         : ’"’ char_list ’"’
                    ;

time_tlr            : ’END’ ’LC_TIME’ EOL
                    ;
```

5.5 Locale Definition Example

The following is an example of a locale definition file that could be used as input to the *localedef* utility. It assumes that the utility is executed with the −**f** option, naming a *charmap* file with (at least) the following content:

```
CHARMAP
<space>      \x20
<dollar>     \x24
<A>          \101
<a>          \141
<A-acute>    \346
<a-acute>    \365
<A-grave>    \300
<a-grave>    \366
<b>          \142
<C>          \103
<c>          \143
<c-cedilla>  \347
<d>          \x64
<H>          \110
<h>          \150
<eszet>      \xb7
<s>          \x73
<z>          \x7a
END CHARMAP
```

It should not be taken as complete or to represent any actual locale, but only to illustrate the syntax.

A further set of examples is offered as part of the **Internationalisation Guide**.

```
#
LC_CTYPE
lower   <a>;<b>;<c>;<c-cedilla>;<d>;...;<z>
upper   A;B;C;Ç;...;Z
space   \x20;\x09;\x0a;\x0b;\x0c;\x0d
blank   \040;\011
toupper (<a>,<A>);(b,B);(c,C);(ç,Ç);(d,D);(z,Z)
END LC_CTYPE
#
LC_COLLATE
#
# The following example of collation is based on the proposed
# Canadian standard Z243.4.1-1990, "Canadian Alphanumeric
# Ordering Standard For Character sets of CSA Z234.4 Standard".
# (Other parts of this example locale definition file do not
# purport to relate to Canada, or to any other real culture.)
# The proposed standard defines a 4-weight collation, such that
# in the first pass, characters are compared without regard to
# case or accents; in second pass, backwards compare without
# regard to case; in the third pass, forward compare without
# regard to diacriticals.  In the 3 first passes, non-alphabetic
# characters are ignored; in the fourth pass, only special
# characters are considered, such that "The string that has a
# special character in the lowest position comes first.  If two
# strings have a special character in the same position, the
# collation value of the special character determines ordering.
#
# Only a subset of the character set is used here; mostly to
# illustrate the set-up.
#
#
collating-symbol <LOW_VALUE>
collating-symbol <LOWER-CASE>
collating-symbol <SUBSCRIPT-LOWER>
collating-symbol <SUPERSCRIPT-LOWER>
collating-symbol <UPPER-CASE>
collating-symbol <NO-ACCENT>
collating-symbol <PECULIAR>
collating-symbol <LIGATURE>
collating-symbol <ACUTE>
collating-symbol <GRAVE>
# Further collating-symbols follow.
#
# Properly, the standard does not include any multi-character
# collating elements; the one below is added for completeness.
#
collating_element <ch> from "<c><h>"
collating_element <CH> from "<C><H>"
collating_element <Ch> from "<C><h>"
#
order_start forward;backward;forward;forward,position
#
# Collating symbols are specified first in the sequence to allocate
# basic collation values to them, lower than that of any character.
<LOW_VALUE>
<LOWER-CASE>
<SUBSCRIPT-LOWER>
<SUPERSCRIPT-LOWER>
<UPPER-CASE>
<NO-ACCENT>
<PECULIAR>
<LIGATURE>
<ACUTE>
<GRAVE>
<RING-ABOVE>
<DIAERESIS>
<TILDE>
# Further collating symbols are given a basic collating value here.
#
# Here follow special characters.
<space>        IGNORE;IGNORE;IGNORE;<space>
# Other special characters follow here.
#
```

```
# Here follow the regular characters.
<a>        <a>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<A>        <a>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<a-acute>  <a>;<ACUTE>;<LOWER-CASE>;IGNORE
<A-acute>  <a>;<ACUTE>;<UPPER-CASE>;IGNORE
<a-grave>  <a>;<GRAVE>;<LOWER-CASE>;IGNORE
<A-grave>  <a>;<GRAVE>;<UPPER-CASE>;IGNORE
<ae>      "<a><e>";"<LIGATURE><LIGATURE>";\
          "<LOWER-CASE><LOWER-CASE>";IGNORE
<AE>      "<a><e>";"<LIGATURE><LIGATURE>";\
          "<UPPER-CASE><UPPER-CASE>";IGNORE
<b>        <b>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<B>        <b>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<c>        <c>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<C>        <c>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<ch>       <ch>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<Ch>       <ch>;<NO-ACCENT>;<PECULIAR>;IGNORE
<CH>       <ch>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
#
# As an example, the strings "Bach" and "bach" could be encoded (for
# compare purposes) as:
# "Bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\
#            <NO_ACCENT>;<LOW_VALUE>;<UPPER>;<LOWER>;<LOWER>;<NULL>
# "bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\
#            <NO_ACCENT>;<LOW_VALUE>;<LOWER>;<LOWER>;<LOWER>;<NULL>
#
# The two strings are equal in pass 1 and 2, but differ in pass 3.
#
# Further characters follow.
#
UNDEFINED    IGNORE;IGNORE;IGNORE;IGNORE
#
order_end
#
END LC_COLLATE
#
LC_MONETARY
int_curr_symbol    "USD "
currency_symbol    "$"
mon_decimal_point  "."
mon_grouping       3;0
positive_sign      ""
negative_sign      "-"
p_cs_precedes      1
n_sign_posn        0
END LC_MONETARY
#
LC_NUMERIC
copy "US_en.ASCII"
END LC_NUMERIC
#
```

```
LC_TIME
abday   "Sun";"Mon";"Tue";"Wed";"Thu";"Fri";"Sat"
#
day     "Sunday";"Monday";"Tuesday";"Wednesday";\
        "Thursday";"Friday";"Saturday"
#
abmon   "Jan";"Feb";"Mar";"Apr";"May";"Jun";\
         "Jul";"Aug";"Sep";"Oct";"Nov";"Dec"
#
mon     "January";"February";"March";"April";\
        "May";"June";"July";"August";"September";\
        "October";"November";"December"
#
d_t_fmt "%a %b %d %T %Z %Y\n"
END LC_TIME
#
LC_MESSAGES
yesexpr "^([yY][[:alpha:]]*)|(OK)"
#
noexpr  "^[nN][[:alpha:]]*"
END LC_MESSAGES
```

Chapter 6

Environment Variables

6.1 Environment Variable Definition

Environment variables defined in this chapter affect the operation of multiple utilities, functions and applications. There are other environment variables that are of interest only to specific utilities. Environment variables that apply to a single utility only are defined as part of the utility description. See the **ENVIRONMENT VARIABLES** section of the utility descriptions in the **XCU** specification for information on environment variable usage.

The value of an environment variable is a string of characters. For a C-language program, an array of strings called the environment is made available when a process begins. The array is pointed to by the external variable *environ*, which is defined as:

```
extern char **environ;
```

These strings have the form *name*=*value*; *name*s do not contain the character =. For values to be portable across XSI-conformant systems, the value must be composed of characters from the portable character set (except NUL and as indicated below). There is no meaning associated with the order of strings in the environment. If more than one string in a process' environment has the same *name*, the consequences are undefined.

Environment variable names used by the utilities in the **XCU** specification consist solely of upper-case letters, digits and the "_" (underscore) from the characters defined in Table 4-1 on page 43. Other characters may be permitted by an implementation; applications must tolerate the presence of such names. Upper- and lower-case letters retain their unique identities and are not folded together. The name space of environment variable names containing lower-case letters is reserved for applications. Applications can define any environment variables with names from this name space without modifying the behaviour of the standard utilities.

The *values* that the environment variables may be assigned are not restricted except that they are considered to end with a null byte and the total space used to store the environment and the arguments to the process is limited to {ARG_MAX} bytes.

EX Other *name*=*value* pairs may be placed in the environment by, for example, calling the *putenv*( ) function, manipulating the *environ* variable, or by using *envp* arguments when creating a process; see *exec* in the **XSH** specification.

It is unwise to conflict with certain variables that are frequently exported by widely used command interpreters and applications:

ARFLAGS
CC
CDPATH
CFLAGS
CHARSET
COLUMNS
DATEMSK
DEAD
EDITOR
ENV
EXINIT
FC
FCEDIT
FFLAGS
GET
GFLAGS
HISTFILE
HISTORY
HISTSIZE
HOME
IFS
LANG
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
LDFLAGS
LEX
LFLAGS
LINENO
LINES
LISTER
LOGNAME
LPDEST
MAIL
MAILCHECK
MAILER
MAILPATH
MAILRC
MAKEFLAGS
MAKESHELL
MANPATH
MBOX
MORE
MSGVERB
NLSPATH
NPROC
OLDPWD
OPTARG
OPTERR
OPTIND
PAGER
PATH
PPID
PRINTER
PROCLANG
PROJECTDIR
PS1
PS2
PS3
PS4
PWD
RANDOM
SECONDS
SHELL
TERM
TERMCAP
TERMINFO
TMPDIR
TZ
USER
VISUAL
YACC
YFLAGS

If the variables in the following two sections are present in the environment during the execution of an application or utility, they are given the meaning described below. Some are placed into the environment by the implementation at the time the user logs in; all can be added or changed by the user or any ancestor of the current process. The implementation will add or change environment variables named in this specification set only as specified in this specification set. If they are defined in the application's environment, the utilities in the **XCU** specification and the functions in the **XSH** specification assume they have the specified meaning. Conforming applications must not set these environment variables to have meanings other than as described. See *getenv*( ) and the **XCU** specification, **Section 2.12**, **Shell Execution Environment** for methods of accessing these variables.

6.2 Internationalisation Variables

This section describes environment variables that are relevant to the operation of internationalised interfaces described in the CAE Specification, **System Interfaces and Headers, Issue 5** and the CAE Specification, **Commands and Utilities, Issue 5**.

Users may use the following environment variables to announce specific localisation requirements to applications. Applications must retrieve this information using the *setlocale*( ) function to initialise the correct behaviour of the internationalised interfaces. The descriptions of the internationalisation environment variables describe the resulting behaviour only when the application locale is initialised in this way.

LANG

This variable determines the locale category for native language, local customs and coded character set in the absence of the *LC_ALL* and other *LC_** (*LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME*) environment variables. This can be used by applications to determine the language to use for error messages and instructions, collating sequences, date formats, and so forth.

LC_ALL

This variable determines the values for all locale categories. The value of the *LC_ALL* environment variable has precedence over any of the other environment variables starting with *LC_* (*LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME*) and the *LANG* environment variable.

LC_COLLATE

This variable determines the locale category for character collation. It determines collation information for regular expressions and sorting, including equivalence classes and multi-character collating elements, in various utilities and the *strcoll*( ) and *strxfrm*( ) functions. Additional semantics of this variable, if any, are implementation-dependent.

LC_CTYPE

This variable determines the locale category for character handling functions, such as *tolower*( ), *toupper*( ) and *isalpha*( ). This environment variable determines the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters), the classification of characters (for example, alpha, digit, graph) and the behaviour of character classes. Additional semantics of this variable, if any, are implementation-dependent.

LC_MESSAGES

This variable determines the locale category for processing affirmative and negative responses and the language and cultural conventions in which messages should be written.
EX It also affects the behaviour of the *catopen*( ) function in determining the message catalogue.
Additional semantics of this variable, if any, are implementation-dependent. The language and cultural conventions of diagnostic and informative messages whose format is unspecified by this specification set should be affected by the setting of *LC_MESSAGES*.

LC_MONETARY

This variable determines the locale category for monetary-related numeric formatting information. Additional semantics of this variable, if any, are implementation-dependent.

LC_NUMERIC

This variable determines the locale category for numeric formatting (for example, thousands separator and radix character) information in various utilities as well as the formatted I/O operations in *printf*( ) and *scanf*( ) and the string conversion functions in *strtod*( ). Additional semantics of this variable, if any, are implementation-dependent.

LC_TIME

This variable determines the locale category for date and time formatting information. It affects the behaviour of the time functions in *strftime*(). Additional semantics of this variable, if any, are implementation-dependent.

EX *NLSPATH*

This variable contains a sequence of templates that the *catopen*() function uses when attempting to locate message catalogues. Each template consists of an optional prefix, one or more substitution fields, a filename and an optional suffix.

For example:

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

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

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

%N The value of the *name* parameter passed to *catopen*().

%L The value of the LC_MESSAGES category.

%l The *language* element from the LC_MESSAGES category.

%t The *territory* element from the LC_MESSAGES category.

%c The *codeset* element from the LC_MESSAGES category.

%% A single % character.

An empty string is substituted if the specified value is not currently defined. The separators underscore (_) and period (.) are not included in %t and %c substitutions.

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

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

indicates to *catopen*() that it should look for the requested message catalogue in *name*, *name***.cat** and **/nlslib/***category***/***name***.cat**, where *category* is the value of the LC_MESSAGES category of the current locale.

Users should not set the *NLSPATH* variable unless they have a specific reason to override the default system path. Doing so causes undefined behaviour in the standard utilities.

The environment variables *LANG*, *LC_ALL*, *LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, EX *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME* (*LC_**) and *NLSPATH* provide for the support of internationalised applications. The standard utilities make use of these environment variables as described in this section and the individual **ENVIRONMENT VARIABLES** sections for the utilities. If these variables specify locale categories that are not based upon the same underlying codeset, the results are unspecified.

The values of locale categories are determined by a precedence order; the first condition met below determines the value:

1. If the *LC_ALL* environment variable is defined and is not null, the value of *LC_ALL* is used.
2. If the *LC_** environment variable (*LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME*) is defined and is not null, the value of the

environment variable is used to initialise the category that corresponds to the environment variable.

3. If the *LANG* environment variable is defined and is not null, the value of the *LANG* environment variable is used.

4. If the *LANG* environment variable is not set or is set to the empty string, the implementation-dependent default locale is used.

If the locale value is "C" or "POSIX", the POSIX locale is used and the standard utilities behave in accordance with the rules in Section 5.2 on page 50, for the associated category.

If the locale value begins with a slash, it is interpreted as the pathname of a file that was created in the output format used by the *localedef* utility; see **OUTPUT FILES** under *localedef*. Referencing such a pathname will result in that locale being used for the indicated category.

EX If the locale value has the form:

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

it refers to an implementation-provided locale, where settings of language, territory and codeset are implementation-dependent.

EX *LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC* and *LC_TIME* are defined to accept an additional field ‘‘*@modifier*’’, which allows the user to select a specific instance of localisation data within a single category (for example, for selecting the dictionary as opposed to the character ordering of data). The syntax for these environment variables is thus defined as:

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

For example, if a user wanted to interact with the system in French, but required to sort German text files, *LANG* and *LC_COLLATE* could be defined as:

