-- --

IEEE P1003.2 Draft 11.2 − September 1991

2.2.2.113 privilege: See *appropriate privileges* in 2.2.2.6. [POSIX.1 {8}]

2.2.2.114 process: An address space and single thread of control that executes within that address space, and its required system resources.

A process is created by another process issuing the POSIX.1 {8} *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. [POSIX.1 {8}]

The attributes of processes required by POSIX.2 form a subset of those in POSIX.1 {8}; see 2.9.1.

2.2.2.115 process group: A collection of processes that permits the signaling 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. [POSIX.1 {8}]

2.2.2.116 process group ID: The unique identifier representing a process group during its lifetime.

A process group ID is a positive integer that can be contained in a *pid_t*. It shall not be reused by the system until the process group lifetime ends. [POSIX.1 {8}]

2.2.2.117 process group leader: A process whose process ID is the same as its process group ID. [POSIX.1 {8}]

2.2.2.118 process ID: The unique identifier representing a process.

A process ID is a positive integer that can be contained in a *pid_t*. A process ID shall 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 shall not be reused by the system until the process group lifetime ends. A process that is not a system process shall not have a process ID of 1. [POSIX.1 {8}]

2.2.2.119 program: A prepared sequence of instructions to the system to accomplish a defined task.

The term *program* in POSIX.2 encompasses applications written in the Shell Command Language, complex utility input languages (for example, `awk`, `lex`, `sed`, etc.), and high-level languages.

2.2.2.120 read-only file system: A file system that has implementation-defined characteristics restricting modifications. [POSIX.1 {8}]

2.2.2.121 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* in 2.2.2.75. This value is subject to change during the process lifetime, as described in POSIX.1 {8} 4.2.2 [*setgid*()]. [POSIX.1 {8}]

2.2.2.122 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* in 2.2.2.154. This value is subject to change during the process lifetime, as described in POSIX.1 {8} 4.2.2 [*setuid*()]. [POSIX.1 {8}]

2.2.2.123 regular expression: A pattern (sequence of characters or symbols) 1
constructed according to the rules defined in 2.8. 1

2.2.2.124 regular file: A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. [POSIX.1 {8}]

2.2.2.125 relative pathname: See *pathname resolution* in 2.2.2.104. [POSIX.1 {8}]

2.2.2.126 root directory: A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash. [POSIX.1 {8}]

2.2.2.127 saved set-group-ID: An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, when the saved set-user-ID option is implemented, as described in POSIX.1 {8} 3.1.2 (*exec*) and 4.2.2 [*setgid*()]. [POSIX.1 {8}]

2.2.2.128 saved set-user-ID: An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, when the saved set-user-ID option is implemented, as described in POSIX.1 {8} 3.1.2 and 4.2.2 [*setuid*()]. [POSIX.1 {8}]

2.2.2.129 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 nonnegative, the value is related to a Coordinated Universal Time name according to the expression:

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

[POSIX.1 {8}]

2.2.2.130 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 POSIX.1 {8} 4.3.2 [*setsid*()]. Implementations that support the POSIX.1 {8} *setpgid*() function (see POSIX.1 {8} 4.3.3) can have multiple process groups in the same session. [POSIX.1 {8}]

2.2.2.131 session leader: A process that has created a session; see POSIX.1 {8} 4.3.2 [*setsid*()]. [POSIX.1 {8}]

2.2.2.132 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. [POSIX.1 {8}]

2.2.2.133 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.

2.2.2.134 Shell, The: The Shell Command Language Interpreter (see 4.56), a specific instance of a shell.

2.2.2.135 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 description of *simple command* in 3.9.1). Execution of a shell script causes a shell to execute the commands within the script. Alternately, 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.

2.2.2.136 signal: A mechanism by which a process 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. [POSIX.1 {8}]

2.2.2.137 single-quote: The character ‘‘'’’, also known as *apostrophe*.

2.2.2.138 slash: The character ‘‘/’’, also known as *solidus*.

2.2.2.139 source code: When dealing with the Shell Command Language, source code is input to the command language interpreter.

The term *shell script* is synonymous with this meaning.

When dealing with the C Language Bindings Option, source code is input to a C compiler conforming to the C Standard {7}.

When dealing with another ISO/IEC conforming language, source code is input to a compiler conforming to that ISO/IEC 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.

2.2.2.140 <space>: The character defined in 2.4 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 2.2.2.158.)

