--   --

IEEE P1003.2 Draft 11.2 − September 1991

4.62 `test` — Evaluate expression

4.62.1 Synopsis

`test` [*expression*]

`[` [*expression*] `]`

4.62.2 Description

The `test` utility shall evaluate the *expression* and indicate the result of the 1
evaluation by its exit status. An exit status of zero indicates that the expression 1
evaluated as true and an exit status of 1 indicates that the expression evaluated 1
as false. 1

In the second form of the utility, which uses `[ ]`, rather than `test`, the square brackets shall be separate arguments.

4.62.3 Options

The `test` utility shall not recognize the `--` argument in the manner specified by utility syntax guideline 10 in 2.10.2.

Implementations shall not support any options. 1

4.62.4 Operands

All operators and elements of primaries shall be presented as separate arguments 2
to the `test` utility.

The following primaries can be used to construct *expression* :

- `-b` *file* True if *file* exists and is a block special file.
- `-c` *file* True if *file* exists and is a character special file.
- `-d` *file* True if *file* exists and is a directory.
- `-e` *file* True if *file* exists.
- `-f` *file* True if *file* exists and is a regular file.
- `-g` *file* True if *file* exists and its set group ID flag is set.
- `-n` *string* True if the length of *string* is nonzero.
- `-p` *file* True if *file* is a named pipe (FIFO).
- `-r` *file* True if *file* exists and is readable.

`-s` *file* True if *file* exists and has a size greater than zero.

`-t` *file_descriptor*
True if the file whose file descriptor number is *file_descriptor* is open and is associated with a terminal.

`-u` *file* True if *file* exists and its set-user- ID flag is set.

`-w` *file* True if *file* exists and is writable. True shall indicate only that the write flag is on. The *file* shall not be writable on a read-only file system even if this test indicates true.

`-x` *file* True if *file* exists and is executable. True shall indicate only that the execute flag is on. If *file* is a directory, true indicates that *file* can be searched.

`-z` *string* True if the length of string *string* is zero.

string True if the string *string* is not the null string.

s1 = *s2* True if the strings *s1* and *s2* are identical.

s1 != *s2* True if the strings *s1* and *s2* are not identical.

n1 `-eq` *n2* True if the integers *n1* and *n2* are algebraically equal.

n1 `-ne` *n2* True if the integers *n1* and *n2* are not algebraically equal.

n1 `-gt` *n2* True if the integer *n1* is algebraically greater than the integer *n2*.

n1 `-ge` *n2* True if the integer *n1* is algebraically greater than or equal to the integer *n2*.

n1 `-lt` *n2* True if the integer *n1* is algebraically less than the integer *n2*.

n1 `-le` *n2* True if the integer *n1* is algebraically less than or equal to the integer *n2*.

A primary can be preceded by the ! operator to complement its test, as described 1
below. 1

The primaries with two elements of the form: 2

−*primary_operator* *primary_operand* 2

are known as *unary primaries.* The primaries with three elements in either of the 2
two forms: 2

primary_operand −*primary_operator* *primary_operand* 2

primary_operand *primary_operator* *primary_operand* 2

are known as *binary primaries.* Additional implementation-defined operators and 2
primary_operator s may be provided by implementations. They shall be of the 2
form −*operator* where the first character of *operator* is not a digit. The additional 2
implementation-defined operators ‘‘`(`’’and ‘‘`)`’’may also be provided by implemen- 2
tations. 2

The algorithm for determining the precedence of the operators and the return value that shall be generated is based on the number of arguments presented to `test`. (However, when using the `[...]` form, the right-bracket final argument shall not be counted in this algorithm.) In the following list, `$1`, `$2`, `$3`, and `$4` represent the arguments presented to `test`.

0 arguments:

Exit false (1).

1 argument:

Exit true (0) if `$1` is not null; otherwise, exit false.

2 arguments:

- If `$1` is `!`, exit true if `$2` is null, false if `$2` is not null.
- If `$1` is a unary primary, exit true if the unary test is true, false if the unary test is false.
- Otherwise, produce unspecified results.

3 arguments:

- If `$2` is a binary primary, perform the binary test of `$1` and `$3`.
- If `$1` is `!`, negate the two-argument test of `$2` and `$3`.
- Otherwise, produce unspecified results.

4 arguments:

- If `$1` is `!`, negate the three-argument test of `$2`, `$3`, and `$4`.
- Otherwise, the results are unspecified.

>4 arguments:

The results are unspecified.

4.62.5 External Influences

4.62.5.1 Standard Input

None.

4.62.5.2 Input Files

None.

4.62.5.3 Environment Variables

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

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).

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

4.62.5.4 Asynchronous Events

Default.

4.62.6 External Effects

4.62.6.1 Standard Output

None.

4.62.6.2 Standard Error

Used only for diagnostic messages.

4.62.6.3 Output Files

None.

4.62.7 Extended Description

None.

4.62.8 Exit Status

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

0	*expression* evaluated to true.
1	*expression* evaluated to false or *expression* was missing.
>1	An error occurred.

4.62.9 Consequences of Errors

Default.

