--   --

IEEE P1003.2 Draft 11.2 − September 1991

Section 2: Terminology and General Requirements

2.1 Conventions

2.1.1 Editorial Conventions

This standard uses the following editorial and typographical conventions. A summary of typographical conventions is shown in Table 2-1.

Table 2-1 – Typographical Conventions

Reference	Example
C-Language Data Type	*long*
C-Language Function	*system* ()
C-Language Function Argument	*arg1*
C-Language Global External	*errno*
C-Language Header	<sys/stat.h>
C-Language Keyword	#define
Cross Reference: Annex	Annex A
Cross Reference: Clause	2.3
Cross Reference: Other Standard	ISO 9999-1 {*n*}
Cross Reference: Section	Section 2
Cross Reference: Subclause	2.3.4, 2.3.4.5, 2.3.4.5.6
Defined Term	(see text)
Environment Variable	**PATH**
Error Number	[EINTR]
Example Input	echo foo
Example Output	**foo**
Figure Reference	Figure 7
File Name	/tmp
Parameter	<*directory pathname* >
Special Character	<newline>
Symbolic Constant, Limit	{_POSIX_VDISABLE }, {LINE_MAX}
Table Reference	Table 6
Utility Name	awk
Utility Operand	*file_name*
Utility Option	−c
Utility Option with Option-Argument	−w *width*

The **`Bold Courier`** font is used to show brackets that denote optional arguments in a utility synopsis, as in

`cut` **`[`**`−c` *list* **`]`** **`[`***file_name* **`]`**

These brackets shall not be used by the application unless they are specifically mentioned as literal input characters by the utility description.

There are two types of symbols enclosed in angle brackets (< >):

C-Language Headers The header name is in the Courier font, such as `<sys/stat.h>`. When coding C programs, the brackets are used as required by the language.

Parameters Parameters, also called *metavariables* , are in italics, such as *<directory pathname>* . The entire symbol, including the brackets, is meant to be replaced by the value of the symbol described within the brackets.

Numbers within braces, such as ‘‘POSIX.1 {8},’’ represent cross references to the Normative References clause (see 1.2). If the number is preceded by a B, it represents a Bibliographic entry (see Annex D). Bibliographic entries are for information only.

In some examples, the **`Bold Courier`** font is used to indicate the system’s output that resulted from some user input, shown in `Courier`.

Defined terms are shown in three styles, depending on context:

(1) Terms defined in 2.2.1, 2.2.2, and 3.1 are expressed as subclause titles. Alternative forms of the terms appear in [brackets].

(2) The initial appearances of other terms, applying to a limited portion of the text, are in *italics* .

(3) Subsequent appearances of the term are in the Roman font.

Symbolic constants are shown in two styles: those within curly braces are intended to call the reader’s attention to values in `<limits.h>` and `<unistd.h>`; those without braces are usually defined by one or a few related functions. There is no semantic difference between these two forms of presentation.

Filenames and pathnames are shown in `Courier`. When a pathname is shown starting with ‘‘`$HOME/`’’, this indicates the remaining components of the pathname are to be related to the directory named by the user’s **HOME** environment variable.

The style selected for some of the special characters, such as `<newline>`, matches the form of the input given to the `localedef` utility (see 2.5.2). Generally, the characters selected for this special treatment are those that are not visually distinct, such as the control characters `<tab>` or `<newline>`.

Literal characters and strings used as input or output are shown in various ways, depending on context:

`%`, `begin` When no confusion would result, the character or string is rendered in the Courier font and used directly in the text.

`'c'` In some cases a character is enclosed in single-quote characters, similar to a C-language character constant. Unless otherwise noted, the quotes shall not be used as input or output.

`"string"` In some cases, a string is enclosed in double-quote characters, similar to a C-language string constant. Unless otherwise noted, the quotes shall not be used as input or output.

Defined names that are usually in lowercase, particularly function names, are never used at the beginning of a sentence or anywhere else that regular English usage would require them to be capitalized.

Parenthetical expressions within normative text also contain normative information. The general typographic hierarchy of parenthetical expressions is:

{ [( )] }

The square brackets are most frequently used to enclose a parenthetical expression that contains a function name [such as *waitpid* ()], with its built-in parentheses.

In some cases, tabular information is presented inline; in others it is presented in a separately-labeled Table. This arrangement was employed purely for ease of reference and there is no normative difference between these two cases.

Annexes marked as *normative* are parts of the standard that pose requirements, exactly the same as the numbered Sections, but have been moved to near the end of the document for clarity of exposition. *Informative* Annexes are for information only and pose no requirements. All material preceding page 1 of the document (the ‘‘front matter’’) and the two indexes at the end are also only informative.

NOTES that appear in a smaller point size and are indented have one of two different meanings, depending on their location:

— When they are within the normal text of the document, they are the same as footnotes—informative, posing no requirements on implementations or applications.

— When they are attached to Tables or Figures, they are normative, posing requirements.

Text marked as examples (including the use of ‘‘e.g.’’) is for information only. The exception to this comes in the C-language programs and program fragments used to represent algorithms, as described in 2.1.3.

The typographical conventions listed here are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

2.1.2 Grammar Conventions

Portions of this standard are expressed in terms of a special grammar notation. It is used to portray the complex syntax of certain program input. The grammar is based on the syntax used by the `yacc` utility (see A.3). However, it does not represent fully functional `yacc` input, suitable for program use: the lexical processing and all semantic requirements are described only in textual form. The grammar is not based on source used in any traditional implementation and has not been tested with the semantic code that would normally be required to accompany it. Furthermore, there is no implication that the partial `yacc` code presented represents the most efficient, or only, means of supporting the complex syntax within the utility. Implementations may use other programming languages or algorithms, as long as the syntax supported is the same as that represented by the grammar.

The following typographical conventions are used in the grammar; they have no significance except to aid in reading.

- The identifiers for the reserved words of the language are shown with a leading capital letter. (These are terminals in the grammar. Examples: `While`, `Case`.)
- The identifiers for terminals in the grammar are all named with uppercase 1
  letters and underscores. Examples: `NEWLINE`, `ASSIGN_OP`, `NAME`. 1
- The identifiers for nonterminals are all lowercase.

2.1.3 Miscellaneous Conventions

This standard frequently uses the C language to express algorithms in terms of programs or program fragments. The following shall be considered in reading this code:

- The programs use the syntax and semantics described by the C Standard {7}.
- The programs are merely examples and do not represent the most efficient, or only, means of coding the interface. Implementations may use other programming languages or algorithms, as long as the results are the same as those achieved by the programs in this standard.
- C-language comments are informative and pose no requirements.

Further conventions are presented in:

- Utility Conventions, 2.10, describing utility and application command-line syntax
- File Format Notation, 2.12, describing the notation used to represent utility input and output

2.1.4 Conventions Rationale. *(This subclause is not a part of P1003.2)*

The C language was chosen for many examples because:

— It eliminates any requirement to document a different pseudocode.

— It is a familiar language to many of the potential readers of POSIX.2.

— It is the language most widely used for historical implementations of the utilities.

2.2 Definitions

2.2.1 Terminology

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

2.2.1.1 can: The word *can* is to be interpreted as describing a permissible optional feature or behavior available to the application; the implementation shall support such features or behaviors as mandatory requirements.

2.2.1.2 conformance document: A document provided by an implementor that contains implementation details as described in 1.3.1.2.

2.2.1.3 implementation: An object providing to applications and users the services defined by this standard. The word *implementation* is to be interpreted to mean that object, after it has been modified in accordance with the manufacturer's instructions to:

— configure it for conformance with this standard;

— select some of the various optional facilities described by this standard, through customization by local system administrators or operators.

An exception to this meaning occurs when discussing conformance documentation or using the term *implementation defined*. See 2.2.1.4 and 1.3.1.2.

2.2.1.4 implementation defined: When a value or behavior is described by this standard as *implementation defined*, the implementation provider shall document the requirements for correct program construction and correct data in the use of that value or behavior. When the value or behavior in the implementation is designed to be variable or customizable on each instantiation of the system, the implementation provider shall document the nature and permissible ranges of this variation. (See 1.3.1.2.)

2.2.1.5 may: The word *may* is to be interpreted as describing an optional feature or behavior of the implementation that is not required by this standard, but there is no prohibition against providing it. A Strictly Conforming POSIX.2 Application 1