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

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

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

2.2.2.144 standard utilities: The utilities defined by this standard, in the Sections 4, 5, and 6, and Annex A, and Annex C, and in similar sections of utility definitions introduced in future revisions of, and supplements to, this standard.

2.2.2.145 stream: An ordered sequence of characters, as described by the C Standard {7}.

2.2.2.146 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’s effective group ID is included in or omitted from its list of supplementary group IDs is unspecified. [POSIX.1 {8}]

2.2.2.147 system: An implementation of this standard.

2.2.2.148 <tab>: The horizontal tab character.

2.2.2.149 terminal [terminal device]: A character special file that obeys the specifications of the POSIX.1 {8} General Terminal Interface. [POSIX.1 {8}]

2.2.2.150 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.

2.2.2.151 text file: A file that contains characters organized into one or more lines.

The lines shall not contain NUL characters and none shall exceed {LINE_MAX} bytes in length, including the <newline>. Although POSIX.1 {8} does not distinguish between text files and binary files (see the C Standard {7}), 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 Standard Input or Input Files subclauses.

2.2.2.152 tilde: The character ‘‘~’’.

2.2.2.153 user database: See Section 9 in POSIX.1 {8}.

2.2.2.154 user ID: A nonnegative integer, which can be contained in an object of type *uid_t*, that is used to identify a system user.

When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or an (optional) saved set-user-ID. [POSIX.1 {8}]

2.2.2.155 user name: A string that is used to identify a user, as described in POSIX.1 {8} 9.1. [POSIX.1 {8}]

2.2.2.156 utility: A program that can be called by name from a shell to perform a specific task, or related set of tasks.

This program shall either be an executable file, such as might be produced by a compiler/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 implementor of this standard, or acquired from an independent distributor. The term *utility* does not apply to the special built-in utilities provided as part of the shell command language; see 3.14. The system may implement certain utilities as shell functions (see 3.9.5) or built-ins (see 2.3), but only an application that is aware of the command search order described in 3.9.1.1 or of performance characteristics can discern differences between the behavior of such a function or built-in and that of a true executable file.

2.2.2.157 `<vertical-tab>`: The vertical tab character.

2.2.2.158 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>`s (`<space>`s and `<tab>`s), `<newline>`s, `<carriage-return>`s, `<form-feed>`s, and `<vertical-tab>`s.

2.2.2.159 working directory [current working directory]: A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash.

2.2.2.160 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*.

2.2.2.161 General Terms Rationale. *(This subclause is not a part of P1003.2)*

Many of the terms originated in POSIX.1 {8} and are duplicated in this standard to meet editorial requirements. In some cases, there is supplementary text that presents additional information concerning POSIX.2 aspects of the concept.

This standard uses the term *character* to mean a sequence of one or more bytes representing a single graphic symbol, as defined in POSIX.1 {8}. The deviation in the exact text of the C Standard {7} definition for *byte* meets the intent of the C Standard {7} Rationale and the developers of POSIX.1 {8}, but clears up the ambiguity raised by the term *basic execution character set*, which is not defined in POSIX.1 {8}. It is expected that a future version of POSIX.1 {8} will align with the text used here. The octet-minimum requirement is merely a reflection of the {CHAR_BIT} value in POSIX.1 {8} and the C Standard {7}.

The POSIX.1 {8} term *file mode* is a superset of the POSIX.2 *file mode bits*. POSIX.1 {8} defines the file mode as the entire *mode_t* object (which includes the file type in historically the upper four bits, the sticky bit on most implementations, and potentially other nonstandardized attributes), while POSIX.2 file mode bits include only the eleven defined bits.

The terms *command* and *utility* are related but have distinct meanings. Command is defined as ‘‘a directive to a shell to perform a specific task.’’ The directive can be in the form of a single utility name (for example, `ls`), or the directive can take the form of a compound command (for example, `ls | grep name | pr`).

A utility is a program that is callable by name from a shell. Issuing only the utility's name to a shell is the equivalent of a one-word command. A utility may be invoked as a separate program that executes in a different process than the command language interpreter, or may be implemented as a part of the command language interpreter. For example, the `echo` command (the directive to perform a specific task) may be implemented such that the `echo` utility (the logic that performs the task of echoing) is in a separate program; and therefore, is executed in a process that is different than the command language interpreter. Conversely, the logic that performs the `echo` utility could be built into the command language interpreter; and therefore, execute in the same process as the command language interpreter.

The terms *tool* and *application* can be thought of as being synonymous with *utility* from the perspective of the operating system kernel. Tools, applications, and utilities have historically run, typically, in processes above the kernel level. Tools and utilities have been historically a part of the operating system nonkernel code, and performed system related functions such as listing directory contents, checking file systems, repairing file systems, or extracting system status information. Applications have not generally been a part of the operating system, and perform nonsystem related functions such as word processing, architectural design, mechanical design, workstation publishing, or financial analysis. Utilities have most frequently been provided by the operating system vendor, applications by third party software vendors or by the users themselves. Nevertheless, the standard does not differentiate between tools, utilities, and applications when it comes to receiving services from the system, a shell, or the standard utilities. (For example, the `xargs` utility invokes another utility; it would be of fairly limited usefulness if the users couldn't run their own applications in place of the standard utilities.) Utilities are not applications in the sense that they are not themselves subjects to the restrictions of this standard or any other standard— there is no requirement for `grep`, `stty`, or any of the utilities defined here to be any of the classes of Conforming POSIX.2 Applications.

The term *text file* does not prevent the inclusion of control or other nonprintable 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 subclauses. The definition of *text file* has caused a good deal of controversy. The only difference between text and binary here 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>, 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. A related point is that the <newline> character referred to in this standard 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 POSIX systems without some translation process unspecified by this standard.

The term *hard link* is historically-derived. In systems without extensions to `ln`, it is a synonym for *link*. The concept of a *symbolic link* originated with BSD systems and the term *hard* is used to differentiate between the two types of links.

There are some terms used that are undefined in POSIX.2, POSIX.1 {8}, or the C Standard {7}. The working group believes that these terms have a ‘‘common usage,’’ and that a definition in POSIX.2 would not be appropriate. Terms in this category include, but are not limited to, the following: *application, character set, login session, user*. Good sources for general terms of this type are the *ISO / AFNOR Dictionary of Computer Science* {B12} and *IEEE Dictionary* {B18}.

The term *file name* was defined in previous drafts to be a synonym for *pathname*. It was removed in the face of objections that it was too close to *filename*, which means something different (a pathname component). The general solution to this has been to use the term *file* in parameter names, rather than *file_name*, and to make more liberal use of the correct term, *pathname*; an alternate solution has been to replace *file name* with *the name of the file*.

Many character names are included in this subclause. Because of historical usage, some of these names are a bit different than the ones used in international standards for character sets, such as ISO/IEC 646 {1}. It was felt that many more UNIX system people than character set lawyers would be reading and reviewing the standard, so the former group was the one accommodated. On the other hand, the precise definitions of <space>, <blank>, and *white space* have replaced common usage (where they have been used virtually interchangeably), as the standard attempts to balance readability against precision.

In earlier drafts, the names for the character pairs ( ), [ ], and { } were referred to as ‘‘opening’’ and ‘‘closing’’ parentheses, brackets, and braces. These were changed to the current ‘‘left’’ and right.’’ When the characters are used to express natural language, the terms ‘‘open’’ and ‘‘close’’ imply text direction more strongly than ‘‘left’’ and ‘‘right.’’ By POSIX.2 definition, the character <open-parenthesis> will always be mapped to the glyph '(' regardless of the locale. But when reading right-to-left, the opening punctuation of a parenthesized text segment would be ')'. The <left-parenthesis> and <right-parenthesis> forms are the correct ones because the punctuation appears on the left and right, respectively, of the parenthesized text regardless of the direction one might be reading the text.

The <backspace> character and the ERASE special character defined in POSIX.1 {8} should not be confused. The use of the <backspace> character and the ERASE special character defined in the POSIX.1 {8} *termios* clause on special characters (7.1.1.9) are distinct even though the ERASE special character may be set to <backspace>.

In most one-byte character sets, such as ASCII, the concepts of column positions is identical to character positions and to bytes. Therefore, it has been historically acceptable for some implementations to describe line folding or tab stops or table column alignment in terms of bytes or character positions. Other character sets pose complications, as they can have internal representations longer than one octet and they can have displayable characters that have different widths on the terminal screen or printer.

