-- --

IEEE P1003.2 Draft 11.2 − September 1991

(2) For regular files:

(a) The file's user ID, group ID, and permission bits shall not be changed.

(b) The file shall be truncated to zero length.

(c) The *st_ctime* and *st_mtime* fields shall be marked for update.

(3) For other file types, the effect is implementation defined.

When a file is to be appended, the file shall be opened in a manner equivalent to using the O_APPEND flag, without the O_TRUNC flag, in the POSIX.1 {8} *open*() call.

2.9.1.4.1 File Read, Write, and Creation Rationale. *(This subclause is not a part of P1003.2)*

Even though it might be possible for a process to change the mode of a file to match a requested operation and change the mode back to its original state after the operation is completed, utilities are not allowed to do this unless the utility description states otherwise. As an example, the `ed` utility `r` command fails if the file to be read does not exist (even though it could create the file and then read it) or the file permissions do not allow read access [even though it could use the POSIX.1 {8} *chmod*() function to make the file readable before attempting to open the file].

2.9.1.5 File Removal

When a directory that is the root directory or current working directory of any process is removed, the effect is implementation defined. If file access permissions deny access, the requested operation shall fail. Otherwise, when a file is removed:

(1) Its directory entry shall be removed from the file system.

(2) The link count of the file shall be decremented.

(3) If the file is an empty directory (see 2.2.2.43):

(a) If no process has the directory open, the space occupied by the directory shall be freed and the directory shall no longer be accessible.

(b) If one or more processes have the directory open, the directory contents shall be preserved until all references to the file have been closed.

(4) If the file is a directory that is not empty, the *st_ctime* field shall be marked for update.

(5) If the file is not a directory:

(a) If the link count becomes zero:

[1] If no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible.

[2] If one or more processes have the file open, the file contents shall be preserved until all references to the file have been closed.

(b) If the link count is not reduced to zero, the *st_ctime* field shall be marked for update.

(6) The *st_ctime* and *st_mtime* fields of the containing directory shall be marked for update.

2.9.1.5.1 File Removal Rationale. *(This subclause is not a part of P1003.2)*

This is intended to be a summary of the POSIX.1 {8} *unlink*() and *rmdir*() requirements needed by POSIX.2.

2.9.1.6 File Time Values

All files have the three time values described by *file times update* in 2.2.2.69.

2.9.1.6.1 File Time Values Rationale. *(This subclause is not a part of P1003.2)*

All three time stamps specified by POSIX.1 {8} are needed for utilities like `find`, `ls`, `make`, `test`, and `touch` to work as expected.

2.9.1.7 File Contents

When a reference is made to the contents of a file, *pathname*, this means the equivalent of all of the data placed in the space pointed to by *buf* when performing the *read*() function calls in the following POSIX.1 {8} operations:

```
while (read (fildes, buf, nbytes) > 0)
        ;
```

If the file is indicated by a pathname *pathname*, the file descriptor shall be determined by the equivalent of the following POSIX.1 operation:

```
fildes = open (pathname, O_RDONLY);
```

The value of *nbytes* in the above sequence is unspecified; if the file is of a type where the data returned by *read*() would vary with different values, the value shall be one that results in the most data being returned.

If the *read*() function calls would return an error, it is unspecified whether the contents of the file are considered to include any data from offsets in the file beyond where the error would be returned.

2.9.1.7.1 File Contents Rationale. *(This subclause is not a part of P1003.2)*

This description is intended to convey the traditional behavior for all types of files. This matches the intuitive meaning for regular files, but the meaning is not always intuitive for other types of files. In particular, for FIFO s, pipes, and terminals it must be clear that the contents are not necessarily static at the time a file is opened, but they include the data returned by a sequence of reads until end-of-file is indicated. This is why the *open* () call is specified, with the O_NONBLOCK flag not set.

Some files, especially character special files, are sensitive to the size of a *read* () request. The contents of the file are those resulting from proper choice of this size.

2.9.1.8 Pathname Resolution

The pathname resolution algorithm described by *pathname resolution* in 2.2.2.104 shall be used by conforming POSIX. 2 implementations. See also *file hierarchy* in 2.2.2.58.

2.9.1.8.1 Pathname Resolution Rationale. *(This subclause is not a part of P1003.2)*

The whole concept of hierarchical file systems and pathname resolution is assumed to be handled as in POSIX .1 {8}.

2.9.1.9 Changing the Current Working Directory

When the current working directory (see 2.2.2.159) is to be changed, unless the utility or function description states otherwise, the operation shall succeed unless a call to the POSIX .1 {8} *chdir* () function would fail when invoked with the new working directory pathname as its argument.

2.9.1.9.1 Changing the Current Working Directory Rationale. *(This subclause is not a part of P1003.2)*

This subclause covers the access permissions and pathname structures involved with changing directories, such as with `cd` or (the UPE -extended) `mailx` utilities.

2.9.1.10 Establish the Locale

The functionality of the POSIX .1 {8} *setlocale* () function is assumed to be available on all POSIX. 2 conformant systems; i.e., utilities that require the capability of establishing an international operating environment shall be permitted to set the specified category of the international environment.

2.9.1.10.1 Establish the Locale Rationale. *(This subclause is not a part of P1003.2)*

The entire concept of locale categories such as the **LC_*** variables along with any implementation-defined categories is assumed to be handled as in POSIX.1 {8}.

2.9.1.11 Actions Equivalent to POSIX.1 Functions

Some utility descriptions specify that a utility performs actions equivalent to a POSIX.1 {8} function. Such specifications require only that the external effects be equivalent, not that any effect within the utility and visible only to the utility be equivalent.

2.9.1.11.1 Actions Equivalent to POSIX.1 Functions Rationale. *(This subclause is not a part of P1003.2)*

An objection was received to an earlier draft that said this approach of equivalent functions was unreasonable, as the reader (and the person writing a test suite) would be responsible for interpreting which portions of POSIX.1 {8} were included and which were not. For example, would such intermediate effects as the setting of *errno* be required if the related POSIX.1 {8} function called for that? The answer is no: this standard is only concerned with the end results of functions against the file system and the environment, and not any intermediate values or results visible only to the programmer using the POSIX.1 {8} function in a C (or other high-level language) program.

2.9.2 Concepts Derived from the C Standard

Some of the standard utilities perform complex data manipulation using their own procedure and arithmetic languages, as defined in their Extended Description or Operands subclauses. Unless otherwise noted, the arithmetic and semantic concepts (precision, type conversion, control flow, etc.) are equivalent to those defined in the C Standard {7}, as described in the following subclauses. Note that there is no requirement that the standard utilities be implemented in any particular programming language.

2.9.2.0.1 Concepts Derived from the C Standard Rationale. *(This subclause is not a part of P1003.2)*

This subclause was introduced to answer complaints that there was insufficient detail presented by such utilities as `awk` or `sh` about their procedural control statements and their methods of performing arithmetic functions. Earlier drafts, derived heavily from the original manual pages, contained statements such as ‘`for` loops similar to the C Standard {7},’’which was good enough for a general understanding, but insufficient for a real implementation.

The C Standard {7} was selected as a model because most historical implementations of the standard utilities were written in C. Thus, it is more likely that they will act in a manner desired by POSIX.2 without modification.

Using the C Standard {7} is primarily a notational convenience, so the many ''little languages'' in POSIX.2 would not have to be rigorously described in every aspect. Its selection does not require that the standard utilities be written in Standard C; they could be written in common-usage C, Ada, Pascal, assembler language, or anything else.

The sizes of the various numeric values refer to C-language datatypes that are allowed to be different sizes by the C Standard {7}. Thus, like a C-language application, a shell application cannot rely on their exact size. However, it can rely on their minimum sizes expressed in the C Standard {7}, such as {LONG_MAX} for a *long* type.

2.9.2.1 Arithmetic Precision and Operations

Integer variables and constants, including the values of operands and option-arguments, used by the standard utilities shall be implemented as equivalent to the C Standard {7} *signed long* data type; floating point shall be implemented as equivalent to the C Standard {7} *double* type. Conversions between types shall be as described in the C Standard {7}. All variables shall be initialized to zero if they are not otherwise assigned by the application's input.

Arithmetic operators and functions shall be implemented as equivalent to those in the cited C Standard {7} section, as listed in Table 2-14.

The evaluation of arithmetic expressions shall be equivalent to that described in the C Standard {7} section 3.3 Expressions.

2.9.2.2 Mathematic Functions

Any mathematic functions with the same names as those in the C Standard {7}'s sections:

4.5 *Mathematics* `<math.h>`

4.10.2 *Pseudo-random sequence generation functions*

shall be implemented to return the results equivalent to those returned from a call to the corresponding C function described in the C Standard {7}.

Table 2-14 – C Standard Operators and Functions

<table>
<tr><th>Operation</th><th>C Standard {7} Equivalent Reference</th></tr>
<tr><td>( )</td><td>3.3.1 Primary Expressions</td></tr>
<tr><td>postfix ++
postfix −−</td><td>3.3.2 Postfix Operators</td></tr>
<tr><td>unary +
unary −
prefix ++
prefix −−
~
!
<code>sizeof()</code></td><td>3.3.3 Unary Operators</td></tr>
<tr><td>*
/
%</td><td>3.3.5 Multiplicative Operators</td></tr>
<tr><td>+
−</td><td>3.3.6 Additive Operators</td></tr>
<tr><td><<
>></td><td>3.3.7 Bitwise Shift Operators</td></tr>
<tr><td><, <=
>, >=</td><td>3.3.8 Relational Operators</td></tr>
<tr><td>==
!=</td><td>3.3.9 Equality Operators</td></tr>
<tr><td>&</td><td>3.3.10 Bitwise AND Operator</td></tr>
<tr><td>ˆ</td><td>3.3.11 Bitwise Exclusive OR Operator</td></tr>
<tr><td>|</td><td>3.3.12 Bitwise Inclusive OR Operator</td></tr>
<tr><td>&&</td><td>3.3.13 Logical AND Operator</td></tr>
<tr><td>||</td><td>3.3.14 Logical OR Operator</td></tr>
<tr><td>expr ?expr :expr</td><td>3.3.15 Conditional Operator</td></tr>
<tr><td>=, *=, /=, %=, +=, −=
<<=, >>=, &=, ˆ=, |=</td><td>3.3.16 Assignment Operators</td></tr>
<tr><td><code>if ( )</code>
<code>if ( ) ... else</code>
<code>switch ( )</code></td><td>3.6.4 Selection Statements</td></tr>
<tr><td><code>while ( )</code>
<code>do ... while ( )</code>
<code>for ( )</code></td><td>3.6.5 Iteration Statements</td></tr>
<tr><td><code>goto</code>
<code>continue</code>
<code>break</code>
<code>return</code></td><td>3.6.6 Jump Statements</td></tr>
</table>

2.10 Utility Conventions

2.10.1 Utility Argument Syntax

This subclause describes the argument syntax of the standard utilities and introduces terminology used throughout the standard for describing the arguments processed by the utilities.

Within the standard, 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 3.9.1):

utility_name [−a] [−b] [−c *option_argument*] [−d | −e]
    [−f*option_argument*] [*operand* ...]

The notation used for the Synopsis subclauses imposes requirements on the implementors of the standard utilities and provides a simple reference for the reader of the standard.

(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 <blanks>, 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 2.10.2 require that the option be a separate argument from its option-argument, but there are some exceptions in this standard to ensure continued operation of historical applications:

   (a) If the Synopsis of a standard utility shows a <space> between an option and option-argument (as with [−c *option_argument*] in the example), a conforming application shall use separate arguments for that option and its option-argument.

   (b) If a <space> is not shown (as with [−f*option_argument*] in the example), a conforming application shall place an option and its option-argument directly adjacent in the same argument string, without intervening <blank>s.

   (c) Notwithstanding the requirements on conforming applications, a conforming implementation shall permit, but shall not require, an application to specify options and option-arguments as separate arguments whether or not a <space> is shown on the synopsis line.

   (d) A standard utility may also be implemented to operate correctly when the required separation into multiple arguments is violated by a nonconforming application.

(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 subclause, or unless the exception in 2.10.2 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 shall 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 shall be interpreted as a decimal integer.

— numerals in the range 0 to 2 147 483 647 shall be syntactically recognized 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 shall be syntactically recognized as numeric values.

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 this standard, describes that smaller range along with the description of the option-argument or operand. If an error is generated, the utility's diagnostic message shall indicate that the value is out of the supported range, not that it is syntactically incorrect.

(7) Arguments or option-arguments enclosed in the **[** and **]** notation are optional and can be omitted. The **[** and **]** symbols shall never be included in data submitted to the utility.

(8) Arguments separated by the **|** vertical bar notation are mutually exclusive. The **|** symbols shall 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 -e[-b][`*operand* ...`]`

When multiple synopsis lines are given for a utility, that 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 subclause, the relationships between arguments depicted in the Synopsis subclauses are mandatory requirements placed on conforming 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.

(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* ...`]` 1

`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 subclause of the utility. (See also Guideline 11 in 2.10.2.) In the 1
first example, each option-argument requires a preceding `-f` and at least 1
one `-f` *option_argument* must be given. 1

(10) When the synopsis line is too long to be printed on a single line in this document, the indented lines following the initial line are continuation lines. An actual use of the command would appear on a single logical line.

2.10.1.1 Utility Argument Syntax Rationale. *(This subclause is not a part of P1003.2)*

This is the subclause where the definitions of *option*, *option-argument*, and *operand* come together.

The working group felt that recent trends toward diluting the Synopsis subclauses of historical manual pages to something like:

`command` [*options*] [*operands*]

were a disservice to the reader. Therefore, considerable effort was placed into rigorous definitions of all the command line arguments and their interrelationships. The relationships depicted in the Synopses are normative parts of this standard; this information is sometimes repeated in textual form, but that is only for clarity within context.

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 can 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 POSIX.2 to have a sensible meaning and result.

POSIX.2 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 nonstandard) behavior for the utility when more than one such option or operand are specified.

Allowing `<blank>`s after an option (i.e., placing an option and its option-argument into separate argument strings) when the standard does not require it encourages portability of users, while still preserving backward compatibility of scripts. Inserting `<blank>`s between the option and the option-argument is preferred; however, historical usage has not been consistent in this area; therefore, `<blank>`s are required to be handled by all implementations, but implementations are also allowed to handle the historical syntax. Another justification for selecting the multiple-argument method was that the single-argument case is inherently ambiguous when the option-argument can legitimately be a null string.

Wording was also added to explicitly state that digits are permitted as operands and option-arguments. The lower and upper bounds for the values of the numbers used for operands and option-arguments were derived from the C Standard {7} values for {LONG_MIN} and {LONG_MAX}. The requirement on the standard utilities is that numbers in the specified range do not cause a syntax error although the specification of a number need not be semantically correct for a particular operand or option-argument of a utility. For example, the specification of `dd obs=3000000000` would yield undefined behavior for the application and would 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.

2.10.2 Utility Syntax Guidelines

The following guidelines are established for the naming of utilities and for the specification of options, option-arguments, and operands. Clause 7.5 describes a function that assists utilities in handling options and operands that conform to these guidelines.

Operands and option-arguments can contain characters not specified in 2.4.

The guidelines are intended to provide guidance to the authors of future utilities. Some of the standard utilities do not conform to all of these guidelines; in those cases, the Options subclauses describe the deviations.

