IEEE P1003.2 Draft 11.2 – September 1991

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 error before it reaches end-of-file, the utility shall ensure that the file offset in the open file description is properly positioned just past the last byte processed by the 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 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 represents an accessible directory, to select the location of temporary files.

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: