POSIX.DOC
Microsoft® Windows NT™ Resource Kit--POSIX Utilities

Windows NT® Resource Kits

This document contains important information about the POSIX utilities that is not included in the Windows NT Resource Kit online Help or printed documents.

Using Write to View This Document

If you enlarge the Write window to its maximum size, this document will be easier to read. To do so, click the Maximize button in the upper-right corner of the window. Or open the Control menu in the upper-left corner of the Write window (press ALT+SPACEBAR), and then choose the Maximize command.

To move through the document, press PAGE UP or PAGE DOWN or click the arrows at the top and bottom of the scroll bar along the right side of the Write window.

To print the document, choose the Print command from the File menu.

For Help on using Write, press F1.

To read other online documents, choose the Open command from the File menu.

Contents

This document contains information on these commands:

ar, page 2
cat, page 5
cc, page 6
chmod, page 21
cp, page 24
find, page 26
ln, page 30
ls, page 31
make, page 34
mkdir, page 42
mv, page 43
rm, page 44
rmdir, page 45
sh, page 46
touch, page 58
wc, page 59

For more details about the **ar, cc, devsrv, find, ld, make, sh,** and **vi** commands, see RKTOOLS.HLP in the Resource Kit program group.

NAME

ar -- create and maintain library archives

SYNOPSIS

```
ar -d[-Tv] archive file ...
ar -m[-Tv] archive file ...
ar -m[-abiTv] position archive file ...
ar -p[-Tv] archive [file ...]
ar -q[-cTv] archive file ...
ar -r[-cuTv] archive file ...
ar -r[-abciuTv] position archive file ...
ar -t[-Tv] archive [file ...]
ar -x[-ouTv] archive [file ...]
```

DESCRIPTION

The *ar* utility creates and maintains groups of files combined into an archive. Once an archive has been created, new files can be added and existing files can be extracted, deleted, or replaced.

Files are named in the archive by a single component; for example, if a file referenced by a path containing a slash ("/") is archived, it will be named by the last component of that path. When matching paths are listed on the command line against file names stored in the archive, only the last component of the path will be compared.

All information and error messages use the path listed on the command line, if any was specified; otherwise the name in the archive is used. If multiple files in the archive have the same name and paths are listed on the command line to "select" archive files for an operation, only the first file with a matching name will be selected.

The normal use of are is for creating and maintaining libraries that are suitable for use with the loader (see *ld*) although it is not restricted to this purpose. The options follow.

-a

A positioning modifier used with the options **-r** and **-m**. The files are entered or moved after the archive member position, which must be specified.

-b

A positioning modifier used with the options **-r** and **-m**. The files are entered or moved before the archive member *position*, which must be specified.

-c

Whenever an archive is created, an information message to that effect is written to standard error. If the **-c** option is specified, *ar* creates the archive silently.

-d

Deletes the specified archive files.

-i

A positioning modifier used with the options **-r** and **-m**. The files are entered or moved **before** the archive member position, which must be specified. (Identical to the **-b** option.)

-m

Moves the specified archive files within the archive. If one of the options **-a**, **-b**, or **-i** is specified, the files are moved before or after the *position* file in the archive. If none of those options are specified, the files are moved to the end of the archive.

-o

Sets the access and modification times of extracted files to the modification time of the file when it was entered into the archive. This will fail if the user is not the owner of the extracted file or the superuser.

-p

Writes the contents of the specified archive files to the standard output. If no files are specified, the contents of all the files in the archive are written in the order they appear in the archive.

-q

(Quickly) appends the specified files to the archive. If the archive does not exist, a new archive file is created. When creating a large archive piece-by-piece, this is much faster than the **-r** option, as no checking is done to see if the files already exist in the archive.

-r

Replaces or adds the specified files to the archive. If the archive does not exist, a new archive file is created. Files that replace existing files do not change the order of the files within the archive. New files are appended to the archive unless one of the options **-a**, **-b**, or **-i** is specified.

-T

Selects and/or names archive members using only the first 15 characters of the archive member or command line file name. The historic archive format had 16 bytes for the name, but some historic archiver and loader implementations were unable to handle names that used the entire space. This means that file names that are not unique in their first 15 characters can subsequently be confused. A warning message is printed to the standard error output if any file names are truncated.

-t

Lists the specified files in the order in which they appear in the archive, each on a separate line. If no files are specified, all files in the archive are listed.

-u

Updates files. When used with the **-r** option, files in the archive will be replaced only if the disk file has a newer modification time than the file in the archive. When used with the **-x** option, files in the archive will be extracted only if the archive file has a newer modification time than the file on disk.

-v

Provides verbose output. When used with the **-d**, **-m**, **-q**, or **-x** options, *ar* gives a file-by-file description of the archive modification. This description consists of three, white-space separated fields: the option letter, a dash ("-"), and the file name. When used with the **-r** option, *ar* displays the description as above, but the initial letter is an "a" if the file is added to the archive and an "r" if the file replaces a file already in the archive.

When used with the **-p** option, the name of each printed file is written to the standard output before the contents of the file (preceded by a single newline character and followed by two newline characters, enclosed in less-than ("<") and greater-than (">") characters).

