GNU `sh-utils`

A set of shell utilities
for version 2.0.11, 18 October 2000

David MacKenzie et al.

1 Introduction

First of all, this manual is incomplete. The `stty` section, in particular, needs substantial reorganization and additional explanatory text before it will be up to the standard of other GNU manuals. Explanatory text in general is lacking; the manual presently assumes you pretty much know what to do, and just need to be reminded of how. Thus, if you are interested, please get involved in improving this manual. The entire GNU community will benefit.

Some of these programs are useful only when writing shell scripts; utilities like these are, in fact, the "language" of shell scripts (to a great extent). Others are occasionally useful interactively.

The GNU shell utilities are mostly compatible with the POSIX.2 standard.

Please report bugs to '`bug-sh-utils@gnu.org`'. Remember to include the version number, machine architecture, input files, and any other information needed to reproduce the bug: your input, what you expected, what you got, and why it is wrong. Diffs are welcome, but please include a description of the problem as well, since this is sometimes difficult to infer. See section "Bugs" in *GNU CC*.

This manual was originally derived from the Unix man pages in the distribution, which were written by David MacKenzie and updated by Jim Meyering. What you are reading now is the authoritative documentation for these utilities; the man pages are no longer being maintained. François Pinard did the initial conversion to Texinfo format. Karl Berry did the indexing, some reorganization, and editing of the results. Richard Stallman contributed his usual invaluable insights to the overall process.

2 Common options

Certain options are available in all these programs. Rather than writing identical descriptions for each of the programs, they are described here. (In fact, every GNU program accepts (or should accept) these options.)

Many of these programs take arbitrary strings as arguments. In those cases, '`--help`' and '`--version`' are taken as these options only if there is one and exactly one command line argument.

'`--help`' Print a usage message listing all available options, then exit successfully.

'`--version`'
Print the version number, then exit successfully.

3 Date input formats

First, a quote:

> Our units of temporal measurement, from seconds on up to months, are so complicated, asymmetrical and disjunctive so as to make coherent mental reckoning in time all but impossible. Indeed, had some tyrannical god contrived to enslave our minds to time, to make it all but impossible for us to escape subjection to sodden routines and unpleasant surprises, he could hardly have done better than handing down our present system. It is like a set of trapezoidal building blocks, with no vertical or horizontal surfaces, like a language in which the simplest thought demands ornate constructions, useless particles and lengthy circumlocutions. Unlike the more successful patterns of language and science, which enable us to face experience boldly or at least level-headedly, our system of temporal calculation silently and persistently encourages our terror of time.
>
> . . . It is as though architects had to measure length in feet, width in meters and height in ells; as though basic instruction manuals demanded a knowledge of five different languages. It is no wonder then that we often look into our own immediate past or future, last Tuesday or a week from Sunday, with feelings of helpless confusion. . . .
>
> — Robert Grudin, *Time and the Art of Living*.

This section describes the textual date representations that GNU programs accept. These are the strings you, as a user, can supply as arguments to the various programs. The C interface (via the `getdate` function) is not described here.

Although the date syntax here can represent any possible time since zero A.D., computer integers are not big enough for such a (comparatively) long time. The earliest date semantically allowed on Unix systems is midnight, 1 January 1970 UCT.

3.1 General date syntax

A *date* is a string, possibly empty, containing many items separated by whitespace. The whitespace may be omitted when no ambiguity arises. The empty string means the beginning of today (i.e., midnight). Order of the items is immaterial. A date string may contain many flavors of items:

- calendar date items
- time of the day items
- time zone items
- day of the week items
- relative items
- pure numbers.

We describe each of these item types in turn, below.

A few numbers may be written out in words in most contexts. This is most useful for specifying day of the week items or relative items (see below). Here is the list: '`first`' for 1, '`next`' for 2, '`third`' for 3, '`fourth`' for 4, '`fifth`' for 5, '`sixth`' for 6, '`seventh`' for 7, '`eighth`' for 8, '`ninth`' for 9, '`tenth`' for 10, '`eleventh`' for 11 and '`twelfth`' for 12. Also, '`last`' means exactly −1.

When a month is written this way, it is still considered to be written numerically, instead of being "spelled in full"; this changes the allowed strings.

Alphabetic case is completely ignored in dates. Comments may be introduced between round parentheses, as long as included parentheses are properly nested. Hyphens not followed by a digit are currently ignored. Leading zeros on numbers are ignored.

3.2 Calendar date items

A *calendar date item* specifies a day of the year. It is specified differently, depending on whether the month is specified numerically or literally. All these strings specify the same calendar date:

```
1972-09-24     # ISO 8601.
72-9-24        # Assume 19xx for 69 through 99,
               # 20xx for 00 through 68.
72-09-24       # Leading zeros are ignored.
9/24/72        # Common U.S. writing.
24 September 1972
24 Sept 72     # September has a special abbreviation.
24 Sep 72      # Three-letter abbreviations always allowed.
Sep 24, 1972
24-sep-72
24sep72
```

The year can also be omitted. In this case, the last specified year is used, or the current year if none. For example:

```
9/24
sep 24
```

Here are the rules.

For numeric months, the ISO 8601 format '*year*-*month*-*day*' is allowed, where *year* is any positive number, *month* is a number between 01 and 12, and *day* is a number between 01 and 31. A leading zero must be present if a number is less than ten. If *year* is 68 or smaller, then 2000 is added to it; otherwise, if *year* is less than 100, then 1900 is added to it. The construct '*month*/*day*/*year*', popular in the United States, is accepted. Also '*month*/*day*', omitting the year.

Literal months may be spelled out in full: '`January`', '`February`', '`March`', '`April`', '`May`', '`June`', '`July`', '`August`', '`September`', '`October`', '`November`' or '`December`'. Literal months may be abbreviated to their first three letters, possibly followed by an abbreviating dot. It is also permitted to write '`Sept`' instead of '`September`'.

When months are written literally, the calendar date may be given as any of the following:

day month year
day month
month day year
day-*month*-*year*

Or, omitting the year:

month day

3.3 Time of day items

A *time of day item* in date strings specifies the time on a given day. Here are some examples, all of which represent the same time:

```
20:02:0
20:02
8:02pm
20:02-0500      # In EST (Eastern U.S. Standard Time).
```

More generally, the time of the day may be given as '*hour*:*minute*:*second*', where *hour* is a number between 0 and 23, *minute* is a number between 0 and 59, and *second* is a number between 0 and 59. Alternatively, ':*second*' can be omitted, in which case it is taken to be zero.

If the time is followed by 'am' or 'pm' (or 'a.m.' or 'p.m.'), *hour* is restricted to run from 1 to 12, and ':*minute*' may be omitted (taken to be zero). 'am' indicates the first half of the day, 'pm' indicates the second half of the day. In this notation, 12 is the predecessor of 1: midnight is '12am' while noon is '12pm'.

The time may alternatively be followed by a time zone correction, expressed as '*shhmm*', where *s* is '+' or '-', *hh* is a number of zone hours and *mm* is a number of zone minutes. When a time zone correction is given this way, it forces interpretation of the time relative to Coordinated Universal Time (UTC), overriding any previous specification for the time zone or the local time zone. The *minute* part of the time of the day may not be elided when a time zone correction is used. This is the only way to specify a time zone correction by fractional parts of an hour.

Either 'am'/'pm' or a time zone correction may be specified, but not both.

3.4 Time zone items

A *time zone item* specifies an international time zone, indicated by a small set of letters. They are supported for backward compatibility reasons, but they are not recommended because they are ambiguous in practice: for example, the abbreviation 'EST' has different meanings in Australia and the United States. Any included period is ignored. Military time zone designations use a single letter. Currently, only integral zone hours may be represented in a time zone item. See the previous section for a finer control over the time zone correction.

Here are many non-daylight-saving-time time zones, indexed by the zone hour value.

-1200 'Y' for militaries.

-1100 'X' for militaries.

-1000 'W' for militaries.

-0900 'V' for militaries.

-0800 'PST' for Pacific Standard, and 'U' for militaries.

-0700 'MST' for Mountain Standard, and 'T' for militaries.

-0600 'CST' for Central Standard, and 'S' for militaries.

-0500 'EST' for Eastern Standard, and 'R' for militaries.