Guideline 1: Utility names should be between two and nine characters, inclusive.

Guideline 2: Utility names should include lowercase letters (the `lower` character classification) from the set described in 2.4 and digits only.

Guideline 3: Each option name should be a single alphanumeric character (the `alnum` character classification) from the set described in 2.4. The `−W` (capital-W) option shall be reserved for vendor extensions.

NOTE: The other alphanumeric characters are subject to standardization in the future, based on historical usage. Implementors should be aware that future POSIX working groups may offer little sympathy to vendors with isolated extensions in conflict with future drafts.

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 2.10.1, 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>s within that argument to separate them. 2

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.

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.

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).

Any utility claiming conformance to these guidelines shall conform completely to these guidelines, as if these guidelines contained the term ‘‘shall’’ instead of ‘‘should,’’ except that the utility is permitted to accept usage in violation of these guidelines for backward compatibility as long as the required form is also accepted.

Guidelines 1 and 2 are offered as guidance for locales using Latin alphabets. No recommendations are made by this standard concerning utility naming in other locales.

2.10.2.1 Utility Syntax Guidelines Rationale. *(This subclause is not a part of P1003.2)*

This subclause is based on the rules listed in the *SVID*. It was included for two reasons:

(1) The individual utility descriptions in Sections 4, 5, and 6, and Annexes A and C needed a set of common (although not universal) actions on which they could anchor their descriptions of option and operand syntax. Most of the standard utilities actually do use these guidelines, and many of their historical implementations use the *getopt* () function for their parsing. Therefore, it was simpler to cite the rules and merely identify exceptions.

(2) Writers of portable applications need suggested guidelines if the POSIX community is to avoid the chaos of historical UNIX system command syntax.

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.

The voluntary nature of the guidelines is highlighted by repeated uses of the word *should* throughout. This usage should not be misinterpreted to imply that utilities that claim conformance in their Options subclauses do not always conform.

Guideline 2 recommends the naming of utilities. In 3.9.1, it is further stated that a command used in the shell command language cannot be named with a trailing colon.

Guideline 3 was changed to allow alphanumeric characters (letters and digits) from the character set to allow compatibility with historical usage. Historical practice allows the use of digits wherever practical; and there are no portability issues that would prohibit the use of digits. In fact, from an internationalization viewpoint, digits (being nonlanguage dependent) are preferable over letters (a ''`-2`'' is intuitively self-explanatory to any user, while in the ''`-f` *filename*'' the letter `f` is a mnemonic aid only to speakers of Latin based languages where ''filename'' happens to translate to a word that begins with `f`. Since guideline 3 still retains the word ''single,'' multidigit options are not allowed. Instances of historical utilities that used them have been marked obsolescent in this standard, with the numbers being changed from option names to option-arguments.

It is difficult to come up with a satisfactory solution to the problem of namespace in option characters. When the POSIX.2 group desired to extend the historical `cc` utility to accept C Standard {7} programs, it found that all of the portable alphabet was already in use by various vendors. Thus, it had to devise a new name, `c89`, rather than something like `cc -X`. There were suggestions that implementors be restricted to providing extensions through various means (such as using a plus-sign as the option delimiter or using option characters outside the alphanumeric set) that would reserve all of the remaining alphanumeric characters for future POSIX standards. These approaches were resisted because they lacked the historical style of UNIX. Furthermore, if a vendor-provided option should become commonly used in the industry, it would be a candidate for standardization. It would be desireable to standardize such a feature using existing practice for the syntax (the semantics can be standardized with any syntax). This would not be possible if the syntax was one reserved for the vendor. However, since the standardization process may lead to minor changes in the semantics, it may prove to be better for a vendor to use a syntax that will not be affected by standardization. As a compromise, the following statements are made by the developers of POSIX.2:

- In future revisions to this standard, and in other POSIX standards, every attempt will be made to develop new utilities and features that conform to the Utility Syntax Guidelines.