When used with the **-t** option, *ar* displays an "ls -l" style listing of information about the members of the archive. This listing consists of eight, white-space separated fields: the file permissions, the decimal user and group ID's separated by a single slash ("/"), the file size (in bytes), the file modification time (in the *date*(1) format "%b %e %H:%M %Y"), and the name of the file.

-x

Extracts the specified archive members into the files named by the command line arguments. If no members are specified, all the members of the archive are extracted into the current directory.
If the file does not exist, it is created; if it does exist, the owner and group will be unchanged. The file access and modification times are the time of the extraction (also see the **-o** option). The file permissions will be set to those of the file when it was entered into the archive; this will fail if the user is not the owner of the extracted file or the superuser.

The *ar* utility exits 0 on success, and >0 if an error occurs.

ENVIRONMENT

TMPDIR

The pathname of the directory to use when creating temporary files.

FILES

/tmp

default temporary file directory

ar.XXXXXX

temporary file names

COMPATIBILITY

By default, *ar* writes archives that may be incompatible with historic archives, as the format used for storing archive members with names longer than 15 characters has changed. This implementation of *ar* is backward-compatible with previous versions of *ar* in that it can read and write (using the **-T** option) historic archives. The **-T** option is provided for compatibility only and will be deleted in a future release.

STANDARDS

The *ar* utility is expected to offer a superset of the POSIX 1003.2 functionality.

NAME

cat -- concatenate and print files

SYNOPSIS

```
cat [-b] [-e] [-n] [-s] [-t] [-u] [-v] [file ...]
```

DESCRIPTION

The *cat* utility reads files sequentially, writing them to the standard output. The *file* operands are processed in command line order. A single dash represents standard input. The options follow.

-b

Implies the **-n** option but does not number blank lines.

-e

Implies the **-v** option, and displays a dollar sign ("$") at the end of each line as well.

-n

Numbers the output lines, starting at 1.

-s

Squeezes multiple adjacent empty lines, causing the output to be single-spaced.

-t

Implies the **-v** option and displays tab characters as "^I" as well.

-u

The **-u** option guarantees that the output is unbuffered.

-v

Displays nonprinting characters so they are visible. Control characters print line "^X" for control-X; the delete character (octal 0177) prints as "^?". Non-ASCII characters (with the high bit set) are printed as "M-" (for meta) followed by the character for the low 7 bits.

The *cat* utility exits 0 on success and >0 if an error occurs.

BUGS

Because of the shell language mechanism used to perform output redirection, the command

```
cat file1 file2 > file1
```

will cause the original data in file1 to be destroyed!

HISTORY

A *cat* command appeared in Sixth Edition AT&T UNIX.

NAME

cc -- GNU project C Compiler

SYNOPSIS

```
cc [options] file ...
```

DESCRIPTION

cc is a version of the GNU C compiler. It accepts a dialect of ANSI C with extensions; this dialect is different from the dialect used in 4.3 BSD and earlier distributions. The **-traditional** flag causes the compiler to accept a dialect of extended Classic C, much like the C of these earlier distributions. If you are not already familiar with ANSI C and its new features, you will want to build your software with
-traditional.

DIFFERENCES

Most older C compiler flags are supported by *cc*. Three that are not are **-go**, to generate symbol tables for the unsupported *sdb* debugger; **-f**, for single precision floating point in expressions (which is now the default); and **-t**, for alternate compiler passes.

The differences between ANSI C and Classic C dialects are too numerous to describe here in detail. The following quick summary is intended to make users aware of potential subtle problems when converting Classic C code to ANSI C.

The most obvious change is the pervasive use of *function prototypes*. Under the ANSI C dialect, the compiler checks number and type of arguments to C library functions when standard header files are included; calls that fail to match will yield errors. A subtle consequence of adding prototype declarations is that user code which inadvertently redefines a C library function may break; for example it is no longer possible to write an *abort* function that takes different parameters or returns a different value from the standard *abort*, when including standard header files.

Another issue with prototypes is that functions which take different parameter types no longer have the same type; function pointers now differ by parameter types as well as return types. Variable argument lists are handled differently; the old *varargs*(3) package is obsolete; it was replaced by *stdarg*(3), which unfortunately is not completely compatible. A subtle change in type promotion can be confusing: small unsigned types are now widened into signed types rather than unsigned types. A similar problem can occur with the **sizeof** operator, which now yields an unsigned type rather than a signed type. One common problem is due to a change in scoping: external declarations are now scoped to the block they occur in, so a declaration for (say) **errno** inside one block will no longer declare it in all subsequent blocks. The syntax for braces in structure initializations is now a bit stricter, and it is sometimes necessary to add braces to please the compiler.

Two very subtle and sometimes very annoying features apply to constant strings and to the *longjmp*(3) function. Constant strings in the ANSI dialect are read-only; attempts to alter them cause protection violations. This ANSI feature permits the compiler to coalesce identical strings in the same source file; and, since the read-only part of a binary is sharable, it saves space when multiple copies of a binary are running at the same time. The most common difficulty with read-only strings lies with the use of the **mktemp** function, which in the past often altered a constant string argument. It is now necessary to copy a constant string before it may be altered. The **longjmp** function may now destroy any register *or* stack variable in the function that made the corresponding call to the **setjmp** function; to protect a local variable, the new ANSI **volatile** modifier must be used. This often leads to confusing situations upon 'return' from **setjmp**. The compiler has extended warning flags for dealing with read-only strings and **setjmp**, but these are not very effective.