In this standard the term *column positions* has been defined to mean character—not byte—positions in input files (such as ‘‘column position 7 of the FORTRAN input''). Output files describe the column position in terms of the display width of the narrowest printable character in the character set, adjusted to fit the characteristics of the output device. It is very possible that *n* column positions will not be able to hold *n* characters in some character sets, unless all of those characters are of the narrowest width. It is assumed that the implementation is aware of the width of the various characters, deriving this information from the value of **LC_CTYPE**, and thus can determine how many column positions to allot for each character in those utilities where it is important. This information is not available to the portable application writer because POSIX.2 provides no interface specification to retrieve such information.

The term *column position* was used instead of the more natural *column* as the latter is frequently used in the standard in the different contexts of columns of figures, columns of table values, etc. Wherever confusion might result, these latter types of columns are referred to as *text columns*.

The definition of *binary file* was removed, as the term is not used in the standard.

The ISO/IEC 646 {1} character set standard permits substitution of national currency symbols for the character $ in the ''reference character set'' (which is the same as ASCII). This standard permits the substitution only of the actual characters shown in ISO/IEC 646 {1}: currency sign for the dollar sign and pound sign for the number sign. This document uses the latter names and their symbols, but it is valid for an implementation to accept, for instance, the pound sign (£) as a comment character in the shell, if that is what the locale's character set uses instead of the number sign (#). Other variation of national currency symbols are not allowed, per the request of the WG15 POSIX working group.

The term *stream* is not related to System V's STREAMS communications facility; it is derived from historical UNIX system usage and has been made official by the C Standard {7}. The POSIX.2 standard makes no differentiation between C's *text stream* and *binary stream*.

The formula used in the POSIX.1 {8} definition of *seconds since the Epoch* is not 1
perfect in all cases. See the related rationale in POSIX.1 {8}. 1

2.2.3 Abbreviations

For the purposes of this standard, the following abbreviations apply:

2.2.3.1 C Standard: ISO/IEC 9899: . . . , *Information processing systems—Programming languages—C* {7}.

2.2.3.2 ERE : An Extended Regular Expression, as defined in 2.8.4.

2.2.3.3 LC_*: An abbreviation used to represent all of the environment variables named in 2.6 whose names begin with the characters ‘LC_’’.

2.2.3.4 POSIX. 1: ISO/IEC 9945-1: 1990: *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]* {8}.

2.2.3.5 POSIX. 2: This standard.

2.2.3.6 RE [BRE]: A Basic Regular Expression, as defined in 2.8.3.

2.3 Built-in Utilities

Any of the standard utilities may be implemented as *regular built-in* utilities within the command language interpreter. This is usually done to increase the performance of frequently-used utilities or to achieve functionality that would be more difficult in a separate environment. The utilities named in Table 2-2 are frequently provided in built-in form. All of the utilities named in the table have special properties in terms of command search order within the shell, as described in 3.9.1.1.

Table 2-2 – Regular Built-in Utilities

`cd`	`false`	`kill`	`true`	`wait`
`command`	`getopts`	`read`	`umask`	

However, all of the standard utilities, including the regular built-ins in the table, but not the special built-ins described in 3.14, shall be implemented in a manner so that they can be accessed via the POSIX.1 {8} *exec* family of functions (if the underlying operating system provides the services of such a family to application programs) and can be invoked directly by those standard utilities that require it (`env`, `find`, `nohup`, `xargs`).

Since versions shall be provided for all utilities except for those listed previously, an application running on a system that conforms to both POSIX.1 {8} and Section 7 of this standard can use the *exec* family of functions, in addition to the shell command interface in 7.1 [such as the *system* () and *popen* () functions in the C binding] defined by this standard, to execute any of these utilities.

2.3.1 Built-in Utilities Rationale. *(This subclause is not a part of P1003.2)*

In earlier drafts, the table of built-ins implied two things to a conforming application: these may be built-ins and these need not be executable. The second implication has now been removed and all utilities can be *exec*-ed. There is no requirement that these be actually built into the shell itself, but many shells will want to do so because 3.9.1.1 requires that they be found prior to the **PATH** search. The shell could satisfy its requirements by keeping a list of the names and directly accessing the file-system versions regardless of **PATH**. Providing all of the required functionality for those such as `cd` or `read` would be more difficult.

