-- --

IEEE P1003.2 Draft 11.2 – September 1991

turned out that omitting the limit meant that a corresponding test assertion would also be omitted and no test suite could legitimately stress `ed` with large files. It quickly became clear that restrictions applied to other utilities as well and a solution was needed.

It is not possible for this standard to judge which utilities are in the category with arbitrary file size limits; this would impose too much on implementors. Therefore, the burden is placed on implementors to publicly document any limitations and the resulting pressure in the marketplace should keep most implementations adequate for most portable applications. Typically, larger systems would have larger limits than smaller systems, but since price typically follows function, the user can select a machine that handles his/her problems reasonably given such information. The working group considered adding a limit in 2.13.1 for every file-oriented utility, but felt these limits would not actually be used by real applications and would reduce consensus. This is particularly true for utilities, such as possibly `awk` or `yacc`, that might have rather complex limits not directly related to the actual file size.

The definition of *text file* (see 2.2.2.151) is strictly enforced for input to the standard utilities; very few of them list exceptions to the undefined results called for here. (Of course, ‘‘undefined’’ here does not mean that existing implementations necessarily have to change to start indicating error conditions. Conforming applications cannot rely on implementations succeeding or failing when nontext files are used.)

The utilities that allow line continuation are generally those that accept input languages, rather than pure data. It would be unusual for an input line of this type to exceed {LINE_MAX} bytes and unreasonable to require that the implementation allow unlimited accumulation of multiple lines, each of which could reach {LINE_MAX}. Thus, for a portable application the total of all the continued lines in a set cannot exceed {LINE_MAX}.

The format description is intended to be sufficiently rigorous to allow other applications to generate these input files. However, since `<blank>`s can legitimately be included in some of the fields described by the standard utilities, particularly in locales other than the POSIX Locale, this intent is not always realized.

2.11.5.3 Environment Variables

The Environment Variables subclause lists what variables affect the utility's execution.

The entire manner in which environment variables described in this standard affect the behavior of each utility is described in the Environment Variables subclause for that utility, in conjunction with the global effects of the **LANG** and **LC_ALL** environment variables described in 2.6. The existence or value of environment variables described in this standard shall not otherwise affect the specified behavior of the standard utilities. Any effects of the existence or value of environment variables not described by this standard upon the standard utilities are unspecified.

For those standard utilities that use environment variables as a means for selecting a utility to execute (such as **CC** in `make`), the string provided to the utility shall be subjected to the path search described for **PATH** in 2.6.

Default Behavior: When this subclause is listed as ‘‘None,’’ it means that the behavior of the utility is not directly affected by environment variables described by this standard when the utility is used as described by this standard.

2.11.5.3.1 Environment Variables Rationale. *(This subclause is not a part of P1003.2)*

The global default text about the **PATH** search is overkill in this version of POSIX.2 (prior to the UPE) because only one of the standard utilities specifies variables in this way—`make`'s `$(CC)`, `$(LEX)`, etc. It is described here mostly in anticipation of its heavier usage in POSIX.2a. The description of **PATH** indicates separately that names including slashes do not apply, so they do not apply here either.

2.11.5.4 Asynchronous Events

The Asynchronous Events subclause lists how the utility reacts to such events as signals and what signals are caught.

Default Behavior: When this subclause is listed as ‘‘Default,’’ or it refers to ‘‘the standard action for all other signals; see 2.11.5.4,’’ it means that the action taken as a result of the signal shall be one of the following:

(1) The action is that inherited from the parent according to the rules of inheritance of signal actions defined in POSIX.1 {8}(see 2.9.1), or

(2) When no action has been taken to change the default, the default action is that specified by POSIX.1 {8},or

(3) The result of the utility's execution is as if default actions had been taken.

2.11.5.4.1 Asynchronous Events Rationale. *(This subclause is not a part of P1003.2)*

Because there is no language prohibiting it, a utility is permitted to catch a signal, perform some additional processing (such as deleting temporary files), restore the default signal action (or action inherited from the parent process) and resignal itself.