If your code has problems with any of these ANSI features, you will probably want to use **-traditional**. Even with **-traditional**, there are some differences between this dialect of Classic C and the dialect supported on older distributions.

There are at least two differences that are a consequence of the fact that *cc* uses an ANSI C style grammar for both traditional and ANSI modes. The old C dialect permitted a typedef to replace a simple type in the idiom "unsigned *type*"; this *cc* treats such forms as syntax errors. The old C dialect also permitted formal parameters to have the same names as typedef types; the current dialect does not.

Some questionable or illegal practices that were supported in the old C dialect are not supported by
-traditional: noncomment text at the end of a "#include" preprocessor control line is an error, not ignored; compound assignment operators must not contain white space, *e.g.* "* =" is not the same as "*="; the last member declaration in a structure or union must be terminated by a semicolon; it is not possible to "**switch**" on function pointers; more than one occurrence of "#else" at the same level in a preprocessor "#if" clause is an error, not ignored.

Some truly ancient C practices are no longer supported. The idiom of declaring an anonymous structure and using its members to extract fields from other structures or even nonstructures is illegal. Integers are not automatically converted to pointers when they are dereferenced. The **-traditional** dialect does not retain the so-called "old-fashioned" assignment operators (with the "=" preceding rather than following the operator) or initializations (with no "=" between initializer and initializee).

WARNING

The rest of this topic is an extract of the documentation of the *GNU C compiler* and is limited to the meaning of the options. **It is not kept up to date.** If you want to be certain of the information below, check it in the manual "Using and Porting GCC". Refer to the Info file **gcc.info** or the DVI file **gcc.dvi**, which are made from the Texinfo source file **gcc.texinfo**.

The *GNU C compiler* uses a command syntax much like the UNIX C compiler. The *cc* program accepts options and file names as operands. Multiple single-letter options may *not* be grouped: **-dr** is very different from **-d -r**.

When you invoke GNU CC, it normally does preprocessing, compiling, assembly, and linking. File names which end in **.c** are taken as C source to be preprocessed and compiled; file names ending in **.i** are taken as preprocessor output to be compiled; compiler output files plus any input files with names ending in **.s** are assembled; then the resulting object files, plus any other input files, are linked to produce an executable.
Command options allow you to stop this process at an intermediate stage. For example, the **-c** option says not to run the linker. Then the output consists of object files output by the assembler.

Other command options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; these are not documented here, but you rarely need to use any of them.

OPTIONS

Here are the options to control the overall compilation process, including those that say whether to link, whether to assemble, and so on.

-o *file*

Places output in file *file*. This applies regardless of whatever sort of output is being produced -- whether it is an executable file, an object file, an assembler file, or preprocessed C code.
If **-o** is not specified, the default is to put an executable file in **a.out**, the object file *source***.c** in *source***.o**, an assembler file in *source***.s**, and preprocessed C on standard output.

-c

Compiles or assembles the source files, but does not link. Produces object files with names made by replacing **.c** or **.s** with **.o** at the end of the input file names. Does nothing at all for object files specified as input.

-S

Compiles into assembler code but does not assemble. The assembler output file name is made by replacing **.c** with **.s** at the end of the input file name. Does nothing at all for assembler source files or object files specified as input.

-E

Runs only the C preprocessor. Preprocess all the C source files specified and outputs the results to standard output.

-v

Compiler driver program prints the commands it executes as it runs the preprocessor, compiler proper, assembler, and linker. Some of these are directed to print their own version numbers.

-pipe

Uses pipes rather than temporary files to communicate between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

-B*prefix*

Compiler driver program tries *prefix* as a prefix for each program it tries to run. These programs are *cpp*, *cc1*, *as*, and *ld*.

For each subprogram to be run, the compiler driver first tries the **-B** prefix, if any. If that name is not found, or if **-B** was not specified, the driver tries a standard prefix (which currently is **/usr/libexec/**). If this does not result in a file name that is found, the unmodified program name is searched for using the directories specified in your **PATH** environment variable.

You can get a similar result from the environment variable **GCC_EXEC_PREFIX**; if it is defined, its value is used as a prefix in the same way. If both the **-B** option and the **GCC_EXEC_PREFIX** variable are present, the **-B** option is used first and the environment variable value is used second.

-b*prefix*

The argument *prefix* is used as a second prefix for the compiler executables and libraries. This prefix is optional: the compiler tries each file first with it, then without it. This prefix follows the prefix specified with **-B** or the default prefixes.

Thus, **-bvax- -Bcc/** in the presence of environment variable **GCC_EXEC_PREFIX** with definition **/u/foo/** causes GNU CC to try the following file names for the preprocessor executable.

```
cc/vax-cpp
cc/cpp
/u/foo/vax-cpp
/u/foo/cpp
/usr/libexec/vax-cpp
/usr/libexec/cpp
```

The following options control the details of C compilation itself.

-ansi

Supports all ANSI standard C programs.

This turns off certain features of GNU C that are incompatible with ANSI C, such as the **asm**, **inline**, and **typeof** keywords and predefined macros such as **unix** and **vax** that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature.
The alternate keywords **__asm__**, **__inline__**, and **__typeof__** continue to work despite **-ansi**. You would not want to use them in an ANSI C program, of course; but it useful to put them in header files that might be included in compilations done with **-ansi**.