- Future extensions and additions to POSIX standards will not use the `-W` (capital W) option. This option is forever reserved to implementors for extensions, in a manner reminiscent of the option's use in historical versions of the `cc` utility. The other alphanumeric characters are subject to standardization in the future, based on historical usage.

Implementors should be cognizant of these intentions and aware that future POSIX working groups will offer little sympathy to vendors with extensions in conflict with future drafts. In the first version of POSIX.2, vendors held a virtual veto power when conflicts arose with their extensions; in the future, POSIX working groups may be less concerned about preserving isolated extensions that conflict with these statements of intent.

Guideline 8 includes the concept of comma-separated lists in a single argument. It is up to the utility to parse such a list itself because *getopt* () just returns the single string. This situation was retained so that certain historical utilities wouldn't 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 lists, it should avoid generating noninteger values through one of the means that is influenced by setting the **LC_NUMERIC** variable [such as `awk`, `bc`, `printf`, or *printf* ()].

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 this standard does not specify any options; implementations may provide options as extensions to this standard. The standard utilities that do not support Guideline 10 indicate that fact in the Options subclause of the utility description.

Guideline 11 was modified to clarify that the order of different options should not matter relative to one another. However, 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 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 shall not be used for other purposes in the file reading/writing [but not *stat* ()ing, *unlink* ()ing, `touch`ing, etc.] utilities. All information concerning actual treatment of the `"−"` operand is found in the individual utility clauses.

An area of concern that was expressed during the balloting process was that as implementations mature implementation-defined utilities and implementation-defined utility options will result. The notion was expressed that there needed to be a standard way, say an environment variable or some such mechanism, to identify implementation-defined utilities separately from standard utilities that may have the same name. It was decided that there already exist several ways of dealing with this situation and that it is outside of the scope of the standard to attempt to standardize in the area of nonstandard items. A method that exists on some historical implementations is the use of the so-called `/local/bin` or `/usr/local/bin` directory to separate local or additional copies or versions of utilities. Another method that is also used is to isolate utilities into completely separate domains. Still another method to ensure that the desired utility is being used is to request the utility by its full pathname. There are, to be sure, many approaches to this situation; the examples given above serve to illustrate that there is more than one.

2.11 Utility Description Defaults

This clause describes all of the subclauses used within the utility clauses in Section 4 and the other sections that describe standard utilities. It describes:

(1) Intended usage of the subclause.

(2) Global defaults that affect all the standard utilities.

2.11.0.1 Utility Description Defaults Rationale. *(This subclause is not a part of P1003.2)*

This clause is arranged with headings in the same order as all the utility descriptions. It is a collection of related and unrelated information concerning:

(1) The default actions of utilities.

(2) The meanings of notations used in the standard that are specific to individual utility subclauses.

Although this material may seem out of place in Section 2, it is important that this information appear before any of the utilities to be described later. Unfortunately, since the utilities are split into multiple major sections (chapters), this information could not be placed into any one of those sections without confusing cross references.

2.11.1 Synopsis

The Synopsis subclause summarizes the syntax of the calling sequence for the utility, including options, option-arguments, and operands. Standards for utility naming are described in 2.10.2; for describing the utility's arguments in 2.10.1.

2.11.2 Description

The Description subclause describes the actions of the utility. If the utility has a very complex set of subcommands or its own procedural language, an Extended Description subclause is also provided. Most explanations of optional functionality are omitted here, as they are usually explained in the Options subclause.

Some utilities in this standard are described in terms of equivalent POSIX.1 {8} functionality. As explained in 1.1, a fully conforming POSIX.1 {8} base is not a prerequisite for this standard. When specific functions are cited, the underlying operating system shall provide equivalent functionality and all side effects associated with successful execution of the function. The treatment of errors and intermediate results from the individual functions cited are generally not specified by this standard. See the utility's Exit Status and Consequences of Errors subclauses for all actions associated with errors encountered by the utility.

