IEEE P1003.2 Draft 11.2 − September 1991

type specifier does this with no problem and is completely compatible with the historical implementations of the `c` format character when the current locale uses a superset of ISO/IEC 646 {1} as a code set. The `a` type specifier (from the BSD `a` format character) was left as a portable means to dump ASCII [or more correctly ISO/IEC 646 {1} (IRV)] so that headers produced by `pax` could be deciphered even on systems that do not use ISO/IEC 646 {1} as a subset of their base code set.

The use of ** as an indication of continuation of a multibyte character in `c` specifier output was chosen based on seeing an implementation that uses this method. The continuation bytes have to be marked in a way that will not be ambiguous with another single- or multibyte character.

An earlier draft used `−S` and `−n`, respectively, for the `−j` and `−N` options in this draft. These were changed to avoid conflicts with historical implementations.

4.46 `paste` — Merge corresponding or subsequent lines of files

4.46.1 Synopsis

`paste [−s] [−d` *list* `]` *file* . . .

4.46.2 Description

The `paste` utility shall concatenate the corresponding lines of the given input files, and write the resulting lines to standard output.

The default operation of `paste` shall concatenate the corresponding lines of the input files. The `<newline>` character of every line except the line from the last input file shall be replaced with a `<tab>` character.

If an end-of-file condition is detected on one or more input files, but not all input files, `paste` shall behave as though empty lines were read from the file(s) on which end-of-file was detected, unless the `−s` option is specified.

4.46.3 Options

The `paste` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−d` *list* Unless a backslash character appears in *list*, each character in 2
list is an element specifying a delimiter character. If a backslash 2
character appears in *list*, the backslash character and one or more 2
characters following it are an element specifying a delimiter char- 2
acter as described below. These elements specify one or more 2

delimiters to use, instead of the default `<tab>`, to replace the 2
`<newline>` character of the input lines. The elements in *list* 2
shall be used circularly; i.e., when the list is exhausted the first 2
element from the list shall be re-used. When the `-s` option is 2
specified:

— The last `<newline>` character in a file shall not be modified.

— The delimiter shall be reset to the first element of list after each *file* operand is processed.

When the `-s` option is not specified:

— The `<newline>` characters in the file specified by the last *file* operand shall not be modified.

— The delimiter shall be reset to the first element of list each time a line is processed from each file.

If a backslash character appears in *list*, it and the character following it shall be used to represent the following delimiter characters:

`\n` `<newline>` character

`\t` `<tab>` character

`\\` backslash character

`\0` Empty string (not a null character). If `\0` is immediately followed by the character `x`, the character `X`, or any character defined by the LC_CTYPE `digit` keyword (see 2.5.2.1), the results are unspecified.

If any other characters follow the backslash, the results are unspecified.

`-s` Concatenate all of the lines of each separate input file in command line order. The `<newline>` character of every line except the last line in each input file shall be replaced with the `<tab>` character, unless otherwise specified by the `-d` option.

4.46.4 Operands

The following operand shall be supported by the implementation:

file A pathname of an input file. If `-` is specified for one or more of the *file*s, the standard input shall be used; the standard input shall be read one line at a time, circularly, for each instance of `-`. Implementations shall support pasting of at least 12 *file* operands.

4.46.5 External Influences

4.46.5.1 Standard Input

The standard input shall be used only if one or more *file* operands is −. See Input Files.

4.46.5.2 Input Files

The input files shall be text files, except that line lengths shall be unlimited.

4.46.5.3 Environment Variables

The following environment variables shall affect the execution of `paste`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

LC_MESSAGES This variable shall determine the language in which messages should be written.

4.46.5.4 Asynchronous Events

Default.

4.46.6 External Effects

4.46.6.1 Standard Output

Concatenated lines of input files shall be separated by the `<tab>` character (or other characters under the control of the `−d` option) and terminated by a `<newline>` character.

4.46.6.2 Standard Error

Used only for diagnostic messages.

4.46.6.3 Output Files

None.

4.46.7 Extended Description

None.

4.46.8 Exit Status

The `paste` utility shall exit with one of the following values:

- 0 Successful completion.
- >0 An error occurred.

4.46.9 Consequences of Errors

If one or more input files cannot be opened when the `−s` option is not specified, a diagnostic message shall be written to standard error, but no output shall be written to standard output. If the `−s` option is specified, the `paste` utility shall provide the default behavior described in 2.11.9.

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

Examples, Usage

When the escape sequences of the *list* option-argument are used in a shell script, they must be quoted; otherwise, the shell treats the \ as a special character.

Write out a directory in four columns:

```
ls | paste - - - -
```

Combine pairs of lines from a file into single lines:

```
paste -s -d "\t\n" file
```

Portable applications should only use the specific backslash escaped delimiters presented in this standard. Historical implementations treat `\x`, where `x` is not in this list, as `x`, but future implementations are free to expand this list to recognize other common escapes similar to those accepted by `printf` and other standard utilities.

Most of the standard utilities work on text files. The `cut` utility can be used to turn files with arbitrary line lengths into a set of text files containing the same data. The `paste` utility can be used to create (or recreate) files with arbitrary line lengths. For example, if `file` contains long lines:

```
cut -b 1-500 -n file > file1
cut -b 501- -n file > file2
```

creates `file1` (a text file) with lines no longer than 500 bytes (plus the `<newline>` character) and `file2` that contains the remainder of the data from `file`. (Note that `file2` will not be a text file if there are lines in file that are longer than 500 + {LINE_MAX} bytes.) The original file can be recreated from `file1` and `file2` using the command:

```
paste -d "\0" file1 file2 > file
```

The commands 2

```
paste -d "\0" ...    2
paste -d "" ...      2
```

are not necessarily equivalent; the latter is not specified by POSIX.2 and may 2
result in an error. The construct `\0` is used to mean ‘‘no separator’’ because his- 2
torical versions of `paste` did not follow the syntax guidelines and the command 2

```
paste -d"" ...       2
```

could not be handled properly by *getopt* (). 2

History of Decisions Made

Because most of the standards utilities work on text files, `cut` and `paste` are required to process lines of arbitrary length as a means of converting long lines from arbitrary sources into text files and converting processed text files back into files with arbitrary line lengths to interface with those applications that require long lines as input.

4.47 **pathchk** — Check pathnames

4.47.1 Synopsis

`pathchk [−p]` *pathname* . . .

4.47.2 Description

The `pathchk` utility shall check that one or more pathnames are valid (i.e., they could be used to access or create a file without causing syntax errors) and portable (i.e., no filename truncation will result). More extensive portability checks are provided by the `−p` option.

By default, the `pathchk` utility shall check each component of each *pathname* operand based on the underlying file system. A diagnostic shall be written for each *pathname* operand that:

— is longer than {PATH_MAX} bytes (see Pathname Variable Values in POSIX.1 {8}2.9.5),

— contains any component longer than {NAME_MAX} bytes in its containing directory,

— contains any component in a directory that is not searchable, or

— contains any character in any component that is not valid in its containing directory.

The format of the diagnostic message is not specified, but shall indicate the error detected and the corresponding *pathname* operand.

It shall not be considered an error if one or more components of a *pathname* operand do not exist as long as a file matching the pathname specified by the missing components could be created that does not violate any of the checks specified above.

4.47.3 Options

The `pathchk` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−p` Instead of performing checks based on the underlying file system, write a diagnostic for each *pathname* operand that:

— is longer than {_POSIX_PATH_MAX} bytes (see Minimum Values in POSIX.1 {8}2.9.2),

— contains any component longer than {_POSIX_NAME_MAX} bytes, or