Alternate predefined macros such as **__unix__** and **__vax__** are also available, with or without **-ansi**.

The **-ansi** option does not cause non-ANSI programs to be rejected gratuitously. For that,
-pedantic is required in addition to **-ansi**.

The macro **__STRICT_ANSI__** is predefined when the **-ansi** option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard does not call for; this is to avoid interfering with any programs that might use these names for other things.

-traditional

Attempts to support some aspects of traditional C compilers. Specifically:

* All **extern** declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.

* The keywords **typeof**, **inline**, **signed**, **const**, and **volatile** are not recognized.

* Comparisons between pointers and integers are always allowed.

* Integer types **unsigned short** and **unsigned char** promote to **unsigned int**.

* Out-of-range floating point literals are not an error.

* All automatic variables not declared **register** are preserved by *longjmp*(3C). Ordinarily, GNU C follows ANSI C: automatic variables not declared **volatile** may be clobbered.

* In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.

* In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.

* The predefined macro **__STDC__** is not defined when you use **-traditional**, but **__GNUC__** is (since the GNU extensions which **__GNUC__** indicates are not affected by **-traditional**). If you need to write header files that work differently (depending upon whether **-traditional** is in use) you can distinguish four situations by testing both of these predefined macros: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers.

-O

Optimizes. Optimizing compilation takes somewhat more time and a lot more memory for a large function.

Without **-O**, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent--if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without **-O**, only variables declared **register** are allocated in registers. The resulting compiled code is a little worse than produced by PCC without **-O**.

With **-O**, the compiler tries to reduce code size and execution time.

Some of the **-f** options described below turn specific kinds of optimization on or off.

-g

Produces debugging information in the operating system's native format (for *dbx* or *sdb*). *gdb* also can work with this debugging information.

Unlike most other C compilers, GNU CC allows you to use **-g** with **-O**. The shortcuts taken by optimized code may occasionally produce surprising results--some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops. Nevertheless, it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

-w

Inhibits all warning messages.

-W

Prints extra warning messages for the following events.

* An automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you do not specify **-O**, you simply will not get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared **volatile**; or whose address is taken; or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions, or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
\ \ int x;
\ \ switch (y)
\ \ \ \ {
\ \ \ \ case 1:  x = 1;
\ \ \ \ \ \ break;
\ \ \ \ case 2:  x = 4;
\ \ \ \ \ \ break;
\ \ \ \ case 3:  x = 5;
\ \ \ \ }
\ \ foo (x);
}
```

If the value of *y* is always 1, 2, or 3, then *x* is always initialized; however, GNU CC does not know this. The following example demonstrates another common case.

```
{
\ \ int save_y;
\ \ if (change_y) save_y = y, y = new_y;
\ \ ...
\ \ if (change_y) y = save_y;
}
```

This has no bug because *save_y* is used only if it is set.
Some spurious warnings can be avoided if you declare as **volatile** all the functions you use that never return.

*   A nonvolatile automatic variable might be changed by a call to *longjmp*(3C). These warnings are possible only in optimizing compilation.

The compiler sees only the calls to *setjmp*(3C). It cannot know where *longjmp*(3C) will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is, in fact, no problem because *longjmp*(3C) cannot, in fact, be called at the place that would cause a problem.

*   A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, the following function would evoke such a warning.