2.11.6 External Effects

The External Effects subclause describes the effects of the utility on the operational environment, including the file system. There are three subclauses that contain all the substantive information about external effects; because of this, this level of header is usually 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 effects of such invoked utilities are not described in the subclause concerning the standard utility that invokes them.

2.11.6.1 Standard Output

The Standard Output subclause describes the standard output of the utility. This subclause is frequently merely a reference to the following subclause, Output Files, because many utilities treat standard output and output files in the same manner.

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

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.

The specified standard output 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 output shall not be written when the utility is used as described by this standard.

2.11.6.1.1 Standard Output Rationale. *(This subclause is not a part of P1003.2)*

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

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an `awk` or `lex` parser.

2.11.6.2 Standard Error

The Standard Error subclause describes the standard error output of the utility. Only those messages that are purposely sent by the utility are described.

Use of a terminal for standard error may cause any of the standard utilities that write standard error output 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 format of diagnostic messages for most utilities is unspecified, but the language and cultural conventions of diagnostic and informative messages whose format is unspecified by this standard should be affected by the setting of **LC_MESSAGES** .

The specified standard error output of 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 ‘‘Used only for diagnostic messages,’’ it means that, unless otherwise stated, the diagnostic messages shall be sent to the standard error only when the exit status is nonzero and the utility is used as described by this standard.

When this subclause is listed as ‘‘None,’’ it means that the standard error shall not be used when the utility is used as described in this standard.

2.11.6.2.1 Standard Error Rationale. *(This subclause is not a part of P1003.2)*

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

This subclause does not describe error messages that refer to incorrect operation of the utility. Consider a utility that processes program source code as its input. This subclause is used to describe messages produced by a correctly operating utility that encounters an error in the program source code on which it is processing. However, a message indicating that the utility had insufficient memory in which to operate would not be described.

Some compilers have traditionally produced warning messages without returning a nonzero exit status; these are specifically noted in their subclauses. Other utilities are expected to remain absolutely quiet on the standard error if they want to return zero, unless the implementation provides some sort of extension to increase the verbosity or debugging level.

The format descriptions are intended to be sufficiently rigorous to allow post-processing of output by other programs.

2.11.6.3 Output Files

The Output Files subclause describes the files created or modified by the utility. Temporary or system files that are created for internal usage by this utility or other parts of the implementation (spool, log, audit files, etc.) are not described in this, or any, subclause. The utilities creating such files and the names of such files are unspecified. If applications are written to use temporary or intermediate files, they should use the **TMPDIR** environment variable, if it is set and 1
represents an accessible directory, to select the location of temporary files. 1

Implementations shall ensure that temporary files, when used by the standard utilities, are named so that different utilities or multiple instances of the same utility can operate simultaneously without regard to their working directories, or any other process characteristic other than process ID. There are two exceptions to this requirement:

(1) Resources for temporary files other than the namespace (for example, disk space, available directory entries, or number of processes allowed) are not guaranteed.

(2) Certain standard utilities generate output files that are intended as input for other utilities, (for example, `lex` generates `lex.yy.c`) and these cannot have unique names. These cases are explicitly identified in the descriptions of the respective utilities.

Any temporary files created by the implementation shall be removed by the implementation upon a utility's successful exit, exit because of errors, or before termination by any of the SIGHUP , SIGINT , or SIGTERM signals, unless specified otherwise by the utility description.

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 files are created or modified as a consequence of direct action on the part of the utility when the utility is used as described by this standard. However, the utility may create or modify system files, such as log files, that are outside of the utility's normal execution environment.

2.11.6.3.1 Output Files Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Output 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 format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an `awk` or `lex` parser.

Receipt of the SIGQUIT signal should generally cause termination (unless in some debugging mode) that would bypass any attempted recovery actions.