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

Examples, Usage

Editor's Note: The rationale has been rearranged quite a bit. Only new, not moved, text has been diffmarked.

Historical systems have supported more than four arguments, but there has been a fundamental disagreement between BSD and System V on certain combinations of arguments. Since no accommodation could be reached between the two versions of `test` without breaking numerous applications, the version of `test` in POSIX.2 specifies only the relatively simple tests and relies on the syntax of the shell command language for the construction of more complex expressions. Using the POSIX.2 rules produces completely reliable, portable scripts, which is not always possible using either of the historical forms. Some of the historical behavior is described here to aid conversion of scripts with complex `test` expressions.

Both BSD and System V support the combining of primaries with the following constructs:

expression1 `-a` *expression2*	True if both *expression1* and *expression2* are true.
expression1 `-o` *expression2*	True if at least one of *expression1* and *expression2* are true.
`(` *expression* `)`	True if *expression* is true.

In evaluating these more complex combined expressions, the following precedence rules are used:

- The unary primaries have higher precedence than the algebraic binary primaries.
- On BSD systems, the unary primaries have higher precedence than the string binary primaries. On System V systems, the unary primaries have lower precedence than the string binary primaries.

— The unary and binary primaries have higher precedence than the unary *string* primary.

— The ! operator has higher precedence than the −a operator and the −a operator has higher precedence than the −o operator.

— The −a and −o operators are left associative.

— The parentheses can be used to alter the normal precedence and associativity.

The following guidance is offered for the use of the historical expressions:

— Scripts should be careful when dealing with user-supplied input that could be confused with primaries and operators. Unless the application writer knows all the cases that produce input to the script, invocations like:

```
test "$1" -a "$2"
```

should be written as:

```
test "$1" && test "$2"
```

to avoid problems if a user-supplied values such as `$1` set to ! and `$2` set to the null string. That is, in cases where portability between implementations based on BSD and System V systems is of concern, replace:

```
test expr1 -a expr2
```

with:

```
test expr1 && test expr2
```

and replace:

```
test expr1 -o expr2
```

with:

```
test expr1 || test expr2
```

but note that, in `test`, −a has higher precedence than −o while && and || have equal precedence in the shell.

Parentheses or braces can be used in the shell command language to effect grouping. Historical `test` implementations also support parentheses, but they must be escaped when using `sh`; for example:

```
test \( expr1 -a expr2 \) -o expr3
```

This command is not always portable. The following form can be used instead:

```
( test expr1 && test expr2 ) || test expr3
```

— The two commands:

```
test "$1"
test ! "$1"
```

could not be used reliably on historical systems. Unexpected results would occur if such a *string* expression were used and `$1` expanded to `!`, `(`, or a known unary primary. Better constructs were:

```
test -n "$1"
test -z "$1"
```

respectively. These suggested replacements have always worked on historical BSD-based implementations, and work on historical System V-based implementations as long as `$1` does not expand to `=` or `!=`. Using the POSIX.2 rules, any of the four forms shown will work for any possible value of `$1`.

— Historical systems were also unreliable given the common construct:

```
test "$response" = "expected string"
```

One of the following was a more reliable form:

```
test "X$response" = "Xexpected string"
test "expected string" = "$response"
```

Note that the second form assumes that `expected string` could not be confused with any any unary primary. If `expected string` starts with `-`, `(`, `!`, or even `=`, the first form should be used instead. Using the POSIX.2 rules, any of the three comparison forms is reliable, given any input. (However, note that the strings are quoted in all cases.)

The BSD and System V versions of `−f` are not the same. The BSD definition was:

`−f` *file* True if *file* exists and is not a directory.

The *SVID* version (true if the file exists and is a regular file) was chosen for this standard because its use is consistent with the `−b`, `−c`, `−d`, and `−p` operands (*file* exists and is a specific file type).

The `−e` primary, possessing similar functionality to that provided by the C-shell, was added because it provides the only way for a shell script to find out if a file exists without trying to open the file. (Since implementations are allowed to add additional file types, a portable script cannot use:

```
test -b foo -o -c foo -o -d foo -o -f foo -o -p foo
```

to find out if `foo` is an existing file.) On historical BSD systems, the existence of a file could be determined by:

```
test -f foo -o -d foo
```

but there was no easy way to determine that an existing file was a regular file. An earlier draft used the KornShell `−a` primary (with the same meaning), but this was changed to `−e` because there were concerns about the high probability of humans confusing the `−a` primary with the `−a` binary operator.

History of Decisions Made

The `−a` and `−o` binary operators and the grouping parentheses were omitted from POSIX.2 due to a difference between existing implementations of the `test` utility in the precedence of the binary primaries `=` and `!=` compared to the unary primaries `−b`, `−c`, `−d`, `−f`, `−g`, `−n`, `−p`, `−r`, `−s`, `−t`, `−u`, `−w`, `−x`, and `−z`. On BSD, Version 7, PWB, and 32V systems the unary primaries have higher precedence than the binary operators; on System III and System V implementations, the binary operators `=` and `!=` have higher precedence. The change was apparently made for System III so that the construct:

```
test "$1" = "$2"
```

could be made to work even if `$1` started with `-`. It is believed that this change was a mistake because:

— It is not a complete solution; if `$1` expands to `(` or `!`, it still will not work.