```
foo (a)
{
\ \ if (a > 0)
\ \ \ \ return a;
}
```

Spurious warnings can occur because GNU CC does not realize that certain functions (including *abort*(3C) and *longjmp*(3C)) will never return.

*   An expression statement contains no side effects.

In the future, other useful warnings also may be enabled by this option.

-Wimplicit
Warns whenever a function is implicitly declared.

-Wreturn-type

Warns whenever a function is defined with a return-type that defaults to **int**. Also warns about any **return** statement with no return-value in a function whose return-type is not **void**.

-Wunused

Warns whenever a local variable is unused aside from its declaration and whenever a function is declared static but never defined.

-Wswitch

Warns whenever a **switch** statement has an index of enumeral type and lacks a **case** for one or more of the named codes of that enumeration. (The presence of a **default** label prevents this warning.) **case** labels outside the enumeration range also provoke warnings when this option is used.

-Wcomment

Warns whenever a comment-start sequence /* appears in a comment.

-Wtrigraphs

Warns if any trigraphs are encountered (assuming they are enabled).

-Wall

All of the above **-W** options combined. These are all the options that pertain to usage which we do not recommend and that we believe is always easy to avoid, even in conjunction with macros.
The other **-W...** options below are not implied by **-Wall** because certain kinds of useful macros are almost impossible to write without causing those warnings.

-Wshadow

Warns whenever a local variable shadows another local variable.

-Wid-clash-*len*

Warns whenever two distinct identifiers match in the first *len* characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

-Wpointer-arith

Warns about anything that depends upon the size of a function type or of **void**. GNU C assigns these types a size of 1 for convenience in making calculations with **void *** pointers and pointers to functions.

-Wcast-qual

Warns whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warns if a **const char *** is cast to an ordinary **char ***.

-Wwrite-strings

Gives string constants the type **const char**[*length*] so that copying the address of one into a non-**const char *** pointer will get a warning. At compile time, these warnings will help you find code that can try to write into a string constant, but only if you have been

very careful about using **const** in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make
-Wall request these warnings.

-p
Generates extra code to write profile information suitable for the analysis program *prof*(1).

-pg
Generates extra code to write profile information suitable for the analysis program *gprof*(1).

-a
Generates extra code to write profile information for basic blocks, suitable for the analysis program *tcov*(1). Eventually, GNU *gprof*(1) should be extended to process this data.

-l*library*
Searchs a standard list of directories for a library named *library*, which is actually a file named **lib***library***.a**. The linker uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify
with **-L**.

Normally, the files found this way are library files--archive files whose members are object files. The linker handles an archive file by scanning through it for members that define symbols which have, so far, been referenced but not defined. But, if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an **-l** option and specifying a file name is that **-l** searches several directories.

-L*dir*
Adds directory *dir* to the list of directories to be searched for **-l**.

-nostdlib
Does not use the standard system libraries and startup files when linking. Only the files you specify (plus **gnulib**) will be passed to the linker.

-m*machinespec*
This is a machine-dependent option that specifies something about the type of target machine. These options are defined by the macro **TARGET_SWITCHES** in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.
The following **-m** options are defined in the 68000 machine description:

-m68020
-mc68020

Generates output for a 68020 (rather than a 68000). This is the default if you use the unmodified sources.

-m68000
-mc68000
Generates output for a 68000 (rather than a 68020).

-m68881
Generates output containing 68881 instructions for floating point. This is the default if you use the unmodified sources.

-mfpa
Generates output containing Sun FPA instructions for floating point.

-msoft-float
Generates output containing library calls for floating point.

-mshort
Considers type **int** to be 16 bits wide, like **short int**.

-mnobitfield
Does not use the bit-field instructions. **-m68000** implies **-mnobitfield**.

-mbitfield
Does use the bit-field instructions. **-m68020** implies **-mbitfield**. This is the default if you use the unmodified sources.

-mrtd
Uses a different function-calling convention, in which functions that take a fixed number of arguments return with the **rtd** instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.
This calling convention is incompatible with the one normally used on UNIX, so you cannot use it if you need to call libraries compiled with the UNIX compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including *printf*(3S)); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

The **rtd** instruction is supported by the 68010 and 68020 processors but not by the 68000.

The following **-m** options are defined in the Vax machine description.

-munix
Does not output certain jump instructions (**aobleq** and so on) that the UNIX assembler for the Vax cannot handle across long ranges.

-mgnu
Does output those jump instructions, on the assumption that you will assemble with the GNU assembler.

-mg
Outputs code for g-format floating point numbers instead of d-format.

The following **-m** switches are supported on the Sparc.

-mfpu
Generates output containing floating point instructions. This is the default if you use the unmodified sources.

-msoft-float
Generates output containing library calls for floating point.

-mno-epilogue
Generates separate return instructions for **return** statements. This has both advantages and disadvantages.

The following **-m** options are defined in the Convex machine description.

-mc1
Generates output for a C1. This is the default when the compiler is configured for a C1.

-mc2
Generates output for a C2. This is the default when the compiler is configured for a C2.

-margcount
Generates code that puts an argument count in the word preceding each argument list. Some nonportable Convex and Vax programs need this word. (Debuggers do not; this information is in the symbol table.)

-mnoargcount
Omits the argument count word. This is the default if you use the unmodified sources.

-f*flag*
Specifies machine-independent flags. Most flags have both positive and negative forms; for example, the negative form of **-ffoo** would be **-fno-foo**. In the table below, only one of the forms is listed--the one that is not the default. You can figure out the other form by either removing **no-** or adding it.

-fpcc-struct-return
Uses the same convention for returning **struct** and **union** values that is used by the usual C compiler on your system. This convention is less efficient for small structures; and, on

many machines, it fails to be reentrant. However, it has the advantage of allowing intercallability between GCC-compiled code and PCC-compiled code.

-ffloat-store

Does not store floating-point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a **double** is supposed to have.

For most programs, the excess precision does only good; however, a few programs rely upon the precise definition of IEEE floating point. Use **-ffloat-store** for such programs.

-fno-asm

Does not recognize **asm**, **inline**, or **typeof** as a keyword. These words may then be used as identifiers. You can use **__asm__**, **__inline__**, and **__typeof__** instead.