```
LANG=Fr_FR
LC_COLLATE=De_DE
```

This could be extended to select dictionary collation (say) by use of the *@modifier* field; for example:

```
LC_COLLATE=De_DE@dict
```

An implementation may support other formats.

If the locale value is not recognised by the implementation, the behaviour is unspecified.

At run time, these values are bound to a program’s locale by calling the *setlocale*( ) function.

Additional criteria for determining a valid locale name are implementation-dependent.

6.3 Other Environment Variables

COLUMNS

A decimal integer > 0 used to indicate the user's preferred width in column positions for the terminal screen or window. (See **column position** on page 10.) If this variable is unset or null, the implementation determines the number of columns, appropriate for the terminal or window, in an unspecified manner. When *COLUMNS* is set, any terminal-width information implied by *TERM* will be overridden. Users and portable applications should not set *COLUMNS* unless they wish to override the system selection and produce output unrelated to the terminal characteristics.

The default value for the number of column positions is unspecified because historical implementations use different methods to determine values corresponding to the size of the screen in which the utility is run. This size is typically known to the implementation through the value of *TERM*, or by more elaborate methods such as extensions to the *stty* utility, or knowledge of how the user is dynamically resizing windows on a bit-mapped display terminal. Users should not need to set this variable in the environment unless there is a specific reason to override the implementation's default behaviour, such as to display data in an area arbitrarily smaller than the terminal or window.

EX *DATEMSK*

Indicates the pathname of the template file used by *getdate*( ).

FIPS *HOME* The system will initialise this variable at the time of login to be a pathname of the user's home directory. See **<pwd.h>**.

LINES A decimal integer > 0 used to indicate the user's preferred number of lines on a page or the vertical screen or window size in lines. A line in this case is a vertical measure large enough to hold the tallest character in the character set being displayed. If this variable is unset or null, the implementation determines the number of lines, appropriate for the terminal or window (size, terminal baud rate, and so forth), in an unspecified manner. When *LINES* is set, any terminal-height information implied by *TERM* will be overridden. Users and portable applications should not set *LINES* unless they wish to override the system selection and produce output unrelated to the terminal characteristics.

The default value for the number of lines is unspecified because historical implementations use different methods to determine values corresponding to the size of the screen in which the utility is run. This size is typically known to the implementation through the value of *TERM*, or by more elaborate methods such as extensions to the *stty* utility, or knowledge of how the user is dynamically resizing windows on a bit-mapped display terminal. Users should not need to set this variable in the environment unless there is a specific reason to override the implementation's default behaviour, such as to display data in an area arbitrarily smaller than the terminal or window.

LOGNAME

FIPS The system will initialise this variable at the time of login to be the user's login name. See **<pwd.h>**. For a value of *LOGNAME* to be portable across implementations of the ISO POSIX-1 standard, the value should be composed of characters from the portable filename character set.

EX *MSGVERB*

Describes which message components are to be used in writing messages by *fmtmsg*( ).

PATH The sequence of path prefixes that certain functions and utilities apply in searching for an executable file known only by a filename. The prefixes are separated by a colon (:) When a non-zero-length prefix is applied to this filename, a slash is inserted between the prefix and the filename. A zero-length prefix is a legacy feature that indicates the current working directory. It appears as two adjacent colons (::), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. A portable application must use an actual pathname (such as .) to represent the current working directory in *PATH*. The list is searched from beginning to end, applying the filename to each prefix, until an executable file with the specified name and appropriate execution permissions is found. If the pathname being sought contains a slash, the search through the path prefixes will not be performed. If the pathname begins with a slash, the specified path is resolved (see **pathname resolution** on page 22). If *PATH* is unset or is set to null, the path search is implementation-dependent.

SHELL A pathname of the user's preferred command language interpreter. If this interpreter does not conform to the XSI Shell Command Language in the **XCU** specification, **Chapter 2**, **Shell Command Language**, utilities may behave differently from those described in this specification set.

TMPDIR

A pathname of a directory made available for programs that need a place to create temporary files.

TERM The terminal type for which output is to be prepared. This information is used by utilities and application programs wishing to exploit special capabilities specific to a terminal. The format and allowable values of this environment variable are unspecified.

TZ Timezone information. The contents of the environment variable named *TZ* are used by the *ctime*( ), *localtime*( ), *strftime*( ) and *mktime*( ) functions, and by various utilities, to override the default timezone. The value of *TZ* has one of the two forms (spaces inserted for clarity):

```
:characters
```

or:

```
std offset dst offset, rule
```

If *TZ* is of the first format (that is, if the first character is a colon), the characters following the colon are handled in an implementation-dependent manner.

The expanded format (for all *TZ*s whose value does not have a colon as the first character) is as follows:

```
stdoffset[dst[offset][,start[/time],end[/time]]]
```

Where:

std and *dst*

Indicates no less than three, nor more than {TZNAME_MAX}, bytes that are the designation for the standard (*std*) or the alternative (*dst* — such as Daylight Savings Time) timezone. Only *std* is required; if *dst* is missing, then the alternative time does not apply in this locale. Upper- and lower-case letters are explicitly allowed. Any graphic characters except a leading colon (:) or digits, the comma (,), the minus (–), the plus (+), and the null character are permitted to appear in these fields, but their meaning is unspecified.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

*hh***[**:*mm***[**:*ss***]]**

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, the alternative time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour is between zero and 24, and the minutes (and seconds) if present between zero and 59. Use of values outside these ranges causes undefined behaviour. If preceded by a –, the timezone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding +).

rule Indicates when to change to and back from the alternative time. The *rule* has the form:

`date[/time],date[/time]`

where the first *date* describes when the change from standard to alternative time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

J*n* The Julian day *n* ($1 \leq n \leq 365$). Leap days are not counted. That is, in all years including leap years February 28 is day 59 and March 1 is day 60. It is impossible to refer explicitly to the occasional February 29.

n The zero-based Julian day ($0 \leq n \leq 365$). Leap days are counted, and it is possible to refer to February 29.

M*m*.*n*.*d*
The d^{th} day ($0 \leq d \leq 6$) of week *n* of month *m* of the year ($1 \leq n \leq 5$, $1 \leq m \leq 12$, where week 5 means ‘‘the last *d* day in month *m*’’ which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*’th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign (– or +) is allowed. The default, if *time* is not given, is 02:00:00.

Chapter 7

Regular Expressions

Note: Two versions of regular expressions are supported in this specification set:

- the historical **Simple Regular Expressions**, which provide backward compatibility, but which may be withdrawn from a future issue of this specification set
- the improved internationalised version that complies with the ISO / IEC 9945-2: 1993 standard.

The first (historical) version is described as part of the *regexp*( ) function in the **XSH** specification. The second (improved) version is described in this chapter.

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, where these character sets are interpreted according to the current locale. While many regular expressions can be interpreted differently depending on the current locale, many features, such as character class expressions, provide for contextual invariance across locales.

The Basic Regular Expression (BRE) notation and construction rules in Section 7.3 on page 104 apply to most utilities supporting regular expressions. Some utilities, instead, support the Extended Regular Expressions (ERE) described in Section 7.4 on page 109; any exceptions for both cases are noted in the descriptions of the specific utilities using regular expressions. Both BREs and EREs are supported by the Regular Expression Matching interface in the **XSH** specification under *regcomp*( ), *regexec*( ) and related functions.

7.1 Regular Expression Definitions

For the purposes of this section, the following definitions apply:

entire regular expression
The concatenated set of one or more BREs or EREs that make up the pattern specified for string selection.

matched
A sequence of zero or more characters is said to be matched by a BRE or ERE when the characters in the sequence correspond to a sequence of characters defined by the pattern.

Matching is based on the bit pattern used for encoding the character, not on the graphic representation of the character. This means that if a character set contains two or more encodings for a graphic symbol, or if the strings searched contain text encoded in more than one codeset, no attempt is made to search for any other representation of the encoded symbol. If that is required, the user can specify equivalence classes containing all variations of the desired graphic symbol.

The search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found, where *first* is defined to mean ‘‘begins earliest in the string’’. If the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence will be matched. For example: the BRE bb* matches the second to fourth characters of abbbc, and the ERE (wee | week)(knights | night) matches all ten characters of weeknights.

Consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, matches the longest possible string. For this purpose, a null string is considered to be longer than no match at all. For example, matching the BRE \(.*\).* against abcdef, the subexpression (\1) is abcdef, and matching the BRE \(a*\)* against bc, the subexpression (\1) is the null string.

It is possible to determine what strings correspond to subexpressions by recursively applying the leftmost longest rule to each subexpression, but only with the proviso that the overall match is leftmost longest. For example, matching \(ac*\)c*d[ac]*\1 against acdacaaa matches acdacaaa (with \1=a); simply matching the longest match for \(ac*\) would yield \1=ac, but the overall match would be smaller (acdac). Conceptually, the implementation must examine every possible match and among those that yield the leftmost longest total matches, pick the one that does the longest match for the leftmost subexpression and so on. Note that this means that matching by subexpressions is context-dependent: a subexpression within a larger RE may match a different string from the one it would match as an independent RE, and two instances of the same subexpression within the same larger RE may match different lengths even in similar sequences of characters. For example, in the ERE (a.*b)(a.*b), the two identical subexpressions would match four and six characters, respectively, of accbaccccb.

When a multi-character collating element in a bracket expression (see Section 7.3.5 on page 105) is involved, the longest sequence will be measured in characters consumed from the string to be matched; that is, the collating element counts not as one element, but as the number of characters it matches.

BRE (ERE) matching a single character
A BRE or ERE that matches either a single character or a single collating element.

Only a BRE or ERE of this type that includes a bracket expression (see Section 7.3.5 on page 105) can match a collating element.

The definition of *single character* has been expanded to include also collating elements consisting of two or more characters; this expansion is applicable only when a bracket expression is included in the BRE or ERE. An example of such a collating element may be the Dutch ij, which collates as a y. In some encodings, a ligature ‘‘i with j’’ exists as a character and would represent a single-character collating element. In another encoding, no such ligature exists, and the two-character sequence ij is defined as a multi-character collating element. Outside brackets, the ij is treated as a two-character RE and matches the same characters in a string. Historically, a bracket expression only matched a single character. If, however, the bracket expression defines, for example, a range that includes ij, then this particular bracket expression will also match a sequence of the two characters i and j in the string.

BRE (ERE) matching multiple characters
A BRE or ERE that matches a concatenation of single characters or collating elements.

Such a BRE or ERE is made up from a BRE (ERE) matching a single character and BRE (ERE) special characters.

invalid
This section uses the term *invalid* for certain constructs or conditions. Invalid REs will cause the utility or function using the RE to generate an error condition. When *invalid* is not used, violations of the specified syntax or semantics for REs produce undefined results: this may entail an error, enabling an extended syntax for that RE, or using the construct in error as literal characters to be matched. For example, the BRE construct \{1,2,3\} does not comply with the grammar. A portable application cannot rely on it producing an error nor matching the literal characters \{1,2,3\}.

7.2 Regular Expression General Requirements

The requirements in this section apply to both basic and extended regular expressions.

The use of regular expressions is generally associated with text processing. REs (BREs and EREs) operate on text strings; that is, zero or more characters followed by an end-of-string delimiter (typically NUL). Some utilities employing regular expressions limit the processing to lines; that is, zero or more characters followed by a newline character. In the regular expression processing described in this specification, the newline character is regarded as an ordinary character and both a period and a non-matching list can match one. The **XCU** specification specifies within the individual descriptions of those standard utilities employing regular expressions whether they permit matching of newline characters; if not stated otherwise, the use of literal newline characters or any escape sequence equivalent produces undefined results. Those utilities (like *grep*) that do not allow newline characters to match are responsible for eliminating any newline character from strings before matching against the RE. The *regcomp*( ) function in the **XSH** specification, however, can provide support for such processing without violating the rules of this section.

The interfaces specified in this specification set do not permit the inclusion of a NUL character in an RE or in the string to be matched. If during the operation of a standard utility a NUL is included in the text designated to be matched, that NUL may designate the end of the text string for the purposes of matching.

When a standard utility or function that uses regular expressions specifies that pattern matching will be performed without regard to the case (upper- or lower-) of either data or patterns, then when each character in the string is matched against the pattern, not only the character, but also its case counterpart (if any), will be matched. This definition of case-insensitive processing is intended to allow matching of multi-character collating elements as well as characters. For instance, as each character in the string is matched using both its cases, the RE [[.Ch.]] when matched against the string char, is in reality matched against ch, Ch, cH and CH.

The implementation will support any regular expression that does not exceed 256 bytes in length.

7.3 Basic Regular Expressions

7.3.1 BREs Matching a Single Character or Collating Element

A BRE ordinary character, a special character preceded by a backslash or a period matches a single character. A bracket expression matches a single character or a single collating element.

7.3.2 BRE Ordinary Characters

An ordinary character is a BRE that matches itself: any character in the supported character set, except for the BRE special characters listed in Section 7.3.3.

The interpretation of an ordinary character preceded by a backslash (\) is undefined, except for:

1. the characters), (, { and }
2. the digits 1 to 9 inclusive (see Section 7.3.6 on page 107)
3. a character inside a bracket expression.

7.3.3 BRE Special Characters

A *BRE special character* has special properties in certain contexts. Outside those contexts, or when preceded by a backslash, such a character will be a BRE that matches the special character itself. The BRE special characters and the contexts in which they have their special meaning are:

. [\ The period, left-bracket and backslash is special except when used in a bracket expression (see Section 7.3.5 on page 105). An expression containing a [that is not preceded by a backslash and is not part of a bracket expression produces undefined results.

* The asterisk is special except when used:

- in a bracket expression
- as the first character of an entire BRE (after an initial ˆ, if any)
- as the first character of a subexpression (after an initial ˆ, if any); see Section 7.3.6 on page 107.

ˆ The circumflex is special when used:

- as an anchor (see Section 7.3.8 on page 108)
- as the first character of a bracket expression (see Section 7.3.5 on page 105).

$ The dollar sign is special when used as an anchor.

7.3.4 Periods in BREs

A period (.), when used outside a bracket expression, is a BRE that matches any character in the supported character set except NUL.

7.3.5 RE Bracket Expression

A bracket expression (an expression enclosed in square brackets, [ ]) is an RE that matches a single collating element contained in the non-empty set of collating elements represented by the bracket expression.

The following rules and definitions apply to bracket expressions:

1. A *bracket expression* is either a matching list expression or a non-matching list expression. It consists of one or more expressions: collating elements, collating symbols, equivalence classes, character classes or range expressions. Portable applications must not use range expressions, even though all implementations support them. The right-bracket (]) loses its special meaning and represents itself in a bracket expression if it occurs first in the list (after an initial circumflex (^), if any). Otherwise, it terminates the bracket expression, unless it appears in a collating symbol (such as [.].]) or is the ending right-bracket for a collating symbol, equivalence class or character class. The special characters:

   ```
   .    *    [    \
   ```

   (period, asterisk, left-bracket and backslash, respectively) lose their special meaning within a bracket expression.

   The character sequences:

   ```
   [.    [=    [:
   ```

   (left-bracket followed by a period, equals-sign or colon) are special inside a bracket expression and are used to delimit collating symbols, equivalence class expressions and character class expressions. These symbols must be followed by a valid expression and the matching terminating sequence .], =] or :], as described in the following items.

2. A *matching list* expression specifies a list that matches any one of the expressions represented in the list. The first character in the list must not be the circumflex. For example, [abc] is an RE that matches any of the characters a, b or c.

3. A *non-matching list* expression begins with a circumflex (^), and specifies a list that matches any character or collating element except for the expressions represented in the list after the leading circumflex. For example, [^abc] is an RE that matches any character or collating element except the characters a, b or c. The circumflex will have this special meaning only when it occurs first in the list, immediately following the left-bracket.

4. A *collating symbol* is a collating element enclosed within bracket-period ([. .]) delimiters. Collating elements are defined as described in **Collation Order** on page 63. Multi-character collating elements must be represented as collating symbols when it is necessary to distinguish them from a list of the individual characters that make up the multi-character collating element. For example, if the string ch is a collating element in the current collation sequence with the associated collating symbol <ch>, the expression [[.ch.]] will be treated as an RE matching the character sequence ch, while [ch] will be treated as an RE matching c or h. Collating symbols will be recognised only inside bracket expressions. This implies that the RE [[.ch.]]*c matches the first to fifth character in the string chchch. If the string is not a collating element in the current collating sequence definition, or if the collating element has no characters associated with it (for example, see the symbol <HIGH> in the example collation definition shown in **Collation Order** on page 63), the symbol will be treated as an invalid expression.

5. An *equivalence class expression* represents the set of collating elements belonging to an equivalence class, as described in **Collation Order**. Only primary equivalence classes will be recognised. The class is expressed by enclosing any one of the collating elements in the

equivalence class within bracket-equal ([= =]) delimiters. For example, if a, à and â belong to the same equivalence class, then [[=a=]b], [[=à=]b] and [[=â=]b] will each be equivalent to [aàâb]. If the collating element does not belong to an equivalence class, the equivalence class expression will be treated as a *collating symbol*.

6. A *character class expression* represents the set of characters belonging to a character class, as defined in the LC_CTYPE category in the current locale. All character classes specified in the current locale will be recognised. A character class expression is expressed as a character class name enclosed within bracket-colon ([: :]) delimiters.

   The following character class expressions are supported in all locales:

   ```
   [:alnum:]   [:cntrl:]   [:lower:]   [:space:]
   [:alpha:]   [:digit:]   [:print:]   [:upper:]
   [:blank:]   [:graph:]   [:punct:]   [:xdigit:]
   ```

   EX In addition, character class expressions of the form:

   ```
   [:name:]
   ```

   are recognised in those locales where the *name* keyword has been given a **charclass** definition in the LC_CTYPE category.

7. A *range expression* represents the set of collating elements that fall between two elements in the current collation sequence, inclusively. It is expressed as the starting point and the ending point separated by a hyphen (–).

   Range expressions must not be used in portable applications because their behaviour is dependent on the collating sequence. Ranges will be treated according to the current collating sequence, and include such characters that fall within the range based on that collating sequence, regardless of character values. This, however, means that the interpretation will differ depending on collating sequence. If, for instance, one collating sequence defines ä as a variant of a, while another defines it as a letter following z, then the expression [ä–z] is valid in the first language and invalid in the second.

   In the following, all examples assume the collation sequence specified for the POSIX locale, unless another collation sequence is specifically defined.

   The starting range point and the ending range point must be a collating element or collating symbol. An equivalence class expression used as a starting or ending point of a range expression produces unspecified results. An equivalence class can be used portably within a bracket expression, but only outside the range. For example, the unspecified expression [[=e=]–f] should be given as [[=e=]e–f]. The ending range point must collate equal to or higher than the starting range point; otherwise, the expression will be treated as invalid. The order used is the order in which the collating elements are specified in the current collation definition. One-to-many mappings (see the description of *LC_COLLATE* in Chapter 5 on page 49) will not be performed. For example, assuming that the character eszet (β) is placed in the collation sequence after r and s, but before t and that it maps to the sequence ss for collation purposes, then the expression [r–s] matches only r and s, but the expression [s–t] matches s, β or t.

   The interpretation of range expressions where the ending range point is also the starting range point of a subsequent range expression (for instance [a–m–o]) is undefined.

   The hyphen character will be treated as itself if it occurs first (after an initial ˆ, if any) or last in the list, or as an ending range point in a range expression. As examples, the expressions [–ac] and [ac–] are equivalent and match any of the characters a, c or –; [ˆ–ac] and [ˆac–] are equivalent and match any characters except a, c or –; the expression [%––] matches any of the characters between % and – inclusive; the expression [––@] matches any of the characters between – and @ inclusive; and the expression [a––@] is invalid, because the letter a follows the symbol – in the POSIX locale. To use a hyphen as the starting range point, it must either come first in the bracket expression or be specified as a collating symbol, for example: [][.–.]–0], which matches either a right bracket or any character or collating element that collates between hyphen and 0, inclusive.

If a bracket expression must specify both – and], the] must be placed first (after the ˆ, if any) and the – last within the bracket expression.

7.3.6 BREs Matching Multiple Characters

The following rules can be used to construct BREs matching multiple characters from BREs matching a single character:

1. The concatenation of BREs matches the concatenation of the strings matched by each component of the BRE.

2. A *subexpression* can be defined within a BRE by enclosing it between the character pairs \(and \) . Such a subexpression matches whatever it would have matched without the \(and \), except that anchoring within subexpressions is optional behaviour; see Section 7.3.8 on page 108. Subexpressions can be arbitrarily nested.

3. The *back-reference* expression *n* matches the same (possibly empty) string of characters as was matched by a subexpression enclosed between \(and \) preceding the *n*. The character *n* must be a digit from 1 to 9 inclusive, *n*th subexpression (the one that begins with the *n*th \(and ends with the corresponding paired \)). The expression is invalid if less than *n* subexpressions precede the *n*. For example, the expression ˆ\(.*\)\1$ matches a line consisting of two adjacent appearances of the same string, and the expression \(a\)*\1 fails to match a. The limit of nine back-references to subexpressions in the RE is based on the use of a single digit identifier. This does not imply that only nine subexpressions are allowed in REs. The following is a valid BRE with ten subexpressions:

   ```
   \(\(\(ab\)*c\)*d\)\(ef\)*\(gh\)\{2\}\(ij\)*\(kl\)*\(mn\)*\(op\)*\(qr\)*
   ```

4. When a BRE matching a single character, a subexpression or a back-reference is followed by the special character asterisk (*), together with that asterisk it matches what zero or more consecutive occurrences of the BRE would match. For example, [ab]* and [ab][ab] are equivalent when matching the string ab.

5. When a BRE matching a single character, a subexpression or a back-reference is followed by an *interval expression* of the format \{*m*\}, \{*m*,\} or \{*m*,*n*\}, together with that interval expression it matches what repeated consecutive occurrences of the BRE would match. The values of *m* and *n* will be decimal integers in the range 0 ≤ *m* ≤ *n* ≤ {RE_DUP_MAX}, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression \{*m*\} matches exactly *m* occurrences of the preceding BRE, \{*m*,\} matches at least *m* occurrences and \{*m*,*n*\} matches any number of occurrences between *m* and *n*, inclusive.

   For example, in the string abababccccccd the BRE c\{3\} is matched by characters seven to nine, the BRE \(ab\)\{4,\} is not matched at all and the BRE c\{1,3\}d is matched by characters ten to thirteen.

The behaviour of multiple adjacent duplication symbols (* and intervals) produces undefined results.

7.3.7 BRE Precedence

The order of precedence is as shown in the following table:

BRE Precedence (from high to low)	
collation-related bracket symbols	[= =] [: :] [. .]
escaped characters	\\<*special character*>
bracket expression	[ ]
subexpressions / back-references	\\(\\) \\n
single-character-BRE duplication	* \\{m,n\\}
concatenation	
anchoring	^ $

7.3.8 BRE Expression Anchoring

A BRE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar sign special characters will be considered BRE anchors in the following contexts:

1. A circumflex (^) is an anchor when used as the first character of an entire BRE. The implementation may treat circumflex as an anchor when used as the first character of a subexpression. The circumflex will anchor the expression (or optionally subexpression) to the beginning of a string; only sequences starting at the first character of a string will be matched by the BRE. For example, the BRE ^ab matches ab in the string abcdef, but fails to match in the string cdefab. The BRE \(^ab\) may match the former string. A portable BRE must escape a leading circumflex in a subexpression to match a literal circumflex.

2. A dollar sign ($) is an anchor when used as the last character of an entire BRE. The implementation may treat a dollar sign as an anchor when used as the last character of a subexpression. The dollar sign will anchor the expression (or optionally subexpression) to the end of the string being matched; the dollar sign can be said to match the end-of-string following the last character.

3. A BRE anchored by both "^" and "$" matches only an entire string. For example, the BRE ^abcdef$ matches strings consisting only of abcdef.

7.4 Extended Regular Expressions

The *extended regular expression* (ERE) notation and construction rules will apply to utilities defined as using extended regular expressions; any exceptions to the following rules are noted in the descriptions of the specific utilities using EREs.

7.4.1 EREs Matching a Single Character or Collating Element

An ERE ordinary character, a special character preceded by a backslash or a period matches a single character. A bracket expression matches a single character or a single collating element. An *ERE matching a single character* enclosed in parentheses matches the same as the ERE without parentheses would have matched.

7.4.2 ERE Ordinary Characters

An *ordinary character* is an ERE that matches itself. An ordinary character is any character in the supported character set, except for the ERE special characters listed in Section 7.4.3. The interpretation of an ordinary character preceded by a backslash (\) is undefined.

7.4.3 ERE Special Characters

An *ERE special character* has special properties in certain contexts. Outside those contexts, or when preceded by a backslash, such a character is an ERE that matches the special character itself. The extended regular expression special characters and the contexts in which they have their special meaning are:

. [\ (The period, left-bracket, backslash and left-parenthesis are special except when used in a bracket expression (see Section 7.3.5 on page 105). Outside a bracket expression, a left-parenthesis immediately followed by a right-parenthesis produces undefined results.

) The right-parenthesis is special when matched with a preceding left-parenthesis, both outside a bracket expression.

* + ? { The asterisk, plus-sign, question-mark and left-brace are special except when used in a bracket expression (see Section 7.3.5 on page 105). Any of the following uses produce undefined results:

- if these characters appear first in an ERE, or immediately following a vertical-line, circumflex or left-parenthesis
- if a left-brace is not part of a valid interval expression.

| The vertical-line is special except when used in a bracket expression (see Section 7.3.5 on page 105). A vertical-line appearing first or last in an ERE, or immediately following a vertical-line or a left-parenthesis, or immediately preceding a right-parenthesis, produces undefined results.

^ The circumflex is special when used:

- as an anchor (see Section 7.4.9 on page 111)
- as the first character of a bracket expression (see Section 7.3.5 on page 105).

$ The dollar sign is special when used as an anchor.

7.4.4 Periods in EREs

A period (.), when used outside a bracket expression, is an ERE that matches any character in the supported character set except NUL.

7.4.5 ERE Bracket Expression

The rules for ERE Bracket Expressions are the same as for Basic Regular Expressions; see Section 7.3.5 on page 105.

7.4.6 EREs Matching Multiple Characters

The following rules will be used to construct EREs matching multiple characters from EREs matching a single character:

1. A *concatenation of EREs* matches the concatenation of the character sequences matched by each component of the ERE. A concatenation of EREs enclosed in parentheses matches whatever the concatenation without the parentheses matches. For example, both the ERE cd and the ERE (cd) are matched by the third and fourth character of the string abcdefabcdef.

2. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character plus-sign (+), together with that plus-sign it matches what one or more consecutive occurrences of the ERE would match. For example, the ERE b+(bc) matches the fourth to seventh characters in the string acabbbcde. And, [ab]+ and [ab][ab]* are equivalent.

3. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character asterisk (*), together with that asterisk it matches what zero or more consecutive occurrences of the ERE would match. For example, the ERE b*c matches the first character in the string cabbbcde, and the ERE b*cd matches the third to seventh characters in the string cabbbcdebbbbbbcdbc. And, [ab]* and [ab][ab] are equivalent when matching the string ab.

4. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character question-mark (?), together with that question-mark it matches what zero or one consecutive occurrences of the ERE would match. For example, the ERE b?c matches the second character in the string acabbbcde.

5. When an ERE matching a single character or an ERE enclosed in parentheses is followed by an *interval expression* of the format {*m*}, {*m*,} or {*m*,*n*}, together with that interval expression it matches what repeated consecutive occurrences of the ERE would match. The values of *m* and *n* will be decimal integers in the range 0 ≤ *m* ≤ *n* ≤ {RE_DUP_MAX}, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression {*m*} matches exactly *m* occurrences of the preceding ERE, {*m*,} matches at least *m* occurrences and {*m*,*n*} matches any number of occurrences between *m* and *n*, inclusive.

   For example, in the string abababccccccd the ERE c{3} is matched by characters seven to nine and the ERE (ab){2,} is matched by characters one to six.

The behaviour of multiple adjacent duplication symbols (+, *, ? and intervals) produces undefined results.

7.4.7 ERE Alternation

Two EREs separated by the special character vertical-line (|) match a string that is matched by either. For example, the ERE a((bc)|d) matches the string abc and the string ad. Single characters, or expressions matching single characters, separated by the vertical bar and enclosed in parentheses, will be treated as an ERE matching a single character.

7.4.8 ERE Precedence

The order of precedence will be as shown in the following table:

ERE Precedence (from high to low)	
collation-related bracket symbols	[= =] [: :] [. .]
escaped characters	\<*special character*>
bracket expression	[ ]
grouping	( )
single-character-ERE duplication	* + ? {*m*,*n*}
concatenation	
anchoring	^ $
alternation	|

For example, the ERE abba | cde matches either the string abba or the string cde (rather than the string abbade or abbcde, because concatenation has a higher order of precedence than alternation).

7.4.9 ERE Expression Anchoring

An ERE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar sign special characters are considered ERE anchors when used anywhere outside a bracket expression. This has the following effects:

1. A circumflex (^) outside a bracket expression anchors the expression or subexpression it begins to the beginning of a string; such an expression or subexpression can match only a sequence starting at the first character of a string. For example, the EREs ^ab and (^ab) match ab in the string abcdef, but fail to match in the string cdefab, and the ERE a^b is valid, but can never match because the a prevents the expression ^b from matching starting at the first character.

2. A dollar sign ($) outside a bracket expression anchors the expression or subexpression it ends to the end of a string; such an expression or subexpression can match only a sequence ending at the last character of a string. For example, the EREs ef$ and (ef$) match ef in the string abcdef, but fail to match in the string cdefab, and the ERE e$f is valid, but can never match because the f prevents the expression e$ from matching ending at the last character.

7.5 Regular Expression Grammar

Grammars describing the syntax of both basic and extended regular expressions are presented in this section. The grammar takes precedence over the text. See the **XCU** specification, **Section 1.8**, **Grammar Conventions**.

7.5.1 BRE/ERE Grammar Lexical Conventions

The lexical conventions for regular expressions are as described in this section.

Except as noted, the longest possible token or delimiter beginning at a given point will be recognised.

The following tokens will be processed (in addition to those string constants shown in the grammar):

COLL_ELEM — Any single-character collating element, unless it is a META_CHAR.

BACKREF — Applicable only to basic regular expressions. The character string consisting of "\" followed by a single-digit numeral, 1 to 9.

DUP_COUNT — Represents a numeric constant. It is an integer in the range 0 ≤ DUP_COUNT ≤ {RE_DUP_MAX}. This token will only be recognised when the context of the grammar requires it. At all other times, digits not preceded by "\" will be treated as ORD_CHAR.

META_CHAR — One of the characters:

- ˆ when found first in a bracket expression
- – when found anywhere but first (after an initial "ˆ", if any) or last in a bracket expression, or as the ending range point in a range expression
-] when found anywhere but first (after an initial "ˆ" if any) in a bracket expression.

L_ANCHOR — Applicable only to basic regular expressions. The character "ˆ" when it appears as the first character of a basic regular expression and when not QUOTED_CHAR. The "ˆ" may be recognised as an anchor elsewhere; see Section 7.3.8 on page 108.

ORD_CHAR — A character, other than one of the special characters in SPEC_CHAR.

QUOTED_CHAR — In a BRE, one of the character sequences:

```
\ˆ   \.   \*   \[   \$   \\
```

In an ERE, one of the character sequences:

```
\ˆ   \.   \[   \$   \(   \)   \|
\*   \+   \?   \{   \\
```

R_ANCHOR — (Applicable only to basic regular expressions.) The character "$" when it appears as the last character of a basic regular expression and when not QUOTED_CHAR. The "$" may be recognised as an anchor elsewhere; see Section 7.3.8 on page 108.

SPEC_CHAR — For basic regular expressions, will be one of the following special characters:

. anywhere outside bracket expressions

\ anywhere outside bracket expressions

[anywhere outside bracket expressions

ˆ when used as an anchor (see Section 7.3.8 on page 108) or when first in a bracket expression

$ when used as an anchor

* anywhere except: first in an entire RE; anywhere in a bracket expression; directly following \(; directly following an anchoring "ˆ".

For extended regular expressions, will be one of the following special characters found anywhere outside bracket expressions:

```
ˆ  .  [  $  (  )  |  *  +  ?  {  \
```

(The close-parenthesis is considered special in this context only if matched with a preceding open-parenthesis.)

7.5.2 RE and Bracket Expression Grammar

This section presents the grammar for basic regular expressions, including the bracket expression grammar that is common to both BREs and EREs.

```
%token    ORD_CHAR QUOTED_CHAR DUP_COUNT

%token    BACKREF L_ANCHOR R_ANCHOR

%token    Back_open_paren  Back_close_paren
/*          ’\(’             ’\)’        */

%token    Back_open_brace  Back_close_brace
/*          ’\{’             ’\}’         */

/* The following tokens are for the Bracket Expression
   grammar common to both REs and EREs. */

%token    COLL_ELEM META_CHAR

%token    Open_equal Equal_close Open_dot Dot_close Open_colon Colon_close
/*           ’[=’       ’=]’        ’[.’     ’.]’      ’[:’       ’:]’  */

%token    class_name
/* class_name is a keyword to the LC_CTYPE locale category */
/* (representing a character class) in the current locale  */
/* and is only recognised between [: and :] */

%start    basic_reg_exp
%%

/* --------------------------------------------
   Basic Regular Expression
   --------------------------------------------
*/
basic_reg_exp  :          RE_expression
               | L_ANCHOR
               |                       R_ANCHOR
               | L_ANCHOR              R_ANCHOR
               | L_ANCHOR RE_expression
               |          RE_expression R_ANCHOR
               | L_ANCHOR RE_expression R_ANCHOR
               ;
RE_expression  :               simple_RE
               | RE_expression simple_RE
               ;
simple_RE      : nondupl_RE
               | nondupl_RE RE_dupl_symbol
               ;
nondupl_RE     : one_character_RE
               | Back_open_paren RE_expression Back_close_paren
               | Back_open_paren Back_close_paren
               | BACKREF
               ;
one_character_RE : ORD_CHAR
               | QUOTED_CHAR
               | ’.’
               | bracket_expression
               ;
RE_dupl_symbol : ’*’
               | Back_open_brace DUP_COUNT               Back_close_brace
               | Back_open_brace DUP_COUNT ’,’           Back_close_brace
               | Back_open_brace DUP_COUNT ’,’ DUP_COUNT Back_close_brace
               ;

/* --------------------------------------------
   Bracket Expression
   -------------------------------------------
*/
bracket_expression : ’[’ matching_list    ’]’
               | ’[’ nonmatching_list ’]’
               ;
matching_list  : bracket_list
               ;
nonmatching_list : ’ˆ’ bracket_list
               ;
bracket_list   : follow_list
               | follow_list ’-’
               ;
follow_list    :             expression_term
               | follow_list expression_term
               ;
expression_term : single_expression
               | range_expression
               ;
single_expression : end_range
               | character_class
               | equivalence_class
               ;
range_expression : start_range end_range
               | start_range ’-’
               ;
start_range    : end_range ’-’
```

```
                ;
end_range       : COLL_ELEM
                | collating_symbol
                ;
collating_symbol : Open_dot COLL_ELEM Dot_close
                | Open_dot META_CHAR Dot_close
                ;
equivalence_class : Open_equal COLL_ELEM Equal_close
                ;
character_class : Open_colon class_name Colon_close
                ;
```

The BRE grammar does not permit L_ANCHOR or R_ANCHOR inside \(and \) (which implies that ^ and $ are ordinary characters). This reflects the semantic limits on the application, as noted in Section 7.3.8 on page 108. Implementations are permitted to extend the language to interpret "^" and "$" as anchors in these locations, and as such, portable applications cannot use unescaped "^" and "$" in positions inside \(and \) that might be interpreted as anchors.

7.5.3 ERE Grammar

This section presents the grammar for extended regular expressions, excluding the bracket expression grammar.

Note: The bracket expression grammar and the associated **%token** lines are identical between BREs and EREs. It has been omitted from the ERE section to avoid unnecessary editorial duplication.