There were originally three justifications for allowing the omission of *exec*-able versions:

(1) This would require wasting space in the file system, at the expense of very small systems. However, it has been pointed out that all nine in the table can be provided with nine links to a single-line shell script:

```
$0 "$@"
```

(2) There is no sense in requiring invocation of utilities like `cd` because they have no value outside the shell environment or cannot be useful in a child process. However, counter-examples always seemed to be available for even the strangest cases:

```
find . -type d -exec cd {} ; -exec foo {} ;
```
(which invokes `foo` on accessible directories)

```
ps . . . | sed . . . | xargs kill
```

```
find . -exec true ; -a . . .
```
(where `true` is used for temporary debugging)

(3) It is confusing to have something such as `kill` that can easily be in the file system in the base standard, but requires built-in status for the UPE (for the `%` job control job ID notation). It was decided that it was more appropriate to describe the required functionality (rather than the implementation) to the system implementors and let them decide how to satisfy it.

On the other hand, there were objections raised during balloting that any distinction like this between utilities was not useful to applications and that the cost to correct it was small. These arguments were ultimately the most effective.

There were varying reasons for including utilities in the table of built-ins:

`cd`, `getopts`, `read`, `umask`, `wait`
: The functionality of these utilities is performed more simply within the context of the current process. An example can be taken from the usage of the `cd` utility. The purpose of the utility is to change the working directory for subsequent operations. The actions of `cd` affect the process in which `cd` is executed and all subsequent child processes of that process. Based on the POSIX.1 {8} process model, changes in the process environment of a child process have no effect on the parent process. If the `cd` utility were executed from a child process, the working directory change would be effective only in the child process. Child processes initiated subsequent to the child process that executed the `cd` utility would not have a changed working directory relative to the parent process.

`command`
: This utility was placed in the table primarily to protect scripts that are concerned about their **PATH** being manipulated. The ‘‘secure’’ shell script example in 4.12.10 would not be possible if a **PATH** change retrieved an alien version of `command`. (An alternative would have been to implement `getconf` as a built-in, but it was felt that it carried too many changing configuration strings to require in the shell.)

`kill`
: Since common extensions to `kill` (including the planned User Portability Extension) provide optional job control functionality using shell notation (`%1`, `%2`, etc.), some implementations would

find it extremely difficult to provide this outside the shell.

`true`, `false`

These are in the table as a courtesy to programmers who wish to use the '`while true`'' shell construct without protecting `true` from **PATH** searches. (It is acknowledged that '`while :`'' also works, but the idiom with `true` is historically pervasive.)

All utilities, including those in the table, are accessible via the functions in 7.1.1 or 7.1.2 [such as *system* () or *popen* ()]. There are situations where the return functionality of *system* () and *popen* () is not desirable. Applications that require the exit status of the invoked utility will not be able to use *system* () or *popen* (), since the exit status returned is that of the command language interpreter rather than that of the invoked utility. The alternative for such applications is the use of the *exec* family. (The text concerning conformance to POSIX.1 {8} was included because where *exec* is not provided in the underlying system, there is no way to require that utilities be *exec*-able).

2.4 Character Set

Table 2-3 – Character Set and Symbolic Names

Symbolic Name	Glyph	Symbolic Name	Glyph	Symbolic Name	Glyph
<NUL>		<colon>	:	<circumflex>	ˆ
<alert>		<semicolon>	;	<circumflex-accent>	ˆ
<backspace>		<less-than-sign>	<	<underscore>	_
<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>	~

Conforming implementations shall support one or more coded character sets. Each supported coded character set shall include the *portable character set* specified in Table 2-3. The table 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 names are chosen to correspond closely with character names defined in other international standards. The table contains more than one symbolic character name for characters whose traditional name differs from the chosen name.

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

(1) 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.

(2) The encoded values associated with the digits `'0'` to `'9'` shall be such that the value of each character after `'0'` shall be one greater than the value of the previous character.

(3) A null character, NUL, which has all bits set to zero, shall be in the set of characters.

Conforming implementations shall support certain character and character set attributes, as defined in 2.5.1.

2.4.1 Character Set Description File

Implementations shall 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 standard as *charmap* files. It is implementation defined whether or not users or applications can provide additional character set description files. If such a capability is supported, the system documentation shall describe the rules for the creation of such files.

Each character set description file shall define characteristics for the coded character set and the encoding for the characters specified in Table 2-3, 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 shall be defined using symbolic character names followed by character encoding values.