-fno-defer-pop

Always pops the arguments to each function call as soon as that function returns. Normally, the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

-fstrength-reduce

Optimizes loop strength reduction and eliminates iteration variables.

-fcombine-regs

Allows the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to **-O**.

-fforce-mem

Forces memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load.

-fforce-addr

Forces memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as **-fforce-mem** may.

-fomit-frame-pointer

Does not keep the frame pointer in a register for functions that do not need one. This avoids the instructions to save, set up, and restore frame pointers. It also makes an extra register available in many functions. It also makes debugging impossible.

On some machines, such as the Vax, this flag has no effect because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it does not exist. The machine-description macro **FRAME_POINTER_REQUIRED** controls whether a target machine supports this flag.

-finline-functions

Integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated and the function is declared **static**, then the function is normally not output as assembler code in its own right.

-fcaller-saves

Enables values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those that have no call-preserved registers to use instead.

-fkeep-inline-functions

Outputs a separate run-time callable version of the function, even if all calls to a given function are integrated and the function is declared **static**.

-fwritable-strings

Stores string constants in the writable data segment and does not uniquize them. This is for compatibility with old programs that assume they can write into string constants. Writing into string constants is a very bad idea; constants should be constant.

-fcond-mismatch

Allows conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

-fno-function-cse

Does not put function addresses in registers; instead, it makes each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

-fvolatile

Considers all memory references through pointers to be volatile.

-fshared-data

Requests that the data and non-**const** variables of this compilation be shared data rather than private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

-funsigned-char

Lets the type **char** be the unsigned, like **unsigned char**.
Each kind of machine has a default for what **char** should be. It is either like **unsigned char** by default or like **signed char** by default. (Actually, at present, the default is always signed.)
The type **char** is always a distinct type from either **signed char** or **unsigned char**, even though its behavior is always just like one of those two.

Note that this is equivalent to **-fno-signed-char**, which is the negative form of **-fsigned-char**.

-fsigned-char
Lets the type **char** be signed, like **signed char**.

Note that this is equivalent to **-fno-unsigned-char**, which is the negative form of **-funsigned-char**.

-fdelayed-branch
If supported for the target machine, attempts to reorder instructions to exploit instruction slots available after delayed branch instructions.

-ffixed-*reg*
Treats the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer, or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the **REGISTER_NAMES** macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

-fcall-used-*reg*
Treats the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register *reg*.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

-fcall-saved-*reg*
Treats the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

-d*letters*
Says to make debugging dumps at times specified by *letters*. The following list defines the possible letters.

r	Dump after RTL generation.
j	Dump after first jump optimization.
J	Dump after last jump optimization.
s	Dump after CSE (including the jump optimization that sometimes follows CSE).
L	Dump after loop optimization.
f	Dump after flow analysis.
c	Dump after instruction combination.
l	Dump after local register allocation.
g	Dump after global register allocation.
d	Dump after delayed branch scheduling.
m	Print statistics on memory usage, at the end of the run.

-pedantic

Issues all the warnings demanded by strict ANSI standard C; rejects all programs that use forbidden extensions.

Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require **-ansi**). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected. There is no reason to use this option; it exists only to satisfy pedants.

-pedantic does not cause warning messages for use of the alternate keywords whose names begin and end with __.

-static

On Suns running version 4, this prevents linking with the shared libraries. (**-g** has the same effect.)
These options control the C preprocessor, which is run on each C source file before actual compilation. If you use the '-E' option, nothing is done except C preprocessing. Some of these options make sense only together with '-E' because they request preprocessor output that is not suitable for actual compilation.

-C

Tells the preprocessor not to discard comments. Used with the **-E** option.

-I*dir*

Searchs directory *dir* for include files.

-I-

Searches any directories specified with **-I** options before the **-I-** option only for the case of **#include "***file***"**; they are not searched for **#include** <*file*>.

If additional directories are specified with **-I** options after the **-I-**, searches these directories for all **#include** directives. (Ordinarily *all* **-I** directories are used this way.)

In addition, the **-I-** option inhibits the use of the current directory as the first search directory for **#include "***file***"**. Therefore, the current directory is searched only if it is requested explicitly with
-I.. Specifying both **-I-** and **-I.** allows you to control precisely which directories are searched before the current one and which are searched after.

-nostdinc

Does not search the standard system directories for header files. Only the directories you have specified with **-I** options (and the current directory, if appropriate) are searched.

Between **-nostdinc** and **-I-**, you can eliminate all directories from the search path except those you specify.

-M

Tells the preprocessor to output a rule suitable for *make*(1) describing the dependencies of each source file. For each source file, the preprocessor outputs one **make**-rule whose target is the object file name for that source file and whose dependencies are all the files **#include**d in it. This rule may be a single line or, if it is long, may be continued with \\-**newline**.

-M implies **-E**.

-MM

Like **-M** but the output mentions only the user-header files included with **#include "***file***"**. System header files included with **#include** <*file*> are omitted.

-MM implies **-E**.

-D*macro*

Defines macro *macro* with the empty string as its definition.

-D*macro=defn*

Defines macro *macro* as *defn*.

-U*macro*

Undefines macro *macro*.

-trigraphs

Supports ANSI C trigraphs. You do not want to know about this brain damage. The **-ansi** option also has this effect.

FILES

file.c	C source file
file.s	assembly language file
file.o	object file
a.out	link edited output
/tmp/cc*	temporary files
/usr/libexec/cpp	preprocessor
/usr/libexec/ccl	compiler
/usr/lib/libgnulib.a	library needed by GCC on some machines
/usr/lib/crt0.o	start-up routine
/usr/lib/libc.a	standard C library, see *intro*(3)
/usr/include	standard directory for **#include** files

BUGS

Bugs should be reported to **bug-gcc@prep.ai.mit.edu.** Bugs actually tend to get fixed if they can be isolated, so it is in your interest to report them in such a way that they can be easily reproduced.

COPYING

AUTHORS

See the GNU CC Manual for the contributors to GNU CC.

NAME

chmod -- change file modes

SYNOPSIS