2.11.3 Options

The Options subclause describes the utility options and option-arguments, and how they modify the actions of the utility. Standard utilities that have options either fully comply with the 2.10.2 or describe all deviations. Apparent disagreements between functionality descriptions in the Options and Description (or Extended Description) subclauses are always resolved in favor of the Options subclause.

Each Options subclause that uses the phrase ‘‘The ... utility shall conform to the utility argument syntax guidelines ...’’ refers only to the use of the utility as specified by this standard; implementation extensions should also conform to the guidelines, but may allow exceptions for historical practice.

Unless otherwise stated in the utility description, when given an option unrecognized by the implementation, or when a required option-argument is not provided, standard utilities shall issue a diagnostic message to standard error and exit with a nonzero exit status.

Default Behavior: When this subclause is listed as ‘‘None,’’ it means that the implementation need not support any options. Standard utilities that do not accept options, but that do accept operands, shall recognize `"--"` as a first argument to be discarded.

2.11.3.1 Options Rationale. *(This subclause is not a part of P1003.2)*

Although it has not always been possible, the working group has tried to avoid repeating information and therefore reduced the risk that the duplicate explanations are somehow modified to be out of sync.

The requirement for recognizing `--` is because portable applications need a way to shield their operands from any arbitrary options that the implementation may provide as an extension. For example, if the standard utility `foo` is listed as taking no options, and the application needed to give it a pathname with a leading hyphen, it could safely do it as:

```
foo -- -myfile
```

and avoid any problems with `-m` used as an extension.

2.11.4 Operands

The Operands subclause describes the utility operands, and how they affect the actions of the utility. Apparent disagreements between functionality descriptions in the Operands and Description (or Extended Description) subclauses are always resolved in favor of the Operands subclause.

If an operand naming a file can be specified as `-`, which means to use the standard input instead of a named file, this shall be explicitly stated in this subclause. Unless otherwise stated, the use of multiple instances of `-` to mean standard input in a single command produces unspecified results.

Unless otherwise stated, the standard utilities that accept operands shall process those operands in the order specified in the command line.

Default Behavior: When this subclause is listed as ‘‘None,’’ it means that the implementation need not support any operands.

2.11.4.1 Operands Rationale. *(This subclause is not a part of P1003.2)*

This usage of – is never shown in the Synopsis. Similarly, this usage of -- is never shown.

The requirement for processing operands in command line order is to avoid a ‘‘WeirdNIX’’ utility that might choose to sort the input files alphabetically, by size, or by directory order. Although this might be acceptable for some utilities, in general the programmer has a right to know exactly what order will be chosen.

Some of the standard utilities take multiple *file* operands and act as if they were processing the concatenation of those files. For example,

`asa file1 file2` and `cat file1 file2 | asa`

have similar results when questions of file access, errors, and performance are ignored. Other utilities, such as `grep` or `wc`, have completely different results in these two cases. This latter type of utility is always identified in its Description or Operands subclauses, whereas the former is not. Although it might be possible to create a general assertion about the former case, the following points must be addressed:

— Access times for the files might be different in the operand case versus the `cat` case.

— The utility may have error messages that are cognizant of the input file name and this added value should not be suppressed. (As an example, `awk` sets a variable with the file name at each file boundary.)

2.11.5 External Influences

The External Influences subclause describes all input data that is specified by the invoker, data received from the environment, and other files or databases that may be used by the utility. There are four subclauses that contain all the substantive information about external influences; because of this, this level of header is always left blank.

Certain of the standard utilities describe how they can invoke other utilities or applications, such as by passing a command string to the command interpreter. The external requirements of such invoked utilities are not described in the subclause concerning the standard utility that invokes them.

2.11.5.1 Standard Input

The Standard Input subclause describes the standard input of the utility. This subclause is frequently merely a reference to the following subclause, because many utilities treat standard input and input files in the same manner. Unless otherwise stated, all restrictions described in Input Files apply to this subclause as well.