— It makes it impossible to use the unary primaries `−n` and `−z` to test for a null string if there is any chance that the string will expand to `=`.

— More importantly, there was the well known workaround of specifying:

```
test X"$1" = X"$2"
```

that always worked.

Unfortunately, when the `=` and `!=` binary primaries were given precedence over the unary primaries, there was no workaround provided for scripts that wanted to reliably specify something like:

```
test -n "$1"
```

because if `$1` expands to `=`, it gives a syntax error.

There was some discussion of outlawing the System V behavior and requiring the more logical precedence that originated in its predecessors and remains in BSD-based systems. However, there are simply too many historical applications that would break if System V were required to make this change; this number dwarfed the number of scripts using combination logic that would then no longer be strictly portable.

POSIX.2 requires that if `test` is called with one, two, three, or four operands it correctly interprets the expression even if there is an alternate syntax tree that could lead to a syntax error. It eliminates the requirement that many string comparisons be protected with leading characters, such as

```
test X"$1" = X"$2"
```

and allows the single-argument *string* form to be used with all possible inputs.

The following examples show some of the changes that are required to be made to make historical BSD and System V-based implementations of `test` conform to this standard:

`test -d =`	POSIX.2	True if there is a directory named `=`

	BSD	True if there is a directory named =
	System V	Syntax error; = needs two operands
`test -d = -f`	POSIX.2	False
	BSD	Syntax error; it expects `-a` or `-o` after `-d =`
	System V	False

Implementations are prohibited from extending `test` with options because it would make the ‘`test` *string*’’ case ambiguous for inputs that might match an extended option. Implementations can add primaries and operators, as indicated.

The following options were not included in POSIX.2, although they are provided by some historical implementations, since these facilities and concepts are not supported by POSIX.1 {8}, nor defined in POSIX.2. These operands should not be used by new implementations for other purposes.

`-h` *file* True if *file* exists and is a symbolic link.

`-k` *file* True if *file* exists and its sticky bit is set.

`-L` *file* True if *file* is a symbolic link.

`-C` *file* True if *file* is a contiguous file.

`-S` *file* True if *file* is a socket.

`-V` *file* True if *file* is a version file.

The following option was not included because it was undocumented in most implementations, has been removed from some implementations (including System V), and the functionality is provided by the shell (see 3.6.2).

`-l` *string* The length of the string *string*.

The `-b`, `-c`, `-g`, `-p`, `-u`, and `-x` operands are derived from the *SVID*; historical BSD does not provide them. The `-k` operand is derived from System V; historical BSD does not provide it.

On historical BSD systems, `test -w` *directory* always returned false because `test` tried to open the directory for writing, which always fails.

Some additional primaries newly invented or from the KornShell appeared in an earlier draft as part of the Conditional Command (`[[ ]]`): *s1* > *s2*, *s1* < *s2*, *str* = *pattern*, *str* `!=` *pattern*, *f1* `-nt` *f2*, *f1* `-ot` *f2*, and *f1* `-ef` *f2*. They were not carried forward into the `test` utility when the Conditional Command was removed from the shell because they have not been included in the `test` utility built into historical implementations of the `sh` utility.

The `-t` *file_descriptor* primary is shown with a mandatory argument because the grammar is ambiguous if it can be omitted. Historical implementations have allowed it to be omitted, providing a default of 1.

4.63 `touch` — Change file access and modification times

4.63.1 Synopsis

`touch [−acm][ −r` *ref_file* | `−t` *time* `]` *file* ...

Obsolescent Version :

`touch [−acm][`*date_time* `]` *file* ...

4.63.2 Description

The `touch` utility shall change the modification and/or access times of files. The modification time is equivalent to the value of the *st_mtime* member of the *stat* structure for a file, as described in POSIX.1 {8};the access time is equivalent to the value of *st_atime* .

The time used can be specified by the `−t` *time* option-argument, the corresponding time field(s) of the file referenced by the `−r` *ref_file* option-argument, or the *date_time* operand, as specified in the following subclauses. If none of these are specified, `touch` shall use the current time [the value returned by the equivalent of the POSIX.1 {8} *time* () function].

For each *file* operand, `touch` shall perform actions equivalent to the following functions defined in POSIX.1 {8}:

(1) If *file* does not exist, a *creat* () function call is made with the *file* operand used as the *path* argument and the value of the bitwise inclusive OR of S_IRUSR , S_IWUSR , S_IRGRP , S_IWGRP , S_IROTH , and S_IWOTH used as the *mode* argument.

(2) The *utime* () function is called with the following arguments:

 (a) The *file* operand is used as the *path* argument.

 (b) The *utimbuf* structure members *actime* and *modtime* are determined as described under 4.63.3.

4.63.3 Options

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

The following options shall be supported by the implementation:

`−a` Change the access time of *file* . Do not change the modification time unless `−m` is also specified.