```
chmod [-R] mode file ...
```

DESCRIPTION

The *chmod* utility modifies the file mode bits of the listed files as specified by the *mode* operand. The options follow.

-R

Traverses a file hierarchy. For each file that is of type directory, *chmod* changes the mode of all files in the file hierarchy below it followed by the mode of the directory itself.
Symbolic links are not indirected through nor are their modes altered.

Only the owner of a file or the superuser is permitted to change the mode of a file.
The *chmod* utility exits 0 on success and >0 if an error occurs.

MODES

Modes may be absolute or symbolic. An absolute mode is an octal number constructed by *or*ing the following values.

4000

set-user-ID-on-execution

2000

set-group-ID-on-execution

1000
: sticky bit, see *chmod*(2)

0400
: read by owner

0200
: write by owner

0100
: execute (or search for directories) by owner

0070
: read, write, execute/search by group

0007
: read, write, execute/search by others

The read, write, and execute/search values for group and others are encoded as described for owner.
The symbolic mode is described by the following grammar.

```
mode         ::= clause [, clause ...]
clause       ::= [who ...] [action ...] last_action
action       ::= op [perm ...]
last_action  ::= op [perm ...]
who          ::= a | u | g | o
op           ::= + | - | =
perm         ::= r | s | t | w | X | x | u | g | o
```

The *who* symbols "u", "g", and "o" specify the user, group, and other parts of the mode bits, respectively. The *who* symbol "a" is equivalent to "ugo".

The *perm* symbols represent the portions of the mode bits, as follows.

r
: The read bits.

s
: The set-user-ID-on-execution and set-group-ID-on-execution bits.

t
: The sticky bit.

w
: The write bits.

x
: The execute/search bits.

X
: The execute/search bits if the file is a directory or if any of the execute/search bits are set in the original (unmodified) mode. Operations with the *perm* symbol "X" are only meaningful in conjunction with the *op* symbol "+"; it is ignored in all other cases.

The *op* symbols represent the operation performed, as follows.

+

If no value is supplied for *perm*, the "+" operation has no effect. If no value is supplied for *who*, each permission bit specified in *perm* (for which the corresponding bit in the file mode creation mask is clear) is set. Otherwise, the mode bits represented by the specified *who* and *perm* values are set.

-

If no value is supplied for *perm*, the "-" operation has no effect. If no value is supplied for *who*, the mode bits represented by *perm* are cleared for the owner, group, and other permissions. Otherwise, the mode bits represented by the specified *who* and *perm* values are cleared.

=

The mode bits specified by the *who* value are cleared; or, if no *who* value is specified, the owner, group, and other mode bits are cleared. Then, if no value is supplied for *who*, each permission bit specified in *perm* (for which the corresponding bit in the file mode creation mask is clear) is set. Otherwise, the mode bits represented by the specified *who* and *perm* values are set.

Each *clause* specifies one or more operations to be performed on the mode bits, and each operation is applied to the mode bits in the order specified.

Operations upon the other permissions only (specified by the symbol "o" by itself), in combination with the *perm* symbols "s" or "t", are ignored.

EXAMPLES

644

Makes a file readable by anyone and writable by the owner only.

go-w

Denies write permission to group and others.

=rw,+X

Sets the read and write permissions to the usual defaults but retain any execute permissions that are currently set.

+X

Makes a directory or file searchable/executable by everyone if it is already searchable/executable by anyone.

755
u=rwx,go=rx
u=rwx,go=u-w

Makes a file readable/executable by everyone and writeable by the owner only.

go=

Clears all mode bits for group and others.

g=u-w

Sets the group bits equal to the user bits but clears the group write bit.

BUGS

There is no *perm* option for the naughty bits.

STANDARDS

The *chmod* utility is expected to be POSIX 1003.2-compatible with the exception of the *perm* symbols **t** and **X**, which are not included in that standard.

NAME

cp -- copy files

SYNOPSIS

```
cp [-R] [-f] [-h] [-i] [-p] source_file target_file
cp [-R] [-f] [-h] [-i] [-p] source_file ... target_directory
```

DESCRIPTION

In the first synopsis form, the *cp* utility copies the contents of the *source_file* to the *target_file*. In the second synopsis form, the contents of each named *source_file* is copied to thedestination *target_directory*. The names of the files themselves are not changed. If *cp* detects an attempt to copy a file to itself, the copy will fail.

The following options are available.

-R

If *source_file* designates a directory, *cp* copies the directory and the entire subtree connected at that point. This option also causes symbolic links to be copied, rather than indirected through, and for *cp* to create special files rather than copying them as normal files. Created directories have the same mode as the corresponding source directory, unmodified by the process' umask.

-f

For each existing destination pathname, removes it and creates a new file, without prompting for confirmation regardless of its permissions. (The **-i** option is ignored if the **-f** option is specified.)

-h

Forces *cp* to follow symbolic links. Provided for the **-R** option, which does not follow symbolic links by default.

-i

Causes *cp* to write a prompt to standard error before copying a file that would overwrite an existing file. If the response from the standard input begins with the character "y", the file is copied if permissions allow the copy.

-p

Causes *cp* to preserve in the copy as many of the modification times, access times, file modes, user IDs, and group IDs as allowed by permissions.

If the user ID and group ID cannot be preserved, no error message is displayed and the exit value is not altered.

If the source file has its set user ID bit on and the user ID cannot be preserved, the set user ID bit is not preserved in the copy's permissions. If the source file has its set group ID bit on and the group ID cannot be preserved, the set group ID bit is not preserved in the copy's permissions. If the source file has both the set user ID and set group ID bits on and either the user ID or group ID cannot be preserved, neither the set user ID nor the set group ID bits are preserved in the copy's permissions.

For each destination file that already exists, its contents are overwritten if permissions allow; but its mode, user ID, and group ID are unchanged.

If the destination file does not exist, the mode of the source file is used as modified by the file mode creation mask (see *sh*). If the source file has its set user ID bit on, that bit is removed unless both the source file and the destination file are owned by the same user.

If the source file has its set group ID bit on, that bit is removed unless both the source file and the destination file are in the same group and the user is a member of that group. If both the set user ID and set group ID bits are set, all of the above conditions must be fulfilled or both bits are removed.

Appropriate permissions are required for file creation or overwriting.

Symbolic links are followed unless the -R option is specified, in which case the link itself is copied.

cp exits 0 on success; >0 if an error occurred.

HISTORY

The *cp* command is expected to be POSIX 1003.2 compatible.

NAME

find -- walk a file hierarchy

SYNOPSIS