2.11.7 Extended Description

The Extended Description subclause provides a place for describing the actions of very complicated utilities, such as text editors or language processors, which typically have elaborate command languages.

Default Behavior: When this subclause is listed as ''None,'' no further description is necessary.

2.11.8 Exit Status

The Exit Status subclause describes the values the utility shall return to the calling program, or shell, and the conditions that cause these values to be returned. Usually, utilities return zero for successful completion and values greater than zero for various error conditions. If specific numeric values are listed in this subclause, conforming implementations shall use those values for the errors described. In some cases, status values are listed more loosely, such as ''>0.'' A Strictly Conforming POSIX.2 Application shall not rely on any specific value in the range shown and shall be prepared to receive any value in the range.

Unspecified error conditions may be represented by specific values not listed in the standard.

2.11.8.1 Exit Status Rationale. *(This subclause is not a part of P1003.2)*

Note the additional discussion of exit status values in 3.8.2. It describes require- 1
ments for returning exit values > 125. 1

A utility may list zero as a successful return, 1 as a failure for a specific reason, and >1 as ''an error occurred.'' In this case, unspecified conditions may cause a 2 or 3, or other value, to be returned. A Strictly Conforming POSIX.2 Application should be written so that it tests for successful exit status values (zero in this case), rather than relying upon the single specific error value listed in the standard. In that way, it will have maximum portability, even on implementations with extensions.

The working group is aware that the general nonenumeration of errors makes it difficult to write test suites that test the *incorrect* operation of utilities. There are some historical implementations that have expended effort to provide detailed status messages and a helpful environment to bypass or explain errors, such as prompting, retrying, or ignoring unimportant syntax errors; other implementations have not. Since there is no realistic way to mandate system behavior in cases of undefined application actions or system problems—in a manner acceptable to all cultures and environments—attention has been limited to the correct operation of utilities by the conforming application. Furthermore, the portable application does not need detailed information concerning errors that it caused through incorrect usage or that it cannot correct anyway. The high degree of competition in the emerging POSIX marketplace should ensure that users requiring friendly, resilient environments will be able to purchase such without detailed specification in this standard.

There is no description of defaults for this subclause because all of the standard utilities specify something (or explicitly state ''Unspecified'') for Exit Status.

2.11.9 Consequences of Errors

The Consequences of Errors subclause describes the effects on the environment, file systems, process state, etc., when error conditions occur. It does not describe error messages produced or exit status values used.

The many reasons for failure of a utility are generally not specified by the utility descriptions. Utilities may terminate prematurely if they encounter: invalid usage of options, arguments, or environment variables; invalid usage of the complex syntaxes expressed in Extended Description subclauses; difficulties accessing, creating, reading, or writing files; or, difficulties associated with the privileges of the process.

The following shall apply to each utility, unless otherwise stated:

— If the requested action cannot be performed on an operand representing a file, directory, user, process, etc., the utility shall issue a diagnostic message to standard error and continue processing the next operand in sequence, but the final exit status shall be returned as nonzero.

— If the requested action characterized by an option or option-argument cannot be performed, the utility shall issue a diagnostic message to standard error and the exit status returned shall be nonzero.

— When an unrecoverable error condition is encountered, the utility shall exit with a nonzero exit status.

— A diagnostic message shall be written to standard error whenever an error condition occurs.

Default Behavior: When this subclause is listed as ''Default,'' it means that any changes to the environment are unspecified.

2.11.9.1 Consequences of Errors Rationale. *(This subclause is not a part of P1003.2)*

When a utility encounters an error condition several actions are possible, depending on the severity of the error and the state of the utility. Included in the possible actions of various utilities are: deletion of temporary or intermediate work files; deletion of incomplete files; validity checking of the file system or directory.

In Draft 9, most of the Consequences of Errors subclauses were changed to ''Default.'' This is due to the more elaborate description of the default case now carried in this subclause and the fact that most of the standard utilities actually use that default.

2.11.10 Rationale

This subclause provides historical perspective and justification of working group actions concerning the utility.

Examples, Usage

This subclause provides examples and usage of the utility. In some cases certain characters are interpreted as special characters to the shell. In the rest of the standard, these characters are shown without escape characters or quoting (see 3.2). In all examples, however, quoting has been used, showing how sample commands (utility names combined with arguments) could be passed correctly to a shell (see `sh` in 4.56) or as a string to the *system* () function.

History of Decisions Made

This subclause provides historical perspective for decisions that were made.

Unresolved Objections

These subclauses were removed from Draft 10. The Unresolved Objections are maintained in a separate list and do not meet ISO editing requirements for an informative annex.

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

The Rationale subclauses will be moved to Annex E in the final POSIX.2. Some of the subheadings may be collapsed in that document; in these drafts the working group has not always been very rigorous about what is a description of usage versus a history of decisions made, for example. The final rationale will de-emphasize the chronological aspects of working group decisions.

2.12 File Format Notation

The Standard Input, Standard Output, Standard Error, Input Files, and Output Files subclauses of the utility descriptions, when provided, use a syntax to describe the data organization within the files, when that organization is not otherwise obvious. The syntax is similar to that used by the C language *printf* () function, as described in this clause. When used in Standard Input or Input Files subclauses of the utility descriptions, this syntax describes the format that could have been used to write the text to be read, not a format that could be used by the C language *scanf* () function to read the input file.

The description of an individual record is as follows:

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

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

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

escape sequences
Represent nongraphic characters.

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

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

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

Δ Exactly one `<space>` character.

The escape-sequences in Table 2-15 depict the associated action on display devices capable of the action.

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

Table 2-15 – Escape Sequences

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

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

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

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

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

The *flag* characters and their meanings are:

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

\+ The result of a signed conversion always shall begin with a sign (+ or –).

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

The value is to be converted to an ‘‘alternate form.’’ For c, d, i, u, and s conversions, the behavior is undefined. For o conversion, it shall increase the precision to force the first digit of the result to be a zero. For x or X conversion, a nonzero result shall have 0x or 0X prefixed to it, respectively. For e, E, f, g and G conversions, the result shall always contain a radix character, even if no digits follow the radix character. For g and G conversions, trailing zeroes shall not be removed from the result as they usually are.

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

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

The *conversion characters* and their meanings are:

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

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

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

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

`c` The integer argument shall be converted to an *unsigned char* and the resulting byte shall be written.

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

% Write a % character; no argument shall be converted.

In no case does a nonexistent or insufficient *field width* cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be simply expanded to contain the conversion result.

2.12.1 File Format Notation Rationale. *(This subclause is not a part of P1003.2)*

This clause was originally derived from the description of *printf* () in the *SVID*, but it has been updated following the publication of the C Standard {7}. It is not identical to the C Standard's {7}*printf* (), as it deals with integers as being essentially one type, disregarding possible internal differences between *int*, *short*, and *long*. It has also had some of the internal C language dependencies removed (such as the requirement for null-terminated strings).

This standard provides a rigorous description of the format of utility input and output files. It is the intention of this standard that these descriptions be adequate sources of information so that portable applications can use other utilities such as `lex` or `awk` to reliably parse the output of these utilities as their input in, say a pipeline.

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

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

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

the implementation could use a real *printf*() such as

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

and still conform. It would have been possible for the standard to use "%6d\n", but it would have been difficult to pick a number that would have pleased everyone. This notation is thus somewhat like *scanf*() in addition to *printf*().

The *printf*() function was chosen as a model as most of the working group was familiar with it and it was thought that many of the readers would be as well.

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

Many of the output descriptions in this standard use the term *line*, such as:

"%s", <*input line*>

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

In the language at the end of the clause:

> ‘‘In no case does a nonexistent or insufficient *field width* cause truncation of a field; . . . ’’

the term ‘‘field width’’ should not be confused with the term ‘‘precision’’ used in the description of `%s`.

Examples:

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

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

To show π written to 5 decimal places:

"piΔ=Δ%.5f\n", *<value of* π*>*

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

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

2.13 Configuration Values

2.13.1 Symbolic Limits

This clause lists magnitude limitations imposed by a specific implementation. The braces notation, {LIMIT}, is used in this standard to indicate these values, but the braces are not part of the name.

Table 2-16 – Utility Limit Minimum Values

Name	Description	Value	
{POSIX2_BC_BASE_MAX}	The maximum *obase* value allowed by the `bc` utility.	99	
{POSIX2_BC_DIM_MAX}	The maximum number of elements permitted in an array by the `bc` utility.	2048	
{POSIX2_BC_SCALE_MAX}	The maximum *scale* value allowed by the `bc` utility.	99	
{POSIX2_BC_STRING_MAX}	The maximum length of a string constant accepted by the `bc` utility.	1000	
{POSIX2_COLL_WEIGHTS_MAX}	The maximum number of weights that can be assigned to an entry of the LC_COLLATE `order` keyword in the locale definition file; see 2.5.2.2.3.	2	
{POSIX2_EXPR_NEST_MAX}	The maximum number of expressions that can be nested within parentheses by the `expr` utility.	32	
{POSIX2_LINE_MAX}	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing `<newline>`.	2048	
{POSIX2_RE_DUP_MAX}	The maximum number of repeated occurrences of a regular expression permitted when using the interval notation \{*m*, *n*\}; see 2.8.3.3.	255	
{POSIX2_VERSION}	This value indicates the version of the utilities in this standard that are provided by the implementation. It will change with each published version of this standard.	199???	1

The values specified in Table 2-16 represent the lowest values conforming implementations shall provide; and consequently, the largest values on which an application can rely without further enquiries, as described below. These values shall be accessible to applications via the `getconf` utility (see 4.26) and through the interfaces described in 7.8.2, [such as *sysconf* () in the C binding]. The literal names shown in the table apply only to the `getconf` utility; the high-level-language binding shall describe the exact form of each name to be used by the interfaces in that binding.

Implementations may provide more liberal, or less restrictive, values than shown in Table 2-16. These possibly more liberal values are accessible using the symbols in Table 2-17.

Table 2-17 – Symbolic Utility Limits

Name	Description	Minimum Value
{BC_BASE_MAX}	The maximum *obase* value allowed by the `bc` utility.	{POSIX2_BC_BASE_MAX}
{BC_DIM_MAX}	The maximum number of elements permitted in an array by the `bc` utility.	{POSIX2_BC_DIM_MAX}
{BC_SCALE_MAX}	The maximum *scale* value allowed by the `bc` utility.	{POSIX2_BC_SCALE_MAX}
{BC_STRING_MAX}	The maximum length of a string constant accepted by the `bc` utility.	{POSIX2_BC_STRING_MAX}
{COLL_WEIGHTS_MAX}	The maximum number of weights that can be assigned to an entry of the LC_COLLATE `order` keyword in the locale definition file; see 2.5.2.2.3.	{POSIX2_COLL_WEIGHTS_MAX}
{EXPR_NEST_MAX}	The maximum number of expressions that can be nested within parentheses by the `expr` utility.	{POSIX2_EXPR_NEST_MAX}
{LINE_MAX}	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing `<newline>`.	{POSIX2_LINE_MAX}
{RE_DUP_MAX}	The maximum number of repeated occurrences of a regular expression permitted when using the interval notation `\{`*m*`,` *n*`\}`; see 2.8.3.3.	{POSIX2_RE_DUP_MAX}

The functions in 7.8.2 [such as *sysconf* () in the C binding] or the `getconf` utility shall return the value of each symbol on each specific implementation. The value so retrieved shall be the largest, or most liberal, value that shall be available throughout the session lifetime, as determined at session creation. The literal names shown in the table apply only to the `getconf` utility; the high-level-language binding shall describe the exact form of each name to be used by the interfaces in that binding.

All numerical limits defined by POSIX.1 {8}, such as {PATH_MAX}, also apply to this standard. (See POSIX.1 {8} 2.8.) All the utilities defined by this standard are implicitly limited by these values, unless otherwise noted in the utility descriptions.

It is not guaranteed that the application can in fact push a value to the implementation's specified limit in any given case, or at all, as a lack of virtual memory or other resources may prevent this. The limit value indicates only that the implementation does not specifically impose any arbitrary, more restrictive limit.

2.13.1.1 Symbolic Limits Rationale. *(This subclause is not a part of P1003.2)*

This clause grew out of an idea that originated in POSIX.1 {8}, in the form of *sysconf*() and *pathconf*(). (In fact, the same person wrote the original text for both standards.) The idea is that a Strictly Conforming POSIX.2 Application can be written to use the most restrictive values that a minimal system can provide, but it shouldn't have to. The values shown in Table 2-17 represent compromises so that some vendors can use historically-limited versions of UNIX system utilities. They are the highest values that Strictly Conforming POSIX.2 Applications or Conforming POSIX.2 Applications can assume, given no other information.

However, by using `getconf` or *sysconf*(), the elegant application can tailor itself to the more liberal values on some of the specific instances of specific implementations.

There is no explicitly-stated requirement that an implementation provide finite limits for any of these numeric values; the implementation is free to provide essentially unbounded capabilities (where it makes sense), stopping only at reasonable points such as {ULONG_MAX} (from the C Standard {7} via POSIX.1 {8}). Therefore, applications desiring to tailor themselves to the values on a particular implementation need to be ready for possibly huge values; it may not be a good idea to blindly allocate a buffer for an input line based on the value of {LINE_MAX}, for instance. However, unlike POSIX.1 {8}, there is no set of limits in this standard that return a special indication meaning ''unbounded.'' The implementation should always return an actual number, even if the number is very large.

The statement

> ''It is not guaranteed that the application ...

is an indication that many of these limits are designed to ensure that implementors design their utilities without arbitrary constraints related to unimaginative programming. There are certainly conditions under which combinations of options can cause failures that would not render an implementation nonconforming. For example, {EXPR_NEST_MAX} and {ARG_MAX} could collide when expressions are large; combinations of {BC_SCALE_MAX} and {BC_DIM_MAX} could exceed virtual memory.

In POSIX.2, the notion of a limit being guaranteed for the process lifetime, as it is in POSIX.1 {8}, is not as useful to a shell script. The `getconf` utility is probably a process itself, so the guarantee would be valueless. Therefore, POSIX.2 requires the guarantee to be for the session lifetime. This will mean that many vendors will either return very conservative values or possibly implement `getconf` as a built-in.

It may seem confusing to have limits that apply only to a single utility grouped into one global clause. However, the alternative, which would be to disperse them out into their utility description clauses, would cause great difficulty when *sysconf*() and `getconf` were described. Therefore, the working group chose the global approach.

Each language binding could provide symbol names that are slightly different than are shown here. For example, the C binding prefixes the symbols with a leading underscore.

The following comments describe selection criteria for the symbols and their values.

{ARG_MAX}
: This is defined by POSIX.1 {8}. Unfortunately, it is very difficult for a portable application to deal with this value, as it does not know how much of its argument space is being consumed by the user's environment variables.

{BC_BASE_MAX}
{BC_DIM_MAX}
{BC_SCALE_MAX}
: These were originally one value, {BC_SCALE_MAX}, but it was unreasonable to link all three concepts into one limit.

{CHILD_MAX}
: This is defined by POSIX.1 {8}.

{CUT_FIELD_MAX}
: This value was removed from an earlier draft. It represented the maximum length of the *list* argument to the `cut -c` or `-f` options. Since the length is now unspecified, the utility should have to deal with arbitrarily long lists, as long as {ARG_MAX} is not exceeded.

{CUT_LINE_MAX}
: This value was removed from an earlier draft. Historical `cut`s have had input line limits of 1024; this removal therefore mandates that a conforming `cut` shall process files with lines of unlimited length. 1

{DEPTH_MAX}
: This directory-traversing depth limit (which at one time applied to `rm` and `find`) was removed from an earlier draft for two major reasons:

(1) It could be a security problem if utilities searching for files could not descend below a published depth; this would be a semi-reliable means of hiding files from the administrator.

(2) There is no reason a reasonable implementation should have to limit itself in this way.

{ED_FILE_MAX}

This value was removed from an earlier draft. Historical `ed`s have had very small file limits; since {ED_FILE_MAX} is no longer specified, implementations have to document the limits as described in 2.11. It is recommended that implementations set much more reasonable file size limits as they modify `ed` to deal with other features required by POSIX.2.

{ED_LINE_MAX}

This value was removed from an earlier draft. Historical `ed`s have had small input line limits; this removal therefore mandates that a conforming `ed` shall process files with lines of length {LINE_MAX}.

{COLL_WEIGHTS_MAX}

The weights assigned to `order` can be considered as ''passes'' through the collation algorithm.

{EXPR_NEST_MAX}

The value for expression nesting was borrowed from the C Standard {7}.

{FIND_DEPTH_MAX}

This was removed from an earlier draft in favor of a common value, {DEPTH_MAX}.

{FIND_FILESYS_MAX}

This was removed from an earlier draft. It indicated the limit of the number of file systems that `find` could traverse in its search. It was dropped because this standard does not really acknowledge the historical nature of separate file systems.

{FIND_NEWER_MAX}

This value, which allowed `find` to limit the number of `-newer` operands it processed, was deleted from an earlier draft. It was felt to be a vestige of a particular implementation with an incorrect programming algorithm that should not limit applications.

{JOIN_LINE_MAX}

This value was removed from an earlier draft. Historical `join`s have had input line limits of 1024; this removal therefore mandates that a conforming `join` shall process files with lines of length {LINE_MAX}.

{LINE_MAX}

This is a global limit that affects all utilities, unless otherwise noted. The {MAX_CANON} value from POSIX.1 {8} may further limit input lines from terminals. The {LINE_MAX} value was the subject of much debate and is a compromise between those who wished unlimited lines and those who understood that many historical utilities were written with fixed buffers. Frequently, utility writers selected the UNIX system constant BUFSIZ to allocate these buffers; therefore, some utilities were limited to 512 bytes for I/O lines, while others achieved 4096 or greater.

It should be noted that {LINE_MAX} applies only to input line length; there is no requirement in the standard that limits the length of output lines. Utilities such as `awk`, `sed`, and `paste` could theoretically construct lines longer than any of the input lines they received, depending on the options used or the instructions from the application. They are not required to truncate their output to {LINE_MAX}. It is the responsibility of the application to deal with this. If the output of one of those utilities is to be piped into another of the standard utilities, line lengths restrictions will have to be considered; the `fold` utility, among others, could be used to ensure that only reasonable line lengths reach utilities or applications.

{LINK_MAX}
This is defined by POSIX.1 {8}.

{LP_LINE_MAX}
This value was removed from an earlier draft. Since so little is being required for the details of the `lp` utility, it made little sense to specify how long its output lines are. Thus, implementations of `lp` will be expected to deal with lines up to {LINE_MAX}, but whether those lines print sensibly on every device is unspecified.

{MAX_CANON}
This is defined by POSIX.1 {8}.

{MAX_INPUT}
This is defined by POSIX.1 {8}.

{NAME_MAX}
This is defined by POSIX.1 {8}.

{NGROUPS_MAX}
This is defined by POSIX.1 {8}.

{OPEN_MAX}
This is defined by POSIX.1 {8}.

{PATH_MAX}
This is defined by POSIX.1 {8}.

{PIPE_BUF}
This is defined by POSIX.1 {8}.

{RM_DEPTH_MAX}
This was removed from an earlier draft in favor of a common value, {DEPTH_MAX}.

{RE_DUP_MAX}
The value selected is consistent with historical practice.

{SED_PATTERN_MAX}
This symbolic value, the size of the `sed` pattern space, was replaced by a specific value in the `sed` description. It is unlikely that any real application would ever need to access this value symbolically.

{SORT_LINE_MAX}
This was removed from an earlier draft. Now that `cut` and `fold` can handle unlimited-length input lines, a special long input line limit for `sort` is not needed.

There are different limits associated with command lines and input to utilities, depending on the method of invocation. In the case of a C program *exec*-ing a utility, {ARG_MAX} is the underlying limit. In the case of the shell reading a script and *exec*-ing a utility, {LINE_MAX} limits the length of lines the shell is required to process and {ARG_MAX} will still be a limit. If a user is entering a command on a terminal to the shell, requesting that it invoke the utility, {MAX_INPUT} may restrict the length of the line that can be given to the shell to a value below {LINE_MAX}.

2.13.2 Symbolic Constants for Portability Specifications

Table 2-18 – Optional Facility Configuration Values

Name	Description
{POSIX2_C_BIND}	The C language development facilities in Annex A support the C Language Bindings Option (see Annex B).
{POSIX2_C_DEV}	The system supports the C Language Development Utilities Option (see Annex A).
{POSIX2_FORT_DEV}	The system supports the FORTRAN Development Utilities Option (see Annex C).
{POSIX2_FORT_RUN}	The system supports the FORTRAN Runtime Utilities Option (see Annex C).
{POSIX2_LOCALEDEF}	The system supports the creation of locales as described in 4.35.
{POSIX2_SW_DEV}	The system supports the Software Development Utilities Option (see Section 6).

Table 2-18 lists symbols that can be used by the application to determine which optional facilities are present on the implementation. The functions defined in 7.8.2 [such as *sysconf*()] or the `getconf` utility can be used to retrieve the value of each symbol on each specific implementation. The literal names shown in the table apply only to the `getconf` utility; the high-level-language binding shall describe the exact form of each name to be used by the interfaces in that binding.

Each of these symbols shall be considered valid names by the implementation. Each shall be defined on the system with a value of 1 if the corresponding option is supported; otherwise, the symbol shall be undefined.

2.13.2.1 Symbolic Constants for Portability Specifications Rationale. *(This subclause is not a part of P1003.2)*

When an option is supported, `getconf` returns a value of 1. For example, when C development is supported:

```
if [ "$(getconf POSIX2_C_DEV)" -eq 1 ]; then
        echo C supported
fi
```

The *sysconf*() function in the C binding would return 1.

The following comments describe selection criteria for the symbols and their values.

{POSIX2_C_BIND}
{POSIX2_C_DEV}
{POSIX2_FORT_DEV}
{POSIX2_SW_DEV}

These were renamed from _POSIX_* in Draft 9 after it was pointed out that each of the POSIX standards should keep generally in its own namespace.

It is possible for some (usually privileged) operations to remove utilities that support these options, or otherwise render these options unsupported. The header files, the *sysconf*() function, or the `getconf` utility will not necessarily detect such actions, in which case they should not be considered as rendering the implementation nonconforming. A test suite should not attempt tests like:

```
rm /usr/bin/c89
getconf POSIX2_C_DEV
```

{_POSIX_LOCALEDEF}

This symbol was introduced to allow implementations to restrict supported locales to only those supplied by the implementation.