Each symbolic name specified in Table 2-3 shall be included in the file and shall be 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 Table 2-4 are supported by the implementation, the symbolic names and their corresponding encoding values shall be included in the file. Some of the values associated with the symbolic names in this table also may be contained in Table 2-3.

The following declarations can precede the character definitions. Each shall consist of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one of more `<blank>`s, 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 shall be taken from the set of characters with

Table 2-4 – Control Character Set

<ACK>	<DC2>	<ENQ>	<FS>	<IS4>	<SOH>	1
<BEL>	<DC3>	<EOT>	<GS>	<LF>	<STX>	1
<BS>	<DC4>	<ESC>	<HT>	<NAK>	<SUB>	1
<CAN>	<DEL>	<ETB>	<IS1>	<RS>	<SYN>	1
<CR>	<DLE>	<ETX>	<IS2>	<SI>	<US>	1
<DC1>	<EM>	<FF>	<IS3>	<SO>	<VT>	1

visible glyphs defined in Table 2-3. 1

`<mb_cur_max>` The maximum number of bytes in a multibyte character. This shall default to 1.

`<mb_cur_min>` An unsigned positive integer value that shall define the minimum number of bytes in a character for the encoded character set. The value shall be less than or equal to `mb_cur_max`. If not specified, the minimum number shall be equal to `mb_cur_max`.

`<escape_char>` The escape character used to indicate that the characters following shall be interpreted in a special way, as defined later in this subclause. This shall default 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 shall be ignored. The default character shall be the number-sign (#).

The character set mapping definitions shall 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 shall be ignored. Each noncomment line of the character set mapping definition (i.e., between the `CHARMAP` and `END CHARMAP` lines of the file) shall 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 2-3, enclosed between angle brackets. A character following an escape character shall be 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 shall consist of zero or more nonnumeric characters from the set shown with visible glyphs in Table 2-3, followed by an integer formed by one or more decimal digits. The characters preceding the integer shall be identical in the two symbolic names, and the integer formed by the digits in the second symbolic name shall be equal to or greater than the integer formed by the digits in the first name. This shall be 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 shall exist for all symbolic names specified in Table 2-3, and shall define the coded character value that corresponds with 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 2-4 are supported by the implementation, the symbolic name and the corresponding encoding value shall 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 shall be expressed as one (for single-byte character values) or 1
more concatenated decimal, octal, or hexadecimal constants in the following for- 1
mats:

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

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

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

Decimal constants shall be represented by two or three decimal digits, preceded 2
by the escape character and the lowercase letter `d`; for example, `\d05`, `\d97`, or 2
`\d143`. Hexadecimal constants shall be represented by two hexadecimal digits, 2
preceded by the escape character and the lowercase letter `x`; for example, `\x05`, 2
`\x61`, or `\x8f`. Octal constants shall be represented by two or three octal digits, 2
preceded by the escape character; for example, `\05`, `\141`, or `\217`. In a portable 2
charmap file, each constant shall represent an 8-bit byte. Implementations sup- 2
porting other byte sizes may allow constants to represent values larger than those 2
that can be represented in 8-bit bytes, and to allow additional digits in constants. 2
When constants are concatenated for multibyte character values, they shall be of 2
the same type, and interpreted in byte order from left to right. The manner in 2
which constants are represented in the character is implementation defined. 2
Omitting bytes from a multibyte character definition produces undefined results. 2

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 shall have encoding values in increasing order. For example, the line

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

shall be interpreted as

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

The comment is optional.

For the interpretation of the dollar-sign and the number-sign, see 2.2.2.37 and 2.2.2.93.

2.4.2 Character Set Rationale. *(This subclause is not a part of P1003.2)*

The portable character set is listed in full so there is no dependency on the ISO/IEC 646 {1} (or historically ASCII) encoded character set, although the set is identical to the characters defined in the International Reference Version of ISO/IEC 646 {1}.

This standard poses no requirement that multiple character sets or code sets be supported, leaving this as a marketing differentiation for implementors. Although multiple *charmap* files are supported, it is the responsibility of the implementation to provide the file(s); 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).

The statement about invariance in code sets for the portable character set is worded as it is to avoid precluding implementations where multiple incompatible code sets are available (say, ASCII and EBCDIC). The standard utilities cannot be expected to produce predictable results if they access portable characters that vary on the same implementation.

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 code sets that share the portable character set (such as the ISO 8859 family or IBM Extended ASCII).

— standardized symbolic names for all characters in the portable character set, making it possible to refer to any such character regardless of encoding.