```
find [-d] [-s] [-X] [-x] [-f file] file ...  expression
```

DESCRIPTION

find recursively descends the directory tree for each *file* listed, evaluating an *expression* (composed of the "primaries" and "operands" listed below) in terms of each file in the tree.

If *file* is a symbolic link referencing an existing file, the directory tree referenced by the link is descended instead of the link itself.

The options follow.

-d

The **-d** option causes *find* to perform a depth-first traversal; i.e., directories are visited in post-order and all entries in a directory will be acted upon before the directory itself. By default, *find* visits directories in preorder; i.e., before their contents. Note that the default is *not* a breadth-first traversal.

-f

The **-f** option specifies a file hierarchy for *find* to traverse. File hierarchies also may be specified as the operands immediately following the options.

-s

The **-s** option causes the file information and file type, returned for each symbolic link, to be those of the file referenced by the link, not the link itself. If the referenced file does not exist, the file information and type will be for the link itself.

-X

The **-X** option is a modification to permit *find* to be safely used in conjunction with *xargs*(1). If a file name contains any of the delimiting characters used by *xargs*, a diagnostic message is displayed on standard error and the file is skipped. The delimiting characters include single (" ' ") and double (" " ") quotes, backslash ("\"), space, tab, and newline characters.

-x

The **-x** option prevents *find* from descending into directories that have a device number different than that of the file from which the descent began.

PRIMARIES

-atime *n*

True, if the difference between the file last access time and the time *find* was started (rounded up to the next full 24-hour period) is *n* 24-hour periods.

-ctime *n*

True, if the difference between the time of last change of file status information and the time *find* was started (rounded up to the next full 24-hour period) is *n* 24-hour periods.

-exec *utility* [argument ...] **;**

True, if the program named *utility* returns a zero value as its exit status. Optional arguments may be passed to the utility. The expression must be terminated by a semicolon (";"). If the string "{}" appears anywhere in the utility name or the arguments, it is replaced by the pathname of the current file. *utility* will be executed from the directory from which IfIiInId was executed.

-fstype *type*

True, if the file is contained in a file system of type *type*. Currently supported types are "local", "mfs", "nfs", "pc", "rdonly", and "ufs". The types "local" and "rdonly" are not specific file system types. The former matches any file system physically mounted on the system where the *find* is being executed, and the latter matches any file system which is mounted read-only.

-group *gname*

True, if the file belongs to the group *gname*. If *gname* is numeric and there is no such group name, then *gname* is treated as a group id.

-inum *n*

True, if the file has inode number *n*.

-links *n*

True, if the file has *n* links.

-ls

This primary always evaluates to true. The following information for the current file is written to standard output: its inode number, size in 512-byte blocks, file permissions, number of hard links, owner, group, size in bytes, last modification time, and pathname. If the file is a block or character-special file, the major and minor numbers will be displayed instead of the size in bytes. If the file is a symbolic link, the pathname of the linked-to file will be displayed preceded by "->". The format is identical to that produced by "ls -dgils".

-mtime *n*

True, if the difference between the file last modification time and the time *find* was started (rounded up to the next full 24-hour period) is *n* 24-hour periods.

-ok *utility* [*argument* ...] **;**

The **-ok** primary is identical to the **-exec** primary with the exception that *find* requests user affirmation for executing the utility by printing a message to the terminal and reading a response. If the response is other than "y", the command is not executed and the value of the **-ok** expression is false.

-name *pattern*

True, if the last component of the pathname being examined matches *pattern*. Special shell pattern matching characters ("[", "]", "*", and "?") may be used as part of *pattern*. These characters may be matched explicitly by escaping them with a backslash ("\").

-newer *file*

True, if the current file has a more recent last modification time than *file*.

-nouser

True, if the file belongs to an unknown user.

-nogroup

True, if the file belongs to an unknown group.

-path *pattern*

True, if the pathname being examined matches *pattern*. Special shell pattern matching characters ("[", "]", "*", and "?") may be used as part of *pattern*. These characters may be matched explicitly by escaping them with a backslash ("\"). Slashes ("/") are treated as