```
%token  ORD_CHAR QUOTED_CHAR DUP_COUNT
%start  extended_reg_exp
%%

/* --------------------------------------------
   Extended Regular Expression
   --------------------------------------------
*/
extended_reg_exp   :                          ERE_branch
                   | extended_reg_exp '|' ERE_branch
                   ;
ERE_branch         :              ERE_expression
                   | ERE_branch ERE_expression
                   ;
ERE_expression     : one_character_ERE
                   | '^'
                   | '$'
                   | '(' extended_reg_exp ')'
                   | ERE_expression ERE_dupl_symbol
                   ;
one_character_ERE  : ORD_CHAR
                   | QUOTED_CHAR
                   | '.'
                   | bracket_expression
                   ;
ERE_dupl_symbol    : '*'
                   | '+'
                   | '?'
                   | ’{’ DUP_COUNT               ’}’
                   | ’{’ DUP_COUNT ’,’           ’}’
                   | ’{’ DUP_COUNT ’,’ DUP_COUNT ’}’
                   ;
```

The ERE grammar does not permit several constructs that previous sections specify as having undefined results:

- ORD_CHAR preceded by "\"
- one or more ERE_dupl_symbols appearing first in an ERE, or immediately following "|", "ˆ" or "("
- "{" not part of a valid ERE_dupl_symbol
- "|" appearing first or last in an ERE, or immediately following "|" or "(", or immediately preceding ")".

Implementations are permitted to extend the language to allow these. Portable applications cannot use such constructs.

Chapter 8

Directory Structure and Devices

8.1 Directory Structure and Files

The following directories exist on conforming systems and must be used as described. Portable applications cannot assume the ability to create files in any of these directories.

/ The root directory.

EX **/dev** Contains **/dev/console**, **/dev/null** and **/dev/tty**, described below.

The following directory exists on conforming systems and is used as described.

/tmp A directory made available for programs that need a place to create temporary files. Applications are allowed to create files in this directory, but cannot assume that such files are preserved between invocations of the application.

The **/tmp** directory is defined to accommodate historical applications that assume its availability. Applications are encouraged to use the contents of *TMPDIR* for creating temporary files rather than the specific name **/tmp**. See *tempnam*( ) in the **XSH** specification.

The following files exist on conforming systems and are both readable and writable.

/dev/null An infinite data source and data sink. Data written to **/dev/null** is discarded. Reads from **/dev/null** always return end-of-file (EOF).

/dev/tty In each process, a synonym for the controlling terminal associated with the process group of that process, if any. It is useful for programs or shell procedures that wish to be sure of writing messages to or reading data from the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

EX The following file exists on conforming systems and need not be readable or writable:

/dev/console The **/dev/console** file is a generic name given to the system console. It is usually linked to a particular machine-dependent special file. It provides a basic I/O interface to the system console.

8.2 Output Devices and Terminal Types

The utilities in the **XCU** specification historically have been implemented on a wide range of terminal types, but a conforming implementation need not support all features of all utilities on every conceivable terminal. This specification set states which features are optional for certain classes of terminals in the individual utility description sections. The implementation will document which terminal types it supports and which of these features and utilities are not supported by each terminal.

When a feature or utility is not supported on a specific terminal type, as allowed by this specification set, and the implementation considers such a condition to be an error preventing use of the feature or utility, the implementation will indicate such conditions through diagnostic messages or exit status values or both (as appropriate to the specific utility description) that inform the user that the terminal type lacks the appropriate capability.

This specification set uses a notational convention based on historical practice that identifies some of the control characters defined in Section 4.1 on page 43 in a manner easily remembered by users on many terminals. The correspondence between this ‘‘control–*char*’’ notation and the actual control characters is shown in the following table. When this specification set refers to a character by its control– name, it is referring to the actual control character shown in the Value column of the table, which is not necessarily the exact control key sequence on all terminals. Some terminals have keyboards that do not allow the direct transmission of all the non-alphanumeric characters shown. In such cases, the system documentation will describe which data sequences transmitted by the terminal are interpreted by the system as representing the special characters.

Name	**Value**	**Name**	**Value**	**Name**	**Value**
control-A	<SOH>	control-L	<FF>	control-W	<ETB>
control-B	<STX>	control-M	<CR>	control-X	<CAN>
control-C	<ETX>	control-N	<SO>	control-Y	<EM>
control-D	<EOT>	control-O	<SI>	control-Z	<SUB>
control-E	<ENQ>	control-P	<DLE>	control-[	<ESC>
control-F	<ACK>	control-Q	<DC1>	control-\	<FS>
control-G	<BEL>	control-R	<DC2>	control-]	<GS>
control-H	<BS>	control-S	<DC3>	control-^	<RS>
control-I	<HT>	control-T	<DC4>	control-_	<US>
control-J	<LF>	control-U	<NAK>	control-?	<DEL>
control-K	<VT>	control-V	<SYN>		

Table 8-1 Control Character Names

Note: The notation uses upper-case letters for arbitrary editorial reasons. There is no implication that the keystrokes represent control-shift-letter sequences.

Chapter 9

General Terminal Interface

This chapter describes a general terminal interface that is provided to control asynchronous communications ports. It is implementation-dependent whether it supports network connections or synchronous ports or both.

9.1 Interface Characteristics

9.1.1 Opening a Terminal Device File

When a terminal device file is opened, it normally causes the thread to wait until a connection is established. In practice, application programs seldom open these files; they are opened by special programs and become an application's standard input, output and error files.

As described in *open*( ), opening a terminal device file with the O_NONBLOCK flag clear causes the thread to block until the terminal device is ready and available. If CLOCAL mode is not set, this means blocking until a connection is established. If CLOCAL mode is set in the terminal, or the O_NONBLOCK flag is specified in the *open*( ), the *open*( ) function returns a file descriptor without waiting for a connection to be established.

9.1.2 Process Groups

A terminal may have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters, as discussed in Section 9.1.9 on page 123.

A command interpreter process supporting job control can allocate the terminal to different jobs, or process groups, by placing related processes in a single process group and associating this process group with the terminal. A terminal's foreground process group may be set or examined by a process, assuming the permission requirements are met; see *tcgetpgrp*( ) and *tcsetpgrp*( ). The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the current process group; see Section 9.1.4 on page 120.

When there is no longer any process whose process ID or process group ID matches the process group ID of the foreground process group, the terminal will have no foreground process group. It is unspecified whether the terminal has a foreground process group when there is a process whose process ID matches the foreground process ID, but whose process group ID does not. No actions defined in this specification set, other than allocation of a controlling terminal or a successful call to *tcsetpgrp*( ), will cause a process group to become the foreground process group of the terminal.

9.1.3 The Controlling Terminal

A terminal may belong to a process as its controlling terminal. Each process of a session that has a controlling terminal has the same controlling terminal. A terminal may be the controlling terminal for at most one session. The controlling terminal for a session is allocated by the session leader in an implementation-dependent manner. If a session leader has no controlling terminal, and opens a terminal device file that is not already associated with a session without using the O_NOCTTY option (see *open*( )), it is implementation-dependent whether the terminal becomes the controlling terminal of the session leader. If a process which is not a session leader opens a terminal file, or the O_NOCTTY option is used on *open*( ), then that terminal does not become the controlling terminal of the calling process. When a controlling terminal becomes associated with a session, its foreground process group is set to the process group of the session leader.

The controlling terminal is inherited by a child process during a *fork*( ) function call. A process relinquishes its controlling terminal when it creates a new session with the *setsid*( ) function; other processes remaining in the old session that had this terminal as their controlling terminal continue to have it. Upon the close of the last file descriptor in the system (whether or not it is in the current session) associated with the controlling terminal, it is unspecified whether all processes that had that terminal as their controlling terminal cease to have any controlling terminal. Whether and how a session leader can reacquire a controlling terminal after the controlling terminal has been relinquished in this fashion is unspecified. A process does not relinquish its controlling terminal simply by closing all of its file descriptors associated with the controlling terminal if other processes continue to have it open.

When a controlling process terminates, the controlling terminal is dissociated from the current session, allowing it to be acquired by a new session leader. Subsequent access to the terminal by other processes in the earlier session may be denied, with attempts to access the terminal treated as if a modem disconnect had been sensed.

9.1.4 Terminal Access Control

If a process is in the foreground process group of its controlling terminal, read operations are allowed, as described in Section 9.1.5. Any attempts by a process in a background process group to read from its controlling terminal cause its process group to be sent a SIGTTIN signal unless one of the following special cases applies: if the reading process is ignoring or blocking the SIGTTIN signal, or if the process group of the reading process is orphaned, the *read*( ) returns −1, with *errno* set to [EIO] and no signal is sent. The default action of the SIGTTIN signal is to stop the process to which it is sent. See **<signal.h>**.

If a process is in the foreground process group of its controlling terminal, write operations are allowed as described in Section 9.1.8 on page 122. Attempts by a process in a background process group to write to its controlling terminal will cause the process group to be sent a SIGTTOU signal unless one of the following special cases applies: if TOSTOP is not set, or if TOSTOP is set and the process is ignoring or blocking the SIGTTOU signal, the process is allowed to write to the terminal and the SIGTTOU signal is not sent. If TOSTOP is set, and the process group of the writing process is orphaned, and the writing process is not ignoring or blocking the SIGTTOU signal, the *write*( ) returns −1, with *errno* set to [EIO] and no signal is sent.

Certain calls that set terminal parameters are treated in the same fashion as *write*( ), except that TOSTOP is ignored; that is, the effect is identical to that of terminal writes when TOSTOP is set (see Section 9.2.5 on page 129, *tcdrain*( ), *tcflow*( ), *tcflush*( ), *tcsendbreak*( ) and *tcsetattr*( )).

9.1.5 Input Processing and Reading Data

A terminal device associated with a terminal device file may operate in full-duplex mode, so that data may arrive even while output is occurring. Each terminal device file has an *input queue*, associated with it, into which incoming data is stored by the system before being read by a process. The system may impose a limit, {MAX_INPUT}, on the number of bytes that may be stored in the input queue. The behaviour of the system when this limit is exceeded is implementation-dependent.

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or non-canonical mode. These modes are described in Section 9.1.6 on page 121 and Section 9.1.7 on page 121. Additionally, input characters are processed according to the **c_iflag** (see Section 9.2.2 on page 125) and **c_lflag** (see Section 9.2.5 on page 129) fields.

Such processing can include *echoing*, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. This is useful for terminals that can operate in full-duplex mode.

The manner in which data is provided to a process reading from a terminal device file is dependent on whether the terminal file is in canonical or non-canonical mode, and on whether or not the O_NONBLOCK flag is set by *open*( ) or *fcntrl*( ).

If the O_NONBLOCK flag is clear, then the read request is blocked until data is available or a signal has been received. If the O_NONBLOCK flag is set, then the read request is completed, without blocking, in one of three ways:

1. If there is enough data available to satisfy the entire request, the *read*( ) completes successfully and returns the number of bytes read.

2. If there is not enough data available to satisfy the entire request, the *read*( ) completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.

3. If there is no data available, the *read*( ) returns −1, with *errno* set to [EAGAIN].

When data is available depends on whether the input processing mode is canonical or non-canonical. The following sections, Section 9.1.6 and Section 9.1.7 describe each of these input processing modes.

9.1.6 Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a newline character (NL), an end-of-file character (EOF), or an end-of-line (EOL) character. See Section 9.1.9 on page 123 for more information on EOF and EOL. This means that a read request will not return until an entire line has been typed or a signal has been received. Also, no matter how many bytes are requested in the *read*( ) call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of bytes, even one, may be requested in a *read*( ) without losing information.

If {MAX_CANON} is defined for this terminal device, it is a limit on the number of bytes in a line. The behaviour of the system when this limit is exceeded is implementation-dependent. If {MAX_CANON} is not defined, there is no such limit; see *pathconf*( ).

Erase and kill processing occur when either of two special characters, the ERASE and KILL characters (see Section 9.1.9 on page 123), is received. This processing affects data in the input queue that has not yet been delimited by a newline (NL), EOF or EOL character. This un-delimited data makes up the current line. The ERASE character deletes the last character in the current line, if there is one. The KILL character deletes all data in the current line, if there are any. The ERASE and KILL characters have no effect if there is no data in the current line. The ERASE and KILL characters themselves are not placed in the input queue.

9.1.7 Non-canonical Mode Input Processing

In non-canonical mode input processing, input bytes are not assembled into lines, and erase and kill processing does not occur. The values of the MIN and TIME members of the **c_cc** array are used to determine how to process the bytes received. The ISO POSIX-1 standard does not specify whether the setting of O_NONBLOCK takes precedence over MIN or TIME settings. Therefore, if O_NONBLOCK is set, *read*( ) may return immediately, regardless of the setting of MIN or TIME. Also, if no data is available, *read*( ) may either return 0, or return −1 with *errno* set to [EAGAIN].

MIN represents the minimum number of bytes that should be received when the *read*( ) function returns successfully. TIME is a timer of 0.1 second granularity that is used to time out bursty and short-term data transmissions. If MIN is greater than {MAX_INPUT}, the response to the request is undefined. The four possible values for MIN and TIME and their interactions are described below.

Case A: MIN > 0, TIME > 0

In this case TIME serves as an inter-byte timer and is activated after the first byte is received. Since it is an inter-byte timer, it is reset after a byte is received. The interaction between MIN and TIME is as follows. As soon as one byte is received, the inter-byte timer is started. If MIN bytes are received before the inter-byte timer expires (remember that the timer is reset upon receipt of each byte), the read is satisfied. If the timer expires before MIN bytes are received, the characters received to that point are returned to the user. Note that if TIME expires at least one byte is returned because the timer would not have been enabled unless a byte was received. In this case (MIN > 0, TIME > 0) the read blocks until the MIN and TIME mechanisms are activated by the receipt of the first byte, or a signal is received. If the data is in the buffer at the time of the *read*( ), the result will be as if the data has been received immediately after the *read*( ).

Case B: MIN > 0, TIME = 0

In this case, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read is not satisfied until MIN bytes are received (that is, the pending read blocks until MIN bytes are received), or a signal is received. A program that uses this case to read record-based terminal I/O may block indefinitely in the read operation.

Case C: MIN = 0, TIME > 0

In this case, since MIN = 0, TIME no longer represents an inter-byte timer. It now serves as a read timer that is activated as soon as the *read*( ) function is processed. A read is satisfied as soon as a single byte is received or the read timer expires. Note that in this case if the timer expires, no bytes are returned. If the timer does not expire, the only way the read can be satisfied is if a byte is received. In this case the read will not block indefinitely waiting for a byte; if no byte is received within TIME*0.1 seconds after the read is initiated, the *read*( ) returns a value of zero, having read no data. If the data is in the buffer at the time of the *read*( ), the timer is started as if the data has been received immediately after the *read*( ).

Case D: MIN = 0, TIME = 0

The minimum of either the number of bytes requested or the number of bytes currently available is returned without waiting for more bytes to be input. If no characters are available, *read*( ) returns a value of zero, having read no data.

9.1.8 Writing Data and Output Processing

When a process writes one or more bytes to a terminal device file, they are processed according to the **c_oflag** field (see Section 9.2.3 on page 126). The implementation may provide a buffering mechanism; as such, when a call to *write*( ) completes, all of the bytes written have been scheduled for transmission to the device, but the transmission will not necessarily have completed. See *write*( ) for the effects of O_NONBLOCK on *write*( ).

9.1.9 Special Characters

Certain characters have special functions on input or output or both. These functions are summarised as follows:

INTR Special character on input, which is recognised if the ISIG flag is set. Generates a SIGINT signal which is sent to all processes in the foreground process group for which the terminal is the controlling terminal. If ISIG is set, the INTR character is discarded when processed.

QUIT Special character on input, which is recognised if the ISIG flag is set. Generates a SIGQUIT signal which is sent to all processes in the foreground process group for which the terminal is the controlling terminal. If ISIG is set, the QUIT character is discarded when processed.

ERASE Special character on input, which is recognised if the ICANON flag is set. Erases the last character in the current line; see Section 9.1.6 on page 121. It will not erase beyond the start of a line, as delimited by an NL, EOF or EOL character. If ICANON is set, the ERASE character is discarded when processed.

KILL Special character on input, which is recognised if the ICANON flag is set. Deletes the entire line, as delimited by an NL, EOF or EOL character. If ICANON is set, the KILL character is discarded when processed.

EOF Special character on input, which is recognised if the ICANON flag is set. When received, all the bytes waiting to be read are immediately passed to the process without waiting for a newline, and the EOF is discarded. Thus, if there are no bytes waiting (that is, the EOF occurred at the beginning of a line), a byte count of zero is returned from the *read*( ), representing an end-of-file indication. If ICANON is set, the EOF character is discarded when processed.

NL Special character on input, which is recognised if the ICANON flag is set. It is the line delimiter newline. It cannot be changed.

EOL Special character on input, which is recognised if the ICANON flag is set. It is an additional line delimiter, like NL.

SUSP If the ISIG flag is set, receipt of the SUSP character causes a SIGTSTP signal to be sent to all processes in the foreground process group for which the terminal is the controlling terminal, and the SUSP character is discarded when processed.

STOP Special character on both input and output, which is recognised if the IXON (output control) or IXOFF (input control) flag is set. Can be used to suspend output temporarily. It is useful with CRT terminals to prevent output from disappearing before it can be read. If IXON is set, the STOP character is discarded when processed.

START Special character on both input and output, which is recognised if the IXON (output control) or IXOFF (input control) flag is set. Can be used to resume output that has been suspended by a STOP character. If IXON is set, the START character is discarded when processed.

CR Special character on input, which is recognised if the ICANON flag is set; it is the carriage-return character. When ICANON and ICRNL are set and IGNCR is not set, this character is translated into an NL, and has the same effect as an NL character.

The NL and CR characters cannot be changed. It is implementation-dependent whether the START and STOP characters can be changed. The values for INTR, QUIT, ERASE, KILL, EOF, EOL and SUSP are changeable to suit individual tastes. Special character functions associated with changeable special control characters can be disabled individually.

FIPS

If two or more special characters have the same value, the function performed when that character is received is undefined.

A special character is recognised not only by its value, but also by its context; for example, an implementation may support multi-byte sequences that have a meaning different from the meaning of the bytes when considered individually. Implementations may also support additional single-byte functions. These implementation-dependent multi-byte or single-byte functions are recognised only if the IEXTEN flag is set; otherwise, data is received without interpretation, except as required to recognise the special characters defined in this section.

EX If IEXTEN is set, the ERASE, KILL and EOF characters can be escaped by a preceding \ character, in which case no special function occurs.

9.1.10 Modem Disconnect

If a modem disconnect is detected by the terminal interface for a controlling terminal, and if CLOCAL is not set in the **c_cflag** field for the terminal (see Section 9.2.4 on page 128), the SIGHUP signal is sent to the controlling process for which the terminal is the controlling terminal. Unless other arrangements have been made, this causes the controlling process to terminate (see *exit*( )). Any subsequent read from the terminal device returns the value of zero, indicating end-of-file. (See *read*( ).) Thus, processes that read a terminal file and test for end-of-file can terminate appropriately after a disconnect. If the EIO condition as specified in *read*( ) also exists, it is unspecified whether on EOF condition or the [EIO] is returned. Any subsequent *write*( ) to the terminal device returns –1, with *errno* set to [EIO], until the device is closed.

9.1.11 Closing a Terminal Device File

The last process to close a terminal device file causes any output to be sent to the device and any input to be discarded. If HUPCL is set in the control modes and the communications port supports a disconnect function, the terminal device will perform a disconnect.

9.2 Parameters that Can be Set

9.2.1 The termios Structure

Routines that need to control certain terminal I/O characteristics do so by using the **termios** structure as defined in the header **<termios.h>**. The members of this structure include (but are not limited to):

Member Type	**Array Size**	**Member Name**	**Description**
tcflag_t		**c_iflag**	Input modes.
tcflag_t		**c_oflag**	Output modes.
tcflag_t		**c_cflag**	Control modes.
tcflag_t		**c_lflag**	Local modes.
cc_t	NCCS	**c_cc**[ ]	Control characters.

The types **tcflag_t** and **cc_t** are defined in the header **<termios.h>**. They are unsigned integral types.

9.2.2 Input Modes

Values of the **c_iflag** field describe the basic terminal input control, and are composed of the bitwise inclusive OR of the masks shown, which will be bitwise distinct. The mask name symbols in this table are defined in **<termios.h>**:

	Mask Name	**Description**
	BRKINT	Signal interrupt on break.
	ICRNL	Map CR to NL on input.
	IGNBRK	Ignore break condition.
	IGNCR	Ignore CR.
	IGNPAR	Ignore characters with parity errors.
	INLCR	Map NL to CR on input.
	INPCK	Enable input parity check.
	ISTRIP	Strip character.
EX	IUCLC	Map upper case to lower case on input. (**LEGACY**)
EX	IXANY	Enable any character to restart output.
	IXOFF	Enable start/stop input control.
	IXON	Enable start/stop output control.
	PARMRK	Mark parity errors.

In the context of asynchronous serial data transmission, a break condition is defined as a sequence of zero-valued bits that continues for more than the time to send one byte. The entire sequence of zero-valued bits is interpreted as a single break condition, even if it continues for a time equivalent to more than one byte. In contexts other than asynchronous serial data transmission, the definition of a break condition is implementation-dependent.

If IGNBRK is set, a break condition detected on input is ignored that is, not put on the input queue and therefore not read by any process. If IGNBRK is not set and BRKINT is set, the break condition will flush the input and output queues, and if the terminal is the controlling terminal of a foreground process group, the break condition will generate a single SIGINT signal to that foreground process group. If neither IGNBRK nor BRKINT is set, a break condition is read as a single 0x00, or if PARMRK is set, as 0xff 0x00 0x00.

If IGNPAR is set, a byte with a framing or parity error (other than break) is ignored.

If PARMRK is set, and IGNPAR is not set, a byte with a framing or parity error (other than break) is given to the application as the three-byte sequence 0xff 0x00 X, where 0xff 0x00 is a two-byte flag preceding each sequence and X is the data of the byte received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid byte of 0xff is given to the application as 0xff 0xff. If neither PARMRK nor IGNPAR is set, a framing or parity error (other than break) is given to the application as a single byte 0x00.

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled, allowing output parity generation without input parity errors. Note that whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled (see Section 9.2.4 on page 128). If parity detection is enabled but input parity checking is disabled, the hardware to which the terminal is connected will recognise the parity bit but the terminal special file will not check whether or not this bit is correctly set.

If ISTRIP is set, valid input bytes are first stripped to seven bits, otherwise all eight bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). If IGNCR is not set and ICRNL is set, a received CR character is translated into an NL character.

EX If IUCLC is set, upper- to lower-case mappings are performed on the received character. In locales other than the POSIX locale, the mapping is unspecified. (**LEGACY**)

If IXANY is set, any input character will restart output that has been suspended.

If IXON is set, start/stop output control is enabled. A received STOP character suspends output and a received START character restarts output. When IXON is set, START and STOP characters are not read, but merely perform flow control functions. When IXON is not set, the START and STOP characters are read.

If IXOFF is set, start/stop input control is enabled. The system transmits STOP characters, which are intended to cause the terminal device to stop transmitting data, as needed to prevent the input queue from overflowing and causing undefined behaviour, and transmits START characters, which are intended to cause the terminal device to resume transmitting data, as soon as the device can continue transmitting data without risk of overflowing the input queue. The precise conditions under which STOP and START characters are transmitted are implementation-dependent.

The initial input control value after *open*( ) is implementation-dependent.

9.2.3 Output Modes

The **c_oflag** field specifies the terminal interface's treatment of output, and is composed of the bitwise inclusive OR of the masks shown, which will be bitwise distinct. The mask name symbols in this table are defined in **<termios.h>**:

Mask Name	Description
OPOST	Perform output processing.
EX OLCUC	Map lower case to upper on output. (**LEGACY**)
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL, else NUL.
NLDLY	Select newline delays:
NL0	Newline character type 0
NL1	Newline character type 1.
CRDLY	Select carriage-return delays:
CR0	Carriage-return delay type 0
CR1	Carriage-return delay type 1
CR2	Carriage-return delay type 2
CR3	Carriage-return delay type 3.
TABDLY	Select horizontal-tab delays:
TAB0	Horizontal-tab delay type 0
TAB1	Horizontal-tab delay type 1
TAB2	Horizontal-tab delay type 2.
TAB3	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	Backspace-delay type 0
BS1	Backspace-delay type 1.
VTDLY	Select vertical-tab delays:
VT0	Vertical-tab delay type 0
VT1	Vertical-tab delay type 1.
FFDLY	Select form-feed delays:
FF0	Form-feed delay type 0
FF1	Form-feed delay type 1.

If OPOST is set, output data is post-processed as described below, so that lines of text are modified to appear appropriately on the terminal device; otherwise, characters are transmitted without change.

EX If OLCUC is set, lower- to upper-case mappings are performed on the characters before they are transmitted. In locales other than the POSIX locale, the mapping is unspecified. (**LEGACY**)

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the newline delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value after *open*( ) is implementation-dependent.

9.2.4 Control Modes

The **c_cflag** field describes the hardware control of the terminal, and is composed of the bitwise inclusive OR of the masks shown, which will be bitwise distinct. The mask name symbols in this table are defined in **<termios.h>**; not all values specified are required to be supported by the underlying hardware:

Mask Name	Description
CLOCAL	Ignore modem status lines.
CREAD	Enable receiver.
CSIZE	Number of bits transmitted or received per byte:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits.
CSTOPB	Send two stop bits, else one.
HUPCL	Hang up on last close.
PARENB	Parity enable.
PARODD	Odd parity, else even.

In addition, the input and output baud rates are stored in the **termios** structure. The following values are supported:

Name	Description	Name	Description
B0	Hang up	B600	600 baud
B50	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134.5 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19200 baud
B300	300 baud	B38400	38400 baud

The following interfaces are provided for getting and setting the values of the input and output baud rates in the **termios** structure: *cfgetispeed*( ), *cfgetospeed*( ), *cfsetispeed*( ) and *cfsetospeed*( ). The effects on the terminal device do not become effective and not all errors are detected until the *tcsetattr*( ) function is successfully called.

The CSIZE bits specify the number of transmitted or received bits per byte. If ISTRIP is not set, the value of all the other bits is unspecified. If ISTRIP is set, the value of all but the 7 low-order bits is zero, but the value of any other bits beyond CSIZE is unspecified when read. CSIZE does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stop bits are normally used.

If CREAD is set, the receiver is enabled. Otherwise, no characters will be received.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each byte. If parity is enabled, PARODD specifies odd parity if set, otherwise even parity is used.

If HUPCL is set, the modem control lines for the port are lowered when the last process with the port open closes the port or the process terminates. The modem connection is broken.

If CLOCAL is set, a connection does not depend on the state of the modem status lines. If CLOCAL is clear, the modem status lines are monitored.

Under normal circumstances, a call to the *open*( ) function waits for the modem connection to complete. However, if the O_NONBLOCK flag is set (see *open*( )) or if CLOCAL has been set, the *open*( ) function returns immediately without waiting for the connection.

If the object for which the control modes are set is not an asynchronous serial connection, some of the modes may be ignored; for example, if an attempt is made to set the baud rate on a network connection to a terminal on another host, the baud rate may or may not be set on the connection between that terminal and the machine to which it is directly connected.

The initial hardware control value after *open*( ) is implementation-dependent.

9.2.5 Local Modes

The **c_lflag** field of the argument structure is used to control various functions. It is composed of the bitwise inclusive OR of the masks shown, which will be bitwise distinct. The mask name symbols in this table are defined in **<termios.h>**; not all values specified are required to be supported by the underlying hardware:

Mask Name	Description
ECHO	Enable echo.
ECHOE	Echo ERASE as an error correcting backspace.
ECHOK	Echo KILL.
ECHONL	Echo <newline>.
ICANON	Canonical input (erase and kill processing).
IEXTEN	Enable extended (implementation-dependent) functions.
ISIG	Enable signals.
NOFLSH	Disable flush after interrupt, quit or suspend.
TOSTOP	Send SIGTTOU for background output.
XCASE	Canonical upper/lower presentation. (**LEGACY**)

EX

If ECHO is set, input characters are echoed back to the terminal. If ECHO is clear, input characters are not echoed.

If ECHOE and ICANON are set, the ERASE character causes the terminal to erase, if possible, the last character in the current line from the display. If there were no character to erase, an implementation might echo an indication that this was the case, or do nothing.

If ECHOK and ICANON are set, the KILL character causes the terminal to erase the line from the display or echoes the newline character after the KILL character.

If ECHONL and ICANON are set, the newline character is echoed even if ECHO is not set.

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF and EOL, as described in Section 9.1.6 on page 121.

If ICANON is not set, read requests are satisfied directly from the input queue. A read is not satisfied until at least MIN bytes have been received or the timeout value TIME expired between bytes. The time value represents tenths of a second. See Section 9.1.7 on page 121 for more details.

If IEXTEN is set, implementation-dependent functions are recognised from the input data. It is implementation-dependent how IEXTEN being set interacts with ICANON, ISIG, IXON or IXOFF. If IEXTEN is not set, implementation-dependent functions are not recognised and the corresponding input characters are processed as described for ICANON, ISIG, IXON and IXOFF.

If ISIG is set, each input character is checked against the special control characters INTR, QUIT and SUSP. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT and SUSP characters is not done.

If TOSTOP is set, the signal SIGTTOU is sent to the process group of a process that tries to write to its controlling terminal if it is not in the foreground process group for that terminal. This signal, by default, stops the members of the process group. Otherwise, the output generated by that process is output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output, and the SIGTTOU signal is not sent.

EX If XCASE is set, canonical lower and canonical upper presentation are performed. In locales other than the POSIX locale, the effect is unspecified. (**LEGACY**)

The initial local control value after *open*( ) is implementation-dependent.

9.2.6 Special Control Characters

The special control characters values are defined by the array **c_cc**. The subscript name and description for each element in both canonical and non-canonical modes are as follows:

Subscript Usage		
Canonical Mode	**Non-canonical Mode**	**Description**
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character

The subscript values are unique, except that the VMIN and VTIME subscripts may have the same values as the VEOF and VEOL subscripts, respectively.

The number of elements in the **c_cc** array, NCCS, is unspecified.

Implementations that do not support changing the START and STOP characters may ignore the character values in the **c_cc** array indexed by the VSTART and VSTOP subscripts when *tcsetattr*( ) is called, but will return the value in use when *tcgetattr*( ) is called.

The initial values of all control characters are implementation-dependent.

If the value of one of the changeable special control characters (see Section 9.1.9 on page 123) is {_POSIX_VDISABLE}, that function is disabled; that is, no input data will be recognised as the disabled special character. If ICANON is not set, the value of {_POSIX_VDISABLE} has no special meaning for the VMIN and VTIME entries of the **c_cc** array.

Chapter 10

Utility Conventions

10.1 Utility Argument Syntax

This section describes the argument syntax of the standard utilities and introduces terminology used throughout this specification set for describing the arguments processed by the utilities.

Within this specification set, a special notation is used for describing the syntax of a utility's arguments. Unless otherwise noted, all utility descriptions use this notation, which is illustrated by this example (see the **XCU** specification, **Section 2.9.1**, **Simple Commands**):