Use of a terminal for standard input may cause any of the standard utilities that read standard input to stop when used in the background. For this reason, applications should not use interactive features in scripts to be placed in the background.

The specified standard input format of the standard utilities shall not depend on the existence or value of the environment variables defined in this standard, except as provided by this standard.

Default Behavior: When this subclause is listed as ''None,'' it means that the standard input shall not be read when the utility is used as described by this standard.

2.11.5.1.1 Standard Input Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Standard Input Format in previous drafts to better reflect its role in describing the existence and usage of the file, in addition to its format.

2.11.5.2 Input Files

The Input Files subclause describes the files, other than the standard input, used as input by the utility. It includes files named as operands and option-arguments as well as other files that are referred to, such as startup/initialization files, databases, etc. Commonly-used files are generally described in one place and cross-referenced by other utilities.

Some of the standard utilities, such as filters, process input files a line or a block at a time and have no restrictions on the maximum input file size. Some utilities may have size limitations that are not as obvious as file space or memory limitations. Such limitations should reflect resource limitations of some sort, not arbitrary limits set by implementors. Implementations shall define in the conformance documentation those utilities that are limited by constraints other than file system space, available memory, and other limits specifically cited by this standard, and identify what the constraint is, and indicate a way of estimating when the constraint would be reached. Similarly, some utilities descend the directory tree (recursively). Implementations shall also document any limits that they may have in descending the directory tree that are beyond limits cited by this standard.

When a standard utility reads a seekable input file and terminates without an 1
error before it reaches end-of-file, the utility shall ensure that the file offset in the 1
open file description is properly positioned just past the last byte processed by the 1

utility. For files that are not seekable, the state of the file offset in the open file description for that file is unspecified.

When an input file is described as a *text file*, the utility produces undefined results if given input that is not from a text file, unless otherwise stated. Some utilities (e.g., `make`, `read`, `sh`, etc.) allow for continued input lines using an escaped `<newline>` convention; unless otherwise stated, the utility need not be able to accumulate more than {LINE_MAX} bytes from a set of multiple, continued input lines. If a utility using the escaped `<newline>` convention detects an end-of-file condition immediately after an escaped `<newline>`, the results are unspecified.

Record formats are described in a notation similar to that used by the C language function, *printf*(). See 2.12 for a description of this notation.

Default Behavior: When this subclause is listed as ‘‘None,’’ it means that no input files are required to be supplied when the utility is used as described by this standard.

2.11.5.2.1 Input Files Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Input File Formats in previous drafts to better reflect its role in describing the existence and usage of the files, in addition to their format.

The description of file offsets answers the question: Are the following three commands equivalent?

```
tail -n +2 file
(sed -n 1q; cat) < file
cat file | (sed -n 1q; cat)
```

The answer is that a conforming application cannot assume they are equivalent. The second command is equivalent to the first only when the file is seekable. In the third command, if the file offset in the open file description were not unspecified, `sed` would have to be implemented so that it read from the pipe one byte at a time or it would have to employ some method to seek backwards on the pipe. Such functionality is not defined currently in POSIX.1 {8} and does not exist on all historical systems. Other utilities, such as `head`, `read`, and `sh`, have similar properties, so the restriction is described globally in this clause. A future revision to this standard may require that the standard utilities leave the file offset in a consistent state for pipes as well as regular files.

The description of conformance documentation about file sizes follows many changes of direction by the working group. Originally, there appeared a limit, {ED_FILE_MAX}, that hoped to impose a minimum file size on `ed`, which has been historically limited to relatively small files. This received objections from various members who said that such a limit merely invited sloppy programming; there should be no limits to a ‘‘well-written’’ `ed`. Thus, Draft 8 removed the limit and inserted rationale that this meant `ed` would have to process files of virtually unlimited size. (Surprisingly, no objections or comments were received about that sentence.) However, in discussing the matter with representatives of POSIX.3, it