-0400 'AST' for Atlantic Standard, and 'Q' for militaries.

-0300 'P' for militaries.

-0200 'O' for militaries.

-0100 'N' for militaries.

+0000 'GMT' for Greenwich Mean, 'UT' for Universal, 'UTC' for Coordinated Universal, 'WET' for Western European, and 'Z' for ISO 8601 and militaries.

+0100 'A' for militaries, 'CET' for Central European, 'MET' for Midden Europesche Tijd (Dutch), and 'MEZ' for Mittel-Europäische Zeit (German).

+0200 'B' for militaries, and 'EET' for Eastern European.

+0300 'C' for militaries.

+0400 'D' for militaries.

+0500 'E' for militaries.

+0600 'F' for militaries.

+0700 'G' for militaries.

+0800 'H' for militaries.

+0900 'I' for militaries, and 'JST' for Japan Standard.

+1000 'GST' for Guam Standard, and 'K' for militaries.

+1100 'L' for militaries.

+1200 'M' for militaries, and 'NZST' for New Zealand Standard.

Here are many daylight-saving time (DST) time zones, indexed by the zone hour value. Also, by following a non-DST time zone by the string 'DST' in a separate word (that is, separated by some whitespace), the corresponding DST time zone may be specified.

-0700 'PDT' for Pacific Daylight.

-0600 'MDT' for Mountain Daylight.

-0500 'CDT' for Central Daylight.

-0400 'EDT' for Eastern Daylight.

-0300 'ADT' for Atlantic Daylight.

+0100 'BST' for British Summer, and 'WEST' for Western European Summer.

+0200 'CEST' for Central European Summer, 'MEST' for Midden Europesche S. Tijd (Dutch), and 'MESZ' for Mittel-Europäische Sommerzeit (German).

+1300 'NZDT' for New Zealand Daylight.

3.5 Day of week items

The explicit mention of a day of the week will forward the date (only if necessary) to reach that day of the week in the future.

Days of the week may be spelled out in full: '`Sunday`', '`Monday`', '`Tuesday`', '`Wednesday`', '`Thursday`', '`Friday`' or '`Saturday`'. Days may be abbreviated to their first three letters, optionally followed by a period. The special abbreviations '`Tues`' for '`Tuesday`', '`Wednes`' for '`Wednesday`' and '`Thur`' or '`Thurs`' for '`Thursday`' are also allowed.

A number may precede a day of the week item to move forward supplementary weeks. It is best used in expression like '`third monday`'. In this context, '`last` *day*' or '`next` *day*' is also acceptable; they move one week before or after the day that *day* by itself would represent.

A comma following a day of the week item is ignored.

3.6 Relative items in date strings

Relative items adjust a date (or the current date if none) forward or backward. The effects of relative items accumulate. Here are some examples:

```
1 year
1 year ago
3 years
2 days
```

The unit of time displacement may be selected by the string '`year`' or '`month`' for moving by whole years or months. These are fuzzy units, as years and months are not all of equal duration. More precise units are '`fortnight`' which is worth 14 days, '`week`' worth 7 days, '`day`' worth 24 hours, '`hour`' worth 60 minutes, '`minute`' or '`min`' worth 60 seconds, and '`second`' or '`sec`' worth one second. An '`s`' suffix on these units is accepted and ignored.

The unit of time may be preceded by a multiplier, given as an optionally signed number. Unsigned numbers are taken as positively signed. No number at all implies 1 for a multiplier. Following a relative item by the string '`ago`' is equivalent to preceding the unit by a multiplicator with value -1.

The string '`tomorrow`' is worth one day in the future (equivalent to '`day`'), the string '`yesterday`' is worth one day in the past (equivalent to '`day ago`').

The strings '`now`' or '`today`' are relative items corresponding to zero-valued time displacement, these strings come from the fact a zero-valued time displacement represents the current time when not otherwise changed by previous items. They may be used to stress other items, like in '`12:00 today`'. The string '`this`' also has the meaning of a zero-valued time displacement, but is preferred in date strings like '`this thursday`'.

When a relative item causes the resulting date to cross the boundary between DST and non-DST (or vice-versa), the hour is adjusted according to the local time.

3.7 Pure numbers in date strings

The precise intepretation of a pure decimal number depends on the context in the date string.

If the decimal number is of the form *yyyymmdd* and no other calendar date item (see Section 3.2 [Calendar date items], page 4) appears before it in the date string, then *yyyy* is read as the year, *mm* as the month number and *dd* as the day of the month, for the specified calendar date.

If the decimal number is of the form *hhmm* and no other time of day item appears before it in the date string, then *hh* is read as the hour of the day and *mm* as the minute of the hour, for the specified time of the day. *mm* can also be omitted.

If both a calendar date and a time of day appear to the left of a number in the date string, but no relative item, then the number overrides the year.

3.8 Authors of `getdate`

`getdate` was originally implemented by Steven M. Bellovin (`smb@research.att.com`) while at the University of North Carolina at Chapel Hill. The code was later tweaked by a couple of people on Usenet, then completely overhauled by Rich $alz (`rsalz@bbn.com`) and Jim Berets (`jberets@bbn.com`) in August, 1990. Various revisions for the GNU system were made by David MacKenzie, Jim Meyering, Paul Eggert and others.

This chapter was originally produced by François Pinard (`pinard@iro.umontreal.ca`) from the '`getdate.y`' source code, and then edited by K. Berry (`kb@cs.umb.edu`).

4 Printing text

This section describes commands that display text strings.

4.1 `echo`: Print a line of text

`echo` writes each given *string* to standard output, with a space between each and a newline after the last one. Synopsis:

`echo [`*option*`]... [`*string*`]...`

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-n`' Do not output the trailing newline.

'`-e`' Enable interpretation of the following backslash-escaped characters in each *string*:

- '`\a`' alert (bell)
- '`\b`' backspace
- '`\c`' suppress trailing newline
- '`\f`' form feed
- '`\n`' new line
- '`\r`' carriage return
- '`\t`' horizontal tab
- '`\v`' vertical tab
- '`\\`' backslash
- '`\`*nnn*' the character whose ASCII code is *nnn* (octal); if *nnn* is not a valid octal number, it is printed literally.

4.2 `printf`: Format and print data

`printf` does formatted printing of text. Synopsis:

`printf` *format* `[`*argument*`]...`

`printf` prints the *format* string, interpreting '`%`' directives and '`\`' escapes in the same way as the C `printf` function. The *format* argument is re-used as necessary to convert all of the given *arguments*.

`printf` has one additional directive, '`%b`', which prints its argument string with '`\`' escapes interpreted in the same way as in the *format* string.

`printf` interprets '`\0ooo`' in *format* as an octal number (if *ooo* is 0 to 3 octal digits) specifying a character to print, and '`\xhhh`' as a hexadecimal number (if *hhh* is 1 to 3 hex digits) specifying a character to print.

`printf` interprets two character syntaxes introduced in ISO C 99: '`\u`' for 16-bit Unicode characters, specified as 4 hex digits *hhhh*, and '`\U`' for 32-bit Unicode characters, specified as

8 hex digits *hhhhhhhh*. `printf` outputs the Unicode characters according to the LC_CTYPE part of the current locale, i.e. depending on the values of the environment variables `LC_ALL`, `LC_CTYPE`, `LANG`.

The processing of '`\u`' and '`\U`' requires a full-featured `iconv` facility. It is activated on systems with glibc 2.2 (or newer), or when `libiconv` is installed prior to the sh-utils. Otherwise the use of '`\u`' and '`\U`' will give an error message.

An additional escape, '`\c`', causes `printf` to produce no further output.

The only options are a lone '`--help`' or '`--version`'. See Chapter 2 [Common options], page 2.

The Unicode character syntaxes are useful for writing strings in a locale independent way. For example, a string containing the Euro currency symbol

```
$ /usr/local/bin/printf '\u20AC 14.95'
```

will be output correctly in all locales supporting the Euro symbol (ISO-8859-15, UTF-8, and others). Similarly, a Chinese string

```
$ /usr/local/bin/printf '\u4e2d\u6587'
```

will be output correctly in all chinese locales (GB2312, BIG5, UTF-8, etc).

Note that in these examples, the full pathname of `printf` has been given, to distinguish it from the GNU `bash` builtin function `printf`.

For larger strings, you don't need to look up the hexadecimal code values of each character one by one. ASCII characters mixed with \u escape sequences is also known as the JAVA source file encoding. You can use GNU recode 3.5c (or newer) to convert strings to this encoding. Here is how to convert a piece of text into a shell script which will output this text in a locale- independent way:

```
$ LC_CTYPE=zh_CN.big5 /usr/local/bin/printf \
    '\u4e2d\u6587\n' > sample.txt
$ recode BIG5..JAVA < sample.txt \
    | sed -e "s|^|/usr/local/bin/printf '|" -e "s|$|\\\\n'|" \
    > sample.sh
```

4.3 yes: Print a string until interrupted

`yes` prints the command line arguments, separated by spaces and followed by a newline, forever until it is killed. If no arguments are given, it prints '`y`' followed by a newline forever until killed.

The only options are a lone '`--help`' or '`--version`'. See Chapter 2 [Common options], page 2.

5 Conditions

This section describes commands that are primarily useful for their exit status, rather than their output. Thus, they are often used as the condition of shell `if` statements, or as the last command in a pipeline.

5.1 `false`: Do nothing, unsuccessfully

`false` does nothing except return an exit status of 1, meaning *failure*. It can be used as a place holder in shell scripts where an unsuccessful command is needed.

`false` ignores *all* command line arguments, even '`--help`' and '`--version`', since to do otherwise would change expected behavior that some programmers may be relying on.

This version of `false` is implemented as a C program, and is thus more secure and faster than a shell script implementation, and may safely be used as a dummy shell for the purpose of disabling accounts.

5.2 `true`: Do nothing, successfully

`true` does nothing except return an exit status of 0, meaning *success*. It can be used as a place holder in shell scripts where a successful command is needed, although the shell built-in command `:` (colon) does the same thing faster.

`true` ignores *all* command line arguments, even '`--help`' and '`--version`', since to do otherwise would change expected behavior that some programmers may be relying on.

This version of `true` is implemented as a C program, and is thus more secure and faster than a shell script implementation, and may safely be used as a dummy shell for the purpose of disabling accounts.

5.3 `test`: Check file types and compare values

`test` returns a status of 0 (true) or 1 (false) depending on the evaluation of the conditional expression *expr*. Each part of the expression must be a separate argument.

`test` has file status checks, string operators, and numeric comparison operators.

Because most shells have a built-in command by the same name, using the unadorned command name in a script or interactively may get you different functionality than that described here.

Besides the options below, `test` accepts a lone '`--help`' or '`--version`'. See Chapter 2 [Common options], page 2. A single non-option argument is also allowed: `test` returns true if the argument is not null.

5.3.1 File type tests

These options test for particular types of files. (Everything's a file, but not all files are the same!)

'`-b` *file*' True if *file* exists and is a block special device.

‘`-c` *file*’ True if *file* exists and is a character special device.

‘`-d` *file*’ True if *file* exists and is a directory.

‘`-f` *file*’ True if *file* exists and is a regular file.

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

‘`-p` *file*’ True if *file* exists and is a named pipe.

‘`-S` *file*’ True if *file* exists and is a socket.

‘`-t [`*fd*`]`’ True if *fd* is opened on a terminal. If *fd* is omitted, it defaults to 1 (standard output).

5.3.2 Access permission tests

These options test for particular access permissions.

‘`-g` *file*’ True if *file* exists and has its set-group-id bit set.

‘`-k` *file*’ True if *file* has its *sticky* bit set.

‘`-r` *file*’ True if *file* exists and is readable.

‘`-u` *file*’ True if *file* exists and has its set-user-id bit set.

‘`-w` *file*’ True if *file* exists and is writable.

‘`-x` *file*’ True if *file* exists and is executable.

‘`-O` *file*’ True if *file* exists and is owned by the current effective user id.

‘`-G` *file*’ True if *file* exists and is owned by the current effective group id.

5.3.3 File characteristics tests

These options test other file characteristics.

‘`-e` *file*’ True if *file* exists.

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

‘*file1* `-nt` *file2*’
True if *file1* is newer (according to modification date) than *file2*.

‘*file1* `-ot` *file2*’
True if *file1* is older (according to modification date) than *file2*.

‘*file1* `-ef` *file2*’
True if *file1* and *file2* have the same device and inode numbers, i.e., if they are hard links to each other.

5.3.4 String tests

These options test string characteristics. Strings are not quoted for `test`, though you may need to quote them to protect characters with special meaning to the shell, e.g., spaces.

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

'`-n` *string*'
'*string*' True if the length of *string* is nonzero.

'*string1* `=` *string2*'
True if the strings are equal.

'*string1* `!=` *string2*'
True if the strings are not equal.

5.3.5 Numeric tests

Numeric relationals. The arguments must be entirely numeric (possibly negative), or the special expression `-l` *string*, which evaluates to the length of *string*.

'*arg1* `-eq` *arg2*'
'*arg1* `-ne` *arg2*'
'*arg1* `-lt` *arg2*'
'*arg1* `-le` *arg2*'
'*arg1* `-gt` *arg2*'
'*arg1* `-ge` *arg2*'
These arithmetic binary operators return true if *arg1* is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than *arg2*, respectively.

For example:

```
test -1 -gt -2 && echo yes
⇒ yes
test -l abc -gt 1 && echo yes
⇒ yes
test 0x100 -eq 1
error  test: integer expression expected before -eq
```

5.3.6 Connectives for `test`

The usual logical connectives.

'`!` *expr*' True if *expr* is false.

'*expr1* `-a` *expr2*'
True if both *expr1* and *expr2* are true.

'*expr1* `-o` *expr2*'
True if either *expr1* or *expr2* is true.

5.4 `expr`: Evaluate expressions

`expr` evaluates an expression and writes the result on standard output. Each token of the expression must be a separate argument.

Operands are either numbers or strings. `expr` converts anything appearing in an operand position to an integer or a string depending on the operation being applied to it.

Strings are not quoted for `expr` itself, though you may need to quote them to protect characters with special meaning to the shell, e.g., spaces.

Operators may given as infix symbols or prefix keywords. Parentheses may be used for grouping in the usual manner (you must quote parentheses to avoid the shell evaluating them, however).

Exit status:

0 if the expression is neither null nor 0,
1 if the expression is null or 0,
2 for invalid expressions.

5.4.1 String expressions

`expr` supports pattern matching and other string operators. These have lower precedence than both the numeric and relational operators (in the next sections).

'*string* `:` *regex*'
: Perform pattern matching. The arguments are converted to strings and the second is considered to be a (basic, a la GNU `grep`) regular expression, with a `^` implicitly prepended. The first argument is then matched against this regular expression.

  If the match succeeds and *regex* uses '`\(`' and '`\)`', the `:` expression returns the part of *string* that matched the subexpression; otherwise, it returns the number of characters matched.

  If the match fails, the `:` operator returns the null string if '`\(`' and '`\)`' are used in *regex*, otherwise 0.

  Only the first '`\( ... \)`' pair is relevant to the return value; additional pairs are meaningful only for grouping the regular expression operators.

  In the regular expression, `\+`, `\?`, and `\|` are operators which respectively match one or more, zero or one, or separate alternatives. SunOS and other `expr`'s treat these as regular characters. (POSIX allows either behavior.) See section "Regular Expression Library" in *Regex*, for details of regular expression syntax. Some examples are in Section 5.4.4 [Examples of expr], page 15.

'`match` *string regex*'
: An alternative way to do pattern matching. This is the same as '*string* `:` *regex*'.

'`substr` *string position length*'
: Returns the substring of *string* beginning at *position* with length at most *length*. If either *position* or *length* is negative, zero, or non-numeric, returns the null string.

'`index` *string charset*'
: Returns the first position in *string* where the first character in *charset* was found. If no character in *charset* is found in *string*, return 0.

'`length` *string*'
: Returns the length of *string*.

'`quote` *token*'
: Interpret *token* as a string, even if it is a keyword like *match* or an operator like `/`. This makes it possible to test `expr length quote "$x"` or `expr quote "$x" : '.*/\(.\)'` and have it do the right thing even if the value of *$x* happens to be (for example) `/` or `index`. This operator is a GNU extension. It is disabled when the environment variable `POSIXLY_CORRECT` is set.

To make `expr` interpret keywords as strings, you must use the `quote` operator.

5.4.2 Numeric expressions

`expr` supports the usual numeric operators, in order of increasing precedence. The string operators (previous section) have lower precedence, the connectives (next section) have higher.

'`+ -`'
: Addition and subtraction. Both arguments are converted to numbers; an error occurs if this cannot be done.

'`* / %`'
: Multiplication, division, remainder. Both arguments are converted to numbers; an error occurs if this cannot be done.

5.4.3 Relations for `expr`

`expr` supports the usual logical connectives and relations. These are higher precedence than either the string or numeric operators (previous sections). Here is the list, lowest-precedence operator first.

'`|`'
: Returns its first argument if that is neither null nor 0, otherwise its second argument.

'`&`'
: Return its first argument if neither argument is null or 0, otherwise 0.

'`< <= = == != >= >`'
: Compare the arguments and return 1 if the relation is true, 0 otherwise. `==` is a synonym for `=`. `expr` first tries to convert both arguments to numbers and do a numeric comparison; if either conversion fails, it does a lexicographic comparison.

5.4.4 Examples of using `expr`

Here are a few examples, including quoting for shell metacharacters.

To add 1 to the shell variable `foo`, in Bourne-compatible shells:

```
foo=`expr $foo + 1`
```

To print the non-directory part of the file name stored in `$fname`, which need not contain a `/`.

```
expr $fname : '.*/\(^.*\)' '^|' $fname
```

An example showing that `\+` is an operator:

```
expr aaa : 'a\+'
⇒ 3
expr abc : 'a\(.\)c'
⇒ b
expr index abcdef cz
⇒ 3
expr index index a
error expr: syntax error
expr index quote index a
⇒ 0
```

6 Redirection

Unix shells commonly provide several forms of *redirection*—ways to change the input source or output destination of a command. But one useful redirection is performed by a separate command, not by the shell; it's described here.

6.1 `tee`: Redirect output to multiple files

The `tee` command copies standard input to standard output and also to any files given as arguments. This is useful when you want not only to send some data down a pipe, but also to save a copy. Synopsis:

`tee [`*option*`]... [`*file*`]...`

If a file being written to does not already exist, it is created. If a file being written to already exists, the data it previously contained is overwritten unless the `-a` option is used.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-a`'
'`--append`'
Append standard input to the given files rather than overwriting them.

'`-i`'
'`--ignore-interrupts`'
Ignore interrupt signals.

7 File name manipulation

This section describes commands that manipulate file names.

7.1 basename: Strip directory and suffix from a file name

`basename` removes any leading directory components from *name*. Synopsis:

`basename` *name* `[`*suffix*`]`

If *suffix* is specified and is identical to the end of *name*, it is removed from *name* as well. `basename` prints the result on standard output.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

7.2 dirname: Strip non-directory suffix from a file name

`dirname` prints all but the final slash-delimited component of a string (presumably a filename). Synopsis:

`dirname` *name*

If *name* is a single component, `dirname` prints '`.`' (meaning the current directory).

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

7.3 pathchk: Check file name portability

`pathchk` checks portability of filenames. Synopsis:

`pathchk [`*option*`]...` *name*...

For each *name*, `pathchk` prints a message if any of these conditions is true:

1. one of the existing directories in *name* does not have search (execute) permission,
2. the length of *name* is larger than its filesystem's maximum file name length,
3. the length of one component of *name*, corresponding to an existing directory name, is larger than its filesystem's maximum length for a file name component.

The program accepts the following option. Also see Chapter 2 [Common options], page 2.

'`-p`'
'`--portability`'
: Instead of performing length checks on the underlying filesystem, test the length of each file name and its components against the POSIX.1 minimum limits for portability. Also check that the file name contains no characters not in the portable file name character set.

Exit status:

0 if all specified file names passed all of the tests,
1 otherwise.

8 Working context

This section describes commands that display or alter the context in which you are working: the current directory, the terminal settings, and so forth. See also the user-related commands in the next section.

8.1 pwd: Print working directory

`pwd` prints the fully resolved name of the current directory. That is, all components of the printed name will be actual directory names—none will be symbolic links.

Because most shells have a built-in command by the same name, using the unadorned command name in a script or interactively may get you different functionality than that described here.

The only options are a lone '`--help`' or '`--version`'. See Chapter 2 [Common options], page 2.

8.2 stty: Print or change terminal characteristics

`stty` prints or changes terminal characteristics, such as baud rate. Synopses:

```
stty [option] [setting]...
stty [option]
```

If given no line settings, `stty` prints the baud rate, line discipline number (on systems that support it), and line settings that have been changed from the values set by '`stty sane`'. By default, mode reading and setting are performed on the tty line connected to standard input, although this can be modified by the '`--file`' option.

`stty` accepts many non-option arguments that change aspects of the terminal line operation, as described below.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-a`'
'`--all`' Print all current settings in human-readable form. This option may not be used in combination with any line settings.

'`-F` *device*'
'`--file=`*device*'
Set the line opened by the filename specified in *device* instead of the tty line connected to standard input. This option is necessary because opening a POSIX tty requires use of the `O_NONDELAY` flag to prevent a POSIX tty from blocking until the carrier detect line is high if the `clocal` flag is not set. Hence, it is not always possible to allow the shell to open the device in the traditional manner.

'`-g`'
'`--save`' Print all current settings in a form that can be used as an argument to another `stty` command to restore the current settings. This option may not be used in combination with any line settings.

Many settings can be turned off by preceding them with a '`-`'. Such arguments are marked below with "May be negated" in their description. The descriptions themselves refer to the positive case, that is, when *not* negated (unless stated otherwise, of course).

Some settings are not available on all POSIX systems, since they use extensions. Such arguments are marked below with "Non-POSIX" in their description. On non-POSIX systems, those or other settings also may not be available, but it's not feasible to document all the variations: just try it and see.

8.2.1 Control settings

Control settings:

'`parenb`' Generate parity bit in output and expect parity bit in input. May be negated.

'`parodd`' Set odd parity (even if negated). May be negated.

'`cs5`'
'`cs6`'
'`cs7`'
'`cs8`' Set character size to 5, 6, 7, or 8 bits.

'`hup`'
'`hupcl`' Send a hangup signal when the last process closes the tty. May be negated.

'`cstopb`' Use two stop bits per character (one if negated). May be negated.

'`cread`' Allow input to be received. May be negated.

'`clocal`' Disable modem control signals. May be negated.

'`crtscts`' Enable RTS/CTS flow control. Non-POSIX. May be negated.

8.2.2 Input settings

'`ignbrk`' Ignore break characters. May be negated.

'`brkint`' Make breaks cause an interrupt signal. May be negated.

'`ignpar`' Ignore characters with parity errors. May be negated.

'`parmrk`' Mark parity errors (with a 255-0-character sequence). May be negated.

'`inpck`' Enable input parity checking. May be negated.

'`istrip`' Clear high (8th) bit of input characters. May be negated.

'`inlcr`' Translate newline to carriage return. May be negated.

'`igncr`' Ignore carriage return. May be negated.

'`icrnl`' Translate carriage return to newline. May be negated.

'`ixon`' Enable XON/XOFF flow control (that is, *CTRL-S*/*CTRL-Q*). May be negated.

'`ixoff`'
'`tandem`' Enable sending of `stop` character when the system input buffer is almost full, and `start` character when it becomes almost empty again. May be negated.

'iuclc' Translate uppercase characters to lowercase. Non-POSIX. May be negated.

'ixany' Allow any character to restart output (only the start character if negated). Non-POSIX. May be negated.

'imaxbel' Enable beeping and not flushing input buffer if a character arrives when the input buffer is full. Non-POSIX. May be negated.

8.2.3 Output settings

These arguments specify output-related operations.

'opost' Postprocess output. May be negated.

'olcuc' Translate lowercase characters to uppercase. Non-POSIX. May be negated.

'ocrnl' Translate carriage return to newline. Non-POSIX. May be negated.

'onlcr' Translate newline to carriage return-newline. Non-POSIX. May be negated.

'onocr' Do not print carriage returns in the first column. Non-POSIX. May be negated.

'onlret' Newline performs a carriage return. Non-POSIX. May be negated.

'ofill' Use fill (padding) characters instead of timing for delays. Non-POSIX. May be negated.

'ofdel' Use delete characters for fill instead of null characters. Non-POSIX. May be negated.

'nl1'
'nl0' Newline delay style. Non-POSIX.

'cr3'
'cr2'
'cr1'
'cr0' Carriage return delay style. Non-POSIX.

'tab3'
'tab2'
'tab1'
'tab0' Horizontal tab delay style. Non-POSIX.

'bs1'
'bs0' Backspace delay style. Non-POSIX.

'vt1'
'vt0' Vertical tab delay style. Non-POSIX.

'ff1'
'ff0' Form feed delay style. Non-POSIX.

8.2.4 Local settings

'isig' Enable interrupt, quit, and suspend special characters. May be negated.

'icanon' Enable erase, kill, werase, and rprnt special characters. May be negated.

'iexten' Enable non-POSIX special characters. May be negated.

'echo' Echo input characters. May be negated.

'echoe'
'crterase'
Echo erase characters as backspace-space-backspace. May be negated.

'echok' Echo a newline after a kill character. May be negated.

'echonl' Echo newline even if not echoing other characters. May be negated.

'noflsh' Disable flushing after interrupt and quit special characters. May be negated.

'xcase' Enable input and output of uppercase characters by preceding their lowercase equivalents with '\', when icanon is set. Non-POSIX. May be negated.

'tostop' Stop background jobs that try to write to the terminal. Non-POSIX. May be negated.

'echoprt'
'prterase'
Echo erased characters backward, between '\' and '/'. Non-POSIX. May be negated.

'echoctl'
'ctlecho' Echo control characters in hat notation ('^*c*') instead of literally. Non-POSIX. May be negated.

'echoke'
'crtkill' Echo the kill special character by erasing each character on the line as indicated by the echoprt and echoe settings, instead of by the echoctl and echok settings. Non-POSIX. May be negated.

8.2.5 Combination settings

Combination settings:

'evenp'
'parity' Same as parenb -parodd cs7. May be negated. If negated, same as -parenb cs8.

'oddp' Same as parenb parodd cs7. May be negated. If negated, same as -parenb cs8.

'nl' Same as -icrnl -onlcr. May be negated. If negated, same as icrnl -inlcr -igncr onlcr -ocrnl -onlret.

'ek' Reset the erase and kill special characters to their default values.

'sane' Same as:

```
cread -ignbrk brkint -inlcr -igncr icrnl -ixoff
-iuclc -ixany imaxbel opost -olcuc -ocrnl onlcr
-onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0
ff0 isig icanon iexten echo echoe echok -echonl
-noflsh -xcase -tostop -echoprt echoctl echoke
```

and also sets all special characters to their default values.

'cooked' Same as brkint ignpar istrip icrnl ixon opost isig icanon, plus sets the eof and eol characters to their default values if they are the same as the min and time characters. May be negated. If negated, same as raw.

'raw' Same as:

```
-ignbrk -brkint -ignpar -parmrk -inpck -istrip
-inlcr -igncr -icrnl -ixon -ixoff -iuclc -ixany
-imaxbel -opost -isig -icanon -xcase min 1 time 0
```

May be negated. If negated, same as cooked.

'cbreak' Same as -icanon. May be negated. If negated, same as icanon.

'pass8' Same as -parenb -istrip cs8. May be negated. If negated, same as parenb istrip cs7.

'litout' Same as -parenb -istrip -opost cs8. May be negated. If negated, same as parenb istrip opost cs7.

'decctlq' Same as -ixany. Non-POSIX. May be negated.

'tabs' Same as tab0. Non-POSIX. May be negated. If negated, same as tab3.

'lcase'
'LCASE' Same as xcase iuclc olcuc. Non-POSIX. May be negated.

'crt' Same as echoe echoctl echoke.

'dec' Same as echoe echoctl echoke -ixany intr ^C erase ^? kill C-u.

8.2.6 Special characters

The special characters' default values vary from system to system. They are set with the syntax 'name value', where the names are listed below and the value can be given either literally, in hat notation ('^*c*'), or as an integer which may start with '0x' to indicate hexadecimal, '0' to indicate octal, or any other digit to indicate decimal.

For GNU stty, giving a value of ^- or undef disables that special character. (This is incompatible with Ultrix stty, which uses a value of 'u' to disable a special character. GNU stty treats a value 'u' like any other, namely to set that special character to ⟨U⟩.)

'intr' Send an interrupt signal.

'quit' Send a quit signal.

'erase' Erase the last character typed.

'kill' Erase the current line.

'eof' Send an end of file (terminate the input).

'eol' End the line.

'eol2' Alternate character to end the line. Non-POSIX.

'swtch' Switch to a different shell layer. Non-POSIX.

'start' Restart the output after stopping it.

'stop' Stop the output.

'susp' Send a terminal stop signal.

'dsusp' Send a terminal stop signal after flushing the input. Non-POSIX.

'rprnt' Redraw the current line. Non-POSIX.

'werase' Erase the last word typed. Non-POSIX.

'lnext' Enter the next character typed literally, even if it is a special character. Non-POSIX.

8.2.7 Special settings

'min *n*' Set the minimum number of characters that will satisfy a read until the time value has expired, when `-icanon` is set.

'time *n*' Set the number of tenths of a second before reads time out if the minimum number of characters have not been read, when `-icanon` is set.

'ispeed *n*' Set the input speed to *n*.

'ospeed *n*' Set the output speed to *n*.

'rows *n*' Tell the tty kernel driver that the terminal has *n* rows. Non-POSIX.

'cols *n*'
'columns *n*'
Tell the kernel that the terminal has *n* columns. Non-POSIX.

'size' Print the number of rows and columns that the kernel thinks the terminal has. (Systems that don't support rows and columns in the kernel typically use the environment variables `LINES` and `COLUMNS` instead; however, GNU `stty` does not know anything about them.) Non-POSIX.

'line *n*' Use line discipline *n*. Non-POSIX.

'speed' Print the terminal speed.

'*n*' Set the input and output speeds to *n*. *n* can be one of: 0 50 75 110 134 134.5 150 200 300 600 1200 1800 2400 4800 9600 19200 38400 `exta` `extb`. `exta` is the same as 19200; `extb` is the same as 38400. 0 hangs up the line if `-clocal` is set.

8.3 `printenv`: Print all or some environment variables

`printenv` prints environment variable values. Synopsis:

`printenv [option] [variable]...`

If no *variable*s are specified, `printenv` prints the value of every environment variable. Otherwise, it prints the value of each *variable* that is set, and nothing for those that are not set.

The only options are a lone '`--help`' or '`--version`'. See Chapter 2 [Common options], page 2.

Exit status:

0 if all variables specified were found
1 if at least one specified variable was not found
2 if a write error occurred

8.4 `tty`: Print file name of terminal on standard input

`tty` prints the file name of the terminal connected to its standard input. It prints '`not a tty`' if standard input is not a terminal. Synopsis:

`tty [option]...`

The program accepts the following option. Also see Chapter 2 [Common options], page 2.

'`-s`'
'`--silent`'
'`--quiet`' Print nothing; only return an exit status.

Exit status:

0 if standard input is a terminal
1 if standard input is not a terminal
2 if given incorrect arguments
3 if a write error occurs

9 User information

This section describes commands that print user-related information: logins, groups, and so forth.

9.1 `id`: Print real and effective uid and gid

`id` prints information about the given user, or the process running it if no user is specified. Synopsis:

`id [`*option*`]... [`*username*`]`

By default, it prints the real user id, real group id, effective user id if different from the real user id, effective group id if different from the real group id, and supplemental group ids.

Each of these numeric values is preceded by an identifying string and followed by the corresponding user or group name in parentheses.

The options cause `id` to print only part of the above information. Also see Chapter 2 [Common options], page 2.

'`-g`'
'`--group`' Print only the group id.

'`-G`'
'`--groups`'
Print only the supplementary groups.

'`-n`'
'`--name`' Print the user or group name instead of the ID number. Requires `-u`, `-g`, or `-G`.

'`-r`'
'`--real`' Print the real, instead of effective, user or group id. Requires `-u`, `-g`, or `-G`.

'`-u`'
'`--user`' Print only the user id.

9.2 `logname`: Print current login name

`logname` prints the calling user's name, as found in the file '`/etc/utmp`', and exits with a status of 0. If there is no '`/etc/utmp`' entry for the calling process, `logname` prints an error message and exits with a status of 1.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

9.3 `whoami`: Print effective user id

`whoami` prints the user name associated with the current effective user id. It is equivalent to the command '`id -un`'.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

9.4 groups: Print group names a user is in

`groups` prints the names of the primary and any supplementary groups for each given *username*, or the current process if no names are given. If names are given, the name of each user is printed before the list of that user's groups. Synopsis:

`groups [username]...`

The group lists are equivalent to the output of the command '`id -Gn`'.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

9.5 users: Print login names of users currently logged in

`users` prints on a single line a blank-separated list of user names of users currently logged in to the current host. Each user name corresponds to a login session, so if a user has more than one login session, that user's name will appear the same number of times in the output. Synopsis:

`users [file]`

With no *file* argument, `users` extracts its information from the file '`/etc/utmp`'. If a file argument is given, `users` uses that file instead. A common choice is '`/etc/wtmp`'.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

9.6 who: Print who is currently logged in

`who` prints information about users who are currently logged on. Synopsis:

`who [option] [file] [am i]`

If given no non-option arguments, `who` prints the following information for each user currently logged on: login name, terminal line, login time, and remote hostname or X display.

If given one non-option argument, `who` uses that instead of '`/etc/utmp`' as the name of the file containing the record of users logged on. '`/etc/wtmp`' is commonly given as an argument to `who` to look at who has previously logged on.

If given two non-option arguments, `who` prints only the entry for the user running it (determined from its standard input), preceded by the hostname. Traditionally, the two arguments given are '`am i`', as in '`who am i`'.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-m`' Same as '`who am i`'.

'`-q`'
'`--count`' Print only the login names and the number of users logged on. Overrides all other options.

'`-s`' Ignored; for compatibility with other versions of `who`.

'-i'
'-u'
'--idle' After the login time, print the number of hours and minutes that the user has been idle. '.' means the user was active in last minute. 'old' means the user was idle for more than 24 hours.

'-l'
'--lookup'
Attempt to canonicalize hostnames found in utmp through a DNS lookup. This is not the default because it can cause significant delays on systems with automatic dial-up internet access.

'-H'
'--heading'
Print a line of column headings.

'-w'
'-T'
'--mesg'
'--message'
'--writable'
After each login name print a character indicating the user's message status:

'+' allowing `write` messages
'-' disallowing `write` messages
'?' cannot find terminal device

10 System context

This section describes commands that print or change system-wide information.

10.1 `date`: Print or set system date and time

`date` with no arguments prints the current time and date, in the format of the '`%c`' directive (described below). Synopses:

```
date [option]... [+format]
date [-u|--utc|--universal] [ MMDDhhmm[[CC]YY][.ss] ]
```

If given an argument that starts with a '`+`', `date` prints the current time and date (or the time and date specified by the `--date` option, see below) in the format defined by that argument, which is the same as in the `strftime` function. Except for directives, which start with '`%`', characters in the format string are printed unchanged. The directives are described below.

10.1.1 Time directives

`date` directives related to times.

'`%H`' hour (00...23)

'`%I`' hour (01...12)

'`%k`' hour (0...23)

'`%l`' hour (1...12)

'`%M`' minute (00...59)

'`%p`' locale's AM or PM

'`%r`' time, 12-hour (hh:mm:ss [AP]M)

'`%s`' seconds since the epoch, i.e., 1 January 1970 00:00:00 UTC (a GNU extension). Note that this value is the number of seconds between the epoch and the current date as defined by the localtime system call. It isn't changed by the '`--date`' option.

'`%S`' second (00...60)

'`%T`' time, 24-hour (hh:mm:ss)

'`%X`' locale's time representation (%H:%M:%S)

'`%z`' RFC-822 style numeric time zone (e.g., -0600 or +0100), or nothing if no time zone is determinable. This value reflects the *current* time zone. It isn't changed by the '`--date`' option.

'`%Z`' time zone (e.g., EDT), or nothing if no timezone is determinable. Note that this value reflects the *current* time zone. It isn't changed by the '`--date`' option.

10.1.2 Date directives

`date` directives related to dates.

‘`%a`’ locale’s abbreviated weekday name (Sun. . . Sat)

‘`%A`’ locale’s full weekday name, variable length (Sunday. . . Saturday)

‘`%b`’ locale’s abbreviated month name (Jan. . . Dec)

‘`%B`’ locale’s full month name, variable length (January. . . December)

‘`%c`’ locale’s date and time (Sat Nov 04 12:02:33 EST 1989)

‘`%d`’ day of month (01. . . 31)

‘`%D`’ date (mm/dd/yy)

‘`%h`’ same as %b

‘`%j`’ day of year (001. . . 366)

‘`%m`’ month (01. . . 12)

‘`%U`’ week number of year with Sunday as first day of week (00. . . 53). Days in a new year preceding the first Sunday are in week zero.

‘`%V`’ week number of year with Monday as first day of the week as a decimal (01. . . 53). If the week containing January 1 has four or more days in the new year, then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1. (See the ISO 8601: 1988 standard.)

‘`%w`’ day of week (0. . . 6) with 0 corresponding to Sunday

‘`%W`’ week number of year with Monday as first day of week (00. . . 53). Days in a new year preceding the first Monday are in week zero.

‘`%x`’ locale’s date representation (mm/dd/yy)

‘`%y`’ last two digits of year (00. . . 99)

‘`%Y`’ year (1970. . . .)

10.1.3 Literal directives

`date` directives that produce literal strings.

‘`%%`’ a literal %

‘`%n`’ a newline

‘`%t`’ a horizontal tab

10.1.4 Padding

By default, `date` pads numeric fields with zeroes, so that, for example, numeric months are always output as two digits. GNU `date` recognizes the following numeric modifiers between the ‘`%`’ and the directive.

'-' (hyphen) do not pad the field; useful if the output is intended for human consumption.

'_' (underscore) pad the field with spaces; useful if you need a fixed number of characters in the output, but zeroes are too distracting.

These are GNU extensions.

Here is an example illustrating the differences:

```
date +%d/%m -d "Feb 1"
⇒ 01/02
date +%-d/%-m -d "Feb 1"
⇒ 1/2
date +%_d/%_m -d "Feb 1"
⇒  1/ 2
```

10.1.5 Setting the time

If given an argument that does not start with '+', `date` sets the system clock to the time and date specified by that argument (as described below). You must have appropriate privileges to set the system clock. The '--date' and '--set' options may not be used with such an argument. The '--universal' option may be used with such an argument to indicate that the specified time and date are relative to Coordinated Universal Time rather than to the local time zone.

The argument must consist entirely of digits, which have the following meaning:

'MM' month

'DD' day within month

'hh' hour

'mm' minute

'CC' first two digits of year (optional)

'YY' last two digits of year (optional)

'ss' second (optional)

The '--set' option also sets the system clock; see the next section.

10.1.6 Options for date

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'-d *datestr*'
'--date=*datestr*'
Display the time and date specified in *datestr* instead of the current time and date. *datestr* can be in almost any common format. It can contain month names, timezones, 'am' and 'pm', 'yesterday', 'ago', 'next', etc. See Chapter 3 [Date input formats], page 3.

'-f *datefile*'
'--file=*datefile*'
Parse each line in *datefile* as with '-d' and display the resulting time and date. If *datefile* is '-', use standard input. This is useful when you have many dates to process, because the system overhead of starting up the date executable many times can be considerable.

'-I[*timespec*]'
'--iso-8601[=*timespec*]'
Display the date using the ISO 8601 format, '%Y-%m-%d'.

The optional argument *timespec* specifies the number of additional terms of the time to include. It can be one of the following:

'auto' The default behavior: print just the date.

'hours' Append the hour of the day to the date.

'minutes' Append the hours and minutes.

'seconds' Append the hours, minutes, and seconds.