```
utility_name[-a][-b][-coption_argument][-d|-e][-foption_argument][operand ...]
```

The notation used for the **SYNOPSIS** sections imposes requirements on the implementors of the standard utilities and provides a simple reference for the application developer or system user.

1. The utility in the example is named *utility_name*. It is followed by *options*, *option-arguments* and *operands*. The arguments that consist of hyphens and single letters or digits, such as **–a**, are known as *options* (or, historically, *flags*). Certain options are followed by an *option-argument*, as shown with [**–c** *option_argument*]. The arguments following the last options and option-arguments are named *operands*.

2. Option-arguments are sometimes shown separated from their options by blank characters, sometimes directly adjacent. This reflects the situation that in some cases an option-argument is included within the same argument string as the option; in most cases it is the next argument. The Utility Syntax Guidelines in Section 10.2 on page 136 require that the option be a separate argument from its option-argument, but there are some exceptions in this specification set to ensure continued operation of historical applications:

   a. If the **SYNOPSIS** of a standard utility shows a space character between an option and option-argument (as with [**–c** *option_argument*] in the example), a portable application must use separate arguments for that option and its option-argument.

   b. If a space character is not shown (as with [**–f***option_argument*] in the example), a portable application must place an option and its option-argument directly adjacent in the same argument string, without intervening blank characters.

   c. Notwithstanding the preceding requirements on portable applications, X/Open systems permit, but do not require, an application to specify options and option-arguments as separate arguments whether or not a space character is shown on the synopsis line, except in those cases (marked with the EX portability warning) where an option-argument is optional and no separation can be used.

   d. A standard utility may also be implemented to operate correctly when the required separation into multiple arguments is violated by a non-portable application.

In summary, the following table shows allowable combinations:

	SYNOPSIS Shows:		
	–a *arg*	–b*arg*	–c**[***arg***]**
Portable application must use:	–a *arg*	–b*arg*	n/a
System will support:	–a *arg*	–b*arg*	–c*arg* or –c
System may support:	–a*arg*	–b *arg*	

3. Options are usually listed in alphabetical order unless this would make the utility description more confusing. There are no implied relationships between the options based upon the order in which they appear, unless otherwise stated in the **OPTIONS** section, or unless the exception in Section 10.2 on page 136 guideline 11 applies. If an option that does not have option-arguments is repeated, the results are undefined, unless otherwise stated.

4. Frequently, names of parameters that require substitution by actual values are shown with embedded underscores. Alternatively, parameters are shown as follows:

   <*parameter name*>

   The angle brackets are used for the symbolic grouping of a phrase representing a single parameter and must never be included in data submitted to the utility.

5. When a utility has only a few permissible options, they are sometimes shown individually, as in the example. Utilities with many flags generally show all of the individual flags (that do not take option-arguments) grouped, as in:

   utility_name **[**-abcDxyz**][**-p *arg***][***operand***]**

   Utilities with very complex arguments may be shown as follows:

   utility_name **[**options**][***operands***]**

6. Unless otherwise specified, whenever an operand or option-argument is, or contains, a numeric value:

   - The number is interpreted as a decimal integer.
   - Numerals in the range 0 to 2 147 483 647 are syntactically recognised as numeric values.
   - When the utility description states that it accepts negative numbers as operands or option-arguments, numerals in the range –2 147 483 647 to 2 147 483 647 are syntactically recognised as numeric values.
   - Ranges greater than those listed here are allowed.

   This does not mean that all numbers within the allowable range are necessarily semantically correct. A standard utility that accepts an option-argument or operand that is to be interpreted as a number, and for which a range of values smaller than that shown above is permitted by the **XCU** specification, describes that smaller range along with the description of the option-argument or operand. If an error is generated, the utility's diagnostic message will indicate that the value is out of the supported range, not that it is syntactically incorrect.

   For example, the specification of *dd* obs=3000000000 would yield undefined behaviour for the application and could be a syntax error because the number 3 000 000 000 is outside of the range –2 147 483 647 to +2 147 483 647. On the other hand, *dd* obs=2000000000 may cause some error, such as ''blocksize too large'', rather than a syntax error.

7. Arguments or option-arguments enclosed in the **[** and **]** notation are optional and can be omitted. The **[** and **]** symbols must never be included in data submitted to the utility.

8. Arguments separated by the **|** vertical bar notation are mutually exclusive. The **|** symbols must never be included in data submitted to the utility. Alternatively, mutually exclusive options and operands may be listed with multiple synopsis lines. For example:

```
utility_name -d[-a][-c option_argument][operand...]

utility_name[-a][-b][operand...]
```

When multiple synopsis lines are given for a utility, it is an indication that the utility has mutually exclusive arguments. These mutually exclusive arguments alter the functionality of the utility so that only certain other arguments are valid in combination with one of the mutually exclusive arguments. Only one of the mutually exclusive arguments is allowed for invocation of the utility. Unless otherwise stated in an accompanying **OPTIONS** section, the relationships between arguments depicted in the **SYNOPSIS** sections are mandatory requirements placed on portable applications. The use of conflicting mutually exclusive arguments produces undefined results, unless a utility description specifies otherwise. When an option is shown without the **[ ]** brackets, it means that option is required for that version of the **SYNOPSIS**. However, it is not required to be the first argument, as shown in the example above, unless otherwise stated.

The use of *undefined* for conflicting argument usage and for repeated usage of the same option is meant to prevent portable applications from using conflicting arguments or repeated options, unless specifically allowed, as is the case with *ls* (which allows simultaneous, repeated use of the −**C**, −**l** and −**1** options). Many historical implementations will tolerate this usage, choosing either the first or the last applicable argument, and this tolerance may continue, but portable applications cannot rely upon it. (Other implementations may choose to print usage messages instead.)

The use of *undefined* for conflicting argument usage also allows an implementation to make reasonable extensions to utilities where the implementor considers mutually exclusive options according to the **XCU** specification to have a sensible meaning and result.

9. Ellipses (...) are used to denote that one or more occurrences of an option or operand are allowed. When an option or an operand followed by ellipses is enclosed in brackets, zero or more options or operands can be specified. The forms:

```
utility_name -f option_argument...[operand...]

utility_name [-g option_argument]...[operand...]
```

indicate that multiple occurrences of the option and its option-argument preceding the ellipses are valid, with semantics as indicated in the **OPTIONS** section of the utility. (See also Guideline 11 in Section 10.2 on page 136.) In the first example, each option-argument requires a preceding −**f** and at least one −**f** *option_argument* must be given.

The **XCU** specification does not define the result of a utility when an option-argument or operand is not followed by ellipses and the application specifies more than one of that option-argument or operand. This allows an implementation to define valid (although non-standard) behaviour for the utility when more than one such option or operand are specified.

10. When the synopsis line is too long to be printed on a single line in the **XCU** specification, the indented lines following the initial line are continuation lines. An actual use of the command would appear on a single logical line.

10.2 Utility Syntax Guidelines

The following guidelines are established for the naming of utilities and for the specification of options, option-arguments and operands. The *getopt*( ) function in the **XSH** specification assists utilities in handling options and operands that conform to these guidelines.

Operands and option-arguments can contain characters not specified in the portable character set.

The guidelines are intended to provide guidance to the authors of future utilities, such as those written specific to a local system or that are to be components of a larger application. Some of the standard utilities do not conform to all of these guidelines; in those cases, the **OPTIONS** sections describe the deviations.

Guideline 1: Utility names should be between two and nine characters, inclusive.

Guideline 2: Utility names should include lower-case letters (the **lower** character classification) and digits only from the portable character set.

Guidelines 1 and 2 are offered as guidance for locales using Latin alphabets. No recommendations are made by this specification set concerning utility naming in other locales.

In the **XCU** specification, **Section 2.9.1**, **Simple Commands**, it is further stated that a command used in the XSI Shell Command Language cannot be named with a trailing colon.

Guideline 3: Each option name should be a single alphanumeric character (the **alnum** character classification) from the portable character set.

Multi-digit options are not allowed. Instances of historical utilities that used them have been marked **LEGACY** in the **XCU** specification, with the numbers being changed from option names to option-arguments.

Guideline 4: All options should be preceded by the "–" delimiter character.

Guideline 5: Options without option-arguments should be accepted when grouped behind one "–" delimiter.

Guideline 6: Each option and option-argument should be a separate argument, except as noted in Section 10.1 on page 133, item (2).

Guideline 7: Option-arguments should not be optional.

Guideline 8: When multiple option-arguments are specified to follow a single option, they should be presented as a single argument, using commas within that argument or blank characters within that argument to separate them.

It is up to the utility to parse a comma-separated list itself because *getopt*( ) just returns a single string. This situation was retained so that certain historical utilities would not violate the guidelines. Applications preparing for international use should be aware of an occasional problem with comma-separated lists: in some locales, the comma is used as the radix character. Thus, if an application is preparing operands for a utility that expects a comma-separated list, it should avoid generating non-integer values through one of the means that is influenced by setting the *LC_NUMERIC* variable (such as *awk*, *bc*, *printf* or *printf*( )).

Guideline 9: All options should precede operands on the command line.

Guideline 10: The argument −− should be accepted as a delimiter indicating the end of options. Any following arguments should be treated as operands, even if they begin with the "−" character. The −− argument should not be used as an option or as an operand.

Applications calling any utility with a first operand starting with − should usually specify −−, as indicated by Guideline 10, to mark the end of the options. This is true even if the **SYNOPSIS** in the **XCU** specification does not specify any options; implementations may provide options as extensions to the **XCU** specification. The standard utilities that do not support Guideline 10 indicate that fact in the **OPTIONS** section of the utility description.

Guideline 11: The order of different options relative to one another should not matter, unless the options are documented as mutually exclusive and such an option is documented to override any incompatible options preceding it. If an option that has option-arguments is repeated, the option and option-argument combinations should be interpreted in the order specified on the command line.

The order of repeated options that also have option-arguments may be significant; therefore, such options are required to be interpreted in the order that they are specified. The *make* utility is an instance of a historical utility that uses repeated options in which the order is significant. Multiple files are specified by giving multiple instances of the −**f** option, for example:

```
make -f common_header -f specific_rules target
```

Guideline 12: The order of operands may matter and position-related interpretations should be determined on a utility-specific basis.

Guideline 13: For utilities that use operands to represent files to be opened for either reading or writing, the "−" operand should be used only to mean standard input (or standard output when it is clear from context that an output file is being specified).

Guideline 13 does not imply that all of the standard utilities automatically accept the operand "−" to mean standard input or output, nor does it specify the actions of the utility upon encountering multiple "−" operands. It simply says that, by default, "−" operands are not used for other purposes in the file reading or writing (but not when using *stat*( ), *unlink*( ), *touch*, and so forth) utilities. All information concerning actual treatment of the "−" operand is found in the individual utility sections.

The utilities in the **XCU** specification that claim conformance to these guidelines were written as if the term *should* imposed a specific requirement on their interface and applications and users can rely on the behaviour stated here; the Guidelines are rules for the standard utilities that claim conformance to them. On some systems, the utilities will accept usage in violation of these guidelines for backward compatibility as well as accepting the required form.

It is recommended that all future utilities and applications use these guidelines to enhance user portability. The fact that some historical utilities could not be changed (to avoid breaking existing applications) should not deter this future goal.