Implementations are free to describe more than one code set in a character set description file, as long as only one encoding exists for the characters in Table 2-3. For example, if an implementation defines ISO 8859-1 {5} as the primary code set, and ISO 8859-2 {6} as an alternate set, with each character from the alternate code set preceded in data by a shift code, a character set description file could contain a complete description of the primary set and those characters from the secondary that are not identical, the encoding of the latter including the shift code.

Implementations are free to choose their own symbolic names, as long as the names identified by this standard are also defined; this provides support for already existing ‘‘character names.’’

The names selected for the members of the portable character set follow the ISO 8859 {5} and the ISO/IEC 10646 {B11} standards. However, several commonly used UNIX system names occur as synonyms in the list:

— The traditional UNIX system names are used for control characters.

— The word ‘‘slash’’ is in addition to ‘‘solidus.’’ 1

— The word ‘‘backslash’’ is in addition to ‘‘reverse-solidus.’’ 1

— The word ‘‘hyphen’’ in addition to ‘‘hyphen-minus.’’

— The word ‘‘period’’ in addition to ‘‘full-stop.’’

— For the digits, the word ‘‘digit’’ is eliminated.

— For letters, the words ‘‘Latin Capital Letter’’ and ‘‘Latin Small Letter’’ are eliminated.

— The words ‘‘left-brace’’ and ‘‘right-brace’’ in addition to ‘‘left-curly-bracket’’ and ‘‘right-curly-bracket.’’

— The names of the digits are preferred over the numbers, to avoid possible confusion between ‘‘0’’ and ‘‘O’’, and between ‘‘1’’ and ‘‘l’’ (one and the letter ell).

The names for the control characters in Table 2-4 were taken from ISO 4873 {4}.

The charmap file was introduced to resolve problems with the portability of, espe-
cially, `localedef` sources. This standard assumes that the portable character 1
set is constant across all locales, but does not prohibit implementations from sup- 1
porting two incompatible codings, such as both ASCII and EBCDIC. Such ‘‘dual- 1
support’’ implementations should have all charmaps and `localedef` sources 1
encoded using one portable character set, in effect ‘‘cross-compiling’’ for the other 1
environment. Naturally, charmaps (and `localedef` sources) are only portable 1
without transformation between systems using the same encodings for the port- 1
able character set. They can, however, be transformed between two sets using 1
only a subset of the actual characters (the portable set). However, the particular 1
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 code set. The charmap provides the capability to define a common locale definition for multiple code sets (the same `localedef` source can be used for code sets with different extended characters; the ability in the charmap to define ‘‘empty’’ names allows for characters missing in certain code sets).

In addition, several implementors have expressed an interest in using the charmap concept to provide the information required for support of multiple character sets. Examples of such information is encoding mechanism, string parsing rules, default font information, etc. Such extensions are not described here.

The <escape_char> declaration was added at the request of the international community to ease the creation of portable *charmap* files on terminals not implementing the default backslash escape. (This approach was adopted because this is a new interface invented by POSIX.2. Historical interfaces, such as the shell command language and awk, have not been modified to accommodate this type of terminal.) The <comment_char> declaration was added at the request of the international community to eliminate the potential confusion between the number sign and the pound sign.

The octal number notation with no leading zero required was selected to match 1
those of awk and tr and is consistent with that used by localedef. To avoid 1
confusion between an octal constant and the backreferences used in localedef 1
source, the octal, hexadecimal, and decimal constants must contain at least two 1
digits. As single-digit constants are relatively rare, this should not impose any 1
significant hardship. Each of the constants includes ‘‘two or more’’ digits to 1
account for systems in which the byte size is larger than eight bits. For example, 1
a Unicode system that has defined 16-bit bytes may require six octal, four hexade- 1
cimal, and five decimal digits. 1

The decimal notation is supported because some newer international standards define character values in decimal, rather than in the old column/row notation.

The charmap identifies the coded character sets supported by an implementation. At least one charmap must be provided, but no implementation is required to provide more than one. Likewise, implementations can allow users to generate new charmaps (for instance for a new version of the 8859 family of coded character sets), but does not have to do so. If users are allowed to create new charmaps, the system documentation must describe the rules that apply (for instance: ‘‘only coded character sets that are supersets of ISO/IEC 646 {1} IRV, no multibyte characters, etc.’’)