If showing any time terms, then include the time zone using the format '%z'.

'-R'
'--rfc-822'
Display the time and date using the RFC-822-conforming format, '%a, %_d %b %Y %H:%M:%S %z'.

'-r *file*'
'--reference=*file*'
Display the time and date reference according to the last modification time of *file*, instead of the current time and date.

'-s *datestr*'
'--set=*datestr*'
Set the time and date to *datestr*, See '-d' above.

'-u'
'--utc'
'--universal'
Use Coordinated Universal Time (UTC) by operating as if the TZ environment variable was set to the string 'UTC0'. Normally, date operates in the time zone indicated by TZ, or the system default if TZ is not set. Coordinated Universal Time is often called "Greenwich Mean Time" (GMT) for historical reasons.

10.1.7 Examples of date

Here are a few examples. Also see the documentation for the '-d' option in the previous section.

- To print the date of the day before yesterday:
```
date --date='2 days ago'
```
- To print the date of the day three months and one day hence:

```
date --date='3 months 1 day'
```

- To print the day of year of Christmas in the current year:

  ```
  date --date='25 Dec' +%j
  ```

- To print the current full month name and the day of the month:

  ```
  date '+%B %d'
  ```

  But this may not be what you want because for the first nine days of the month, the '%d' expands to a zero-padded two-digit field, for example 'date -d 1may '+%B %d'' will print 'May 01'.

- To print a date without the leading zero for one-digit days of the month, you can use the (GNU extension) - modifier to suppress the padding altogether.

  ```
  date -d=1may '+%B %-d'
  ```

- To print the current date and time in the format required by many non-GNU versions of date when setting the system clock:

  ```
  date +%m%d%H%M%Y.%S
  ```

- To set the system clock forward by two minutes:

  ```
  date --set='+2 minutes'
  ```

- To print the date in the format specified by RFC-822, use 'date --rfc'. I just did and saw this:

  ```
  Mon, 25 Mar 1996 23:34:17 -0600
  ```

- To convert a date string to the number of seconds since the epoch (which is 1970-01-01 00:00:00 UTC), use the '--date' option with the '%s' format. That can be useful in sorting and/or graphing and/or comparing data by date. The following command outputs the number of the seconds since the epoch for the time one second later than the epoch, but in a time zone five hours later (Cambridge, Massachusetts), thus a total of five hours and one second after the epoch:

  ```
  date --date='1970-01-01 00:00:01 UTC +5 hours' +%s
  18001
  ```

  Suppose you had *not* specified time zone information in the example above. Then, date would have used your computer's idea of the time zone when interpreting the string. Here's what you would get if you were in Greenwich, England:

  ```
  # local time zone used
  date --date='1970-01-01 00:00:01' +%s
  1
  ```

- If you're sorting or graphing dated data, your raw date values may be represented as seconds since the epoch. But few people can look at the date '946684800' and casually note "Oh, that's the first second of the year 2000."

  ```
  date --date='2000-01-01 UTC' +%s
  946684800
  ```

  To convert such an unwieldy number of seconds back to a more readable form, use a command like this:

  ```
  date -d '1970-01-01 946684800 sec' +"%Y-%m-%d %T %z"
  2000-01-01 00:00:00 +0000
  ```

10.2 `uname`: Print system information

`uname` prints information about the machine and operating system it is run on. If no options are given, `uname` acts as if the `-s` option were given. Synopsis:

`uname [`*option*`]...`

If multiple options or `-a` are given, the selected information is printed in this order:

sysname nodename release osversion machine

The *osversion*, at least, may well be multiple words. For example:

```
uname -a
⇒ Linux hayley 1.0.4 #3 Thu May 12 18:06:34 1994 i486
```

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-a`'
'`--all`' Print all of the below information.

'`-m`'
'`--machine`'
Print the machine (hardware) type.

'`-n`'
'`--nodename`'
Print the machine's network node hostname.

'`-p`'
'`--processor`'
Print the machine's processor type

'`-r`'
'`--release`'
Print the operating system release.

'`-s`'
'`--sysname`'
Print the operating system name.

'`-v`' Print the operating system version.

10.3 `hostname`: Print or set system name

With no arguments, `hostname` prints the name of the current host system. With one argument, it sets the current host name to the specified string. You must have appropriate privileges to set the host name. Synopsis:

`hostname [`*name*`]`

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

10.4 `hostid`: Print numeric host identifier.

`hostid` prints the numeric identifier of the current host in hexadecimal. This command accepts no arguments. The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

For example, here's what it prints on one system I use:

```
$ hostid
1bac013d
```

On that system, the 32-bit quantity happens to be closely related to the system's Internet address, but that isn't always the case.

11 Modified command invocation

This section describes commands that run other commands in some context different than the current one: a modified environment, as a different user, etc.

11.1 `chroot`: Run a command with a different root directory

`chroot` runs a command with a specified root directory. On many systems, only the super-user can do this. Synopses:

```
chroot newroot [command [args]...]
chroot option
```

Ordinarily, filenames are looked up starting at the root of the directory structure, i.e., '`/`'. `chroot` changes the root to the directory *newroot* (which must exist) and then runs *command* with optional *args*. If *command* is not specified, the default is the value of the `SHELL` environment variable or `/bin/sh` if not set, invoked with the '`-i`' option.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

Here are a few tips to help avoid common problems in using chroot. To start with a simple example, make *command* refer to a statically linked binary. If you were to use a dynamically linked executable, then you'd have to arrange to have the shared libraries in the right place under your new root directory.

For example, if you create a statically linked 'ls' executable, and put it in /tmp/empty, you can run this command as root:

```
$ chroot /tmp/empty /ls -Rl /
```

Then you'll see output like this:

```
/:
total 1023
-rwxr-xr-x    1 0        0         1041745 Aug 16 11:17 ls
```

If you want to use a dynamically linked executable, say `bash`, then first run '`ldd bash`' to see what shared objects it needs. Then, in addition to copying the actual binary, also copy the listed files to the required positions under your intended new root directory. Finally, if the executable requires any other files (e.g., data, state, device files), copy them into place, too.

11.2 `env`: Run a command in a modified environment

`env` runs a command with a modified environment. Synopses:

```
env [option]... [name=value]... [command [args]...]
env
```

Arguments of the form '*variable*=*value*' set the environment variable *variable* to value *value*. *value* may be empty ('*variable*='). Setting a variable to an empty value is different from unsetting it.

The first remaining argument specifies the program name to invoke; it is searched for according to the `PATH` environment variable. Any remaining arguments are passed as arguments to that program.

If no command name is specified following the environment specifications, the resulting environment is printed. This is like specifying a command name of `printenv`.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-u` *name*'
'`--unset=`*name*'
: Remove variable *name* from the environment, if it was in the environment.

'`-`'
'`-i`'
'`--ignore-environment`'
: Start with an empty environment, ignoring the inherited environment.

11.3 `nice`: Run a command with modified scheduling priority

`nice` prints or modifies the scheduling priority of a job. Synopsis:

`nice [`*option*`]... [`*command* `[`*arg*`]...]`

If no arguments are given, `nice` prints the current scheduling priority, which it inherited. Otherwise, `nice` runs the given *command* with its scheduling priority adjusted. If no *adjustment* is given, the priority of the command is incremented by 10. You must have appropriate privileges to specify a negative adjustment. The priority can be adjusted by `nice` over the range of -20 (the highest priority) to 19 (the lowest).

Because most shells have a built-in command by the same name, using the unadorned command name in a script or interactively may get you different functionality than that described here.

The program accepts the following option. Also see Chapter 2 [Common options], page 2.

'`-n` *adjustment*'
'`-`*adjustment*'
'`--adjustment=`*adjustment*'
: Add *adjustment* instead of 10 to the command's priority.

11.4 `nohup`: Run a command immune to hangups

`nohup` runs the given *command* with hangup signals ignored, so that the command can continue running in the background after you log out. Synopsis:

`nohup` *command* `[`*arg*`]...`

`nohup` increases the scheduling priority of *command* by 5, so it has a slightly smaller change to run. If standard output is a terminal, it and standard error are redirected so that they are appended to the file '`nohup.out`'; if that cannot be written to, they are appended to the file '`$HOME/nohup.out`'. If that cannot be written to, the command is not run.

If `nohup` creates either '`nohup.out`' or '`$HOME/nohup.out`', it creates it with no "group" or "other" access permissions. It does not change the permissions if the output file already existed.

`nohup` does not automatically put the command it runs in the background; you must do that explicitly, by ending the command line with an '`&`'.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

11.5 `su`: Run a command with substitute user and group id

`su` allows one user to temporarily become another user. It runs a command (often an interactive shell) with the real and effective user id, group id, and supplemental groups of a given *user*. Synopsis:

`su [`*option*`]... [`*user* `[`*arg*`]...]`

If no *user* is given, the default is `root`, the super-user. The shell to use is taken from *user*'s `passwd` entry, or '`/bin/sh`' if none is specified there. If *user* has a password, `su` prompts for the password unless run by a user with effective user id of zero (the super-user).

By default, `su` does not change the current directory. It sets the environment variables `HOME` and `SHELL` from the password entry for *user*, and if *user* is not the super-user, sets `USER` and `LOGNAME` to *user*. By default, the shell is not a login shell.

Any additional *args* are passed as additional arguments to the shell.

GNU `su` does not treat '`/bin/sh`' or any other shells specially (e.g., by setting `argv[0]` to '`-su`', passing `-c` only to certain shells, etc.).

`su` can optionally be compiled to use `syslog` to report failed, and optionally successful, `su` attempts. (If the system supports `syslog`.) However, GNU `su` does not check if the user is a member of the `wheel` group; see below.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-c` *command*'
'`--command=`*command*'
: Pass *command*, a single command line to run, to the shell with a `-c` option instead of starting an interactive shell.

'`-f`'
'`--fast`'
: Pass the `-f` option to the shell. This probably only makes sense if the shell run is `csh` or `tcsh`, for which the `-f` option prevents reading the startup file ('`.cshrc`'). With Bourne-like shells, the `-f` option disables file name pattern expansion (globbing), which is not likely to be useful.

'`-`'
'`-l`'
'`--login`'
: Make the shell a login shell. This means the following. Unset all environment variables except `TERM`, `HOME`, and `SHELL` (which are set as described above), and `USER` and `LOGNAME` (which are set, even for the super-user, as described above), and set `PATH` to a compiled-in default value. Change to *user*'s home directory. Prepend '`-`' to the shell's name, intended to make it read its login startup file(s).

'`-m`'
'`-p`'
'`--preserve-environment`'

Do not change the environment variables `HOME`, `USER`, `LOGNAME`, or `SHELL`. Run the shell given in the environment variable `SHELL` instead of the shell from *user*'s passwd entry, unless the user running `su` is not the superuser and *user*'s shell is restricted. A *restricted shell* is one that is not listed in the file '`/etc/shells`', or in a compiled-in list if that file does not exist. Parts of what this option does can be overridden by `--login` and `--shell`.

'`-s` *shell*'
'`--shell=`*shell*'

Run *shell* instead of the shell from *user*'s passwd entry, unless the user running `su` is not the superuser and *user*'s shell is restricted (see '`-m`' just above).

Why GNU `su` does not support the '`wheel`' group

(This section is by Richard Stallman.)

Sometimes a few of the users try to hold total power over all the rest. For example, in 1984, a few users at the MIT AI lab decided to seize power by changing the operator password on the Twenex system and keeping it secret from everyone else. (I was able to thwart this coup and give power back to the users by patching the kernel, but I wouldn't know how to do that in Unix.)

However, occasionally the rulers do tell someone. Under the usual `su` mechanism, once someone learns the root password who sympathizes with the ordinary users, he or she can tell the rest. The "wheel group" feature would make this impossible, and thus cement the power of the rulers.

I'm on the side of the masses, not that of the rulers. If you are used to supporting the bosses and sysadmins in whatever they do, you might find this idea strange at first.

12 Delaying

12.1 `sleep`: Delay for a specified time

`sleep` pauses for an amount of time specified by the sum of the values of the command line arguments. Synopsis:

`sleep` *number*`[smhd]`...

Each argument is a number followed by an optional unit; the default is seconds. The units are:

'`s`' seconds

'`m`' minutes

'`h`' hours

'`d`' days

Historical implementations of `sleep` have required that *number* be an integer. However, GNU `sleep` accepts arbitrary floating point numbers.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

13 Numeric operations

These programs do numerically-related operations.

13.1 `factor`: Print prime factors

`factor` prints prime factors. Synopses:

```
factor [number]...
factor option
```

If no *number* is specified on the command line, `factor` reads numbers from standard input, delimited by newlines, tabs, or spaces.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

The algorithm it uses is not very sophisticated, so for some inputs `factor` runs for a long time. The hardest numbers to factor are the products of large primes. Factoring the product of the two largest 32-bit prime numbers takes over 10 minutes of CPU time on a 400MHz Pentium II.

```
$ p=`echo '4294967279 * 4294967291'|bc`
$ factor $p
18446743979220271189: 4294967279 4294967291
```

In contrast, `factor` factors the largest 64-bit number in just over a tenth of a second:

```
$ factor `echo '2^64-1'|bc`
18446744073709551615: 3 5 17 257 641 65537 6700417
```

13.2 `seq`: Print numeric sequences

`seq` prints a sequence of numbers to standard output. Synopses:

```
seq [option]... [first [increment]] last...
```

`seq` prints the numbers from *first* to *last* by *increment*. By default, *first* and *increment* are both 1, and each number is printed on its own line. All numbers can be reals, not just integers.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-f` *format*'
'`--format=`*format*'
: Print all numbers using *format*; default '`%g`'. *format* must contain exactly one of the floating point output formats '`%e`', '`%f`', or '`%g`'.

'`-s` *string*'
'`--separator=`*string*'
: Separate numbers with *string*; default is a newline. The output always terminates with a newline.

'`-w`'
'`--equal-width`'
: Print all numbers with the same width, by padding with leading zeroes. (To have other kinds of padding, use '`--format`').

If you want to use `seq` to print sequences of large integer values, don't use the default '%g' format since it can result in loss of precision:

```
$ seq 1000000 1000001
1e+06
1e+06
```

Instead, you can use the format, '%1.f', to print large decimal numbers with no exponent and no decimal point.

```
$ seq --format=%1.f 1000000 1000001
1000000
1000001
```

If you want hexadecimal output, you can use `printf` to perform the conversion:

```
$ printf %x'\n' `seq -f %1.f 1048575 1024 1050623`
fffff
1003ff
1007ff
```

For very long lists of numbers, use xargs to avoid system limitations on the length of an argument list:

```
$ seq -f %1.f 1000000 | xargs printf %x'\n' |tail -3
f423e
f423f
f4240
```

To generate octal output, use the printf `%o` format instead of `%x`. Note however that using printf works only for numbers smaller than `2^32`:

```
$ printf "%x\n" `seq -f %1.f 4294967295 4294967296`
ffffffff
bash: printf: 4294967296: Numerical result out of range
```

On most systems, seq can produce whole-number output for values up to `2^53`, so here's a more general approach to base conversion that also happens to be more robust for such large numbers. It works by using `bc` and setting its output radix variable, *obase*, to '16' in this case to produce hexadecimal output.

```
$ (echo obase=16; seq -f %1.f 4294967295 4294967296)|bc
FFFFFFFF
100000000
```

Be careful when using `seq` with a fractional *increment*, otherwise you may see surprising results. Most people would expect to see `0.3` printed as the last number in this example:

```
$ seq -s' ' 0 .1 .3
0 0.1 0.2
```

But doesn't happen on most systems because `seq` is implemented using binary floating point arithmetic (via the C `double` type) – which means some decimal numbers like `.1` cannot be represented exactly. That in turn means some nonintuitive conditions like `.1 * 3 > .3` will end up being true.

To work around that in the above example, use a slightly larger number as the *last* value:

```
$ seq -s' ' 0 .1 .31
0 0.1 0.2 0.3
```

In general, when using an *increment* with a fractional part, where (*last* - *first*) / *increment* is (mathematically) a whole number, specify a slightly larger (or smaller, if *increment* is negative) value for *last* to ensure that *last* is the final value printed by seq.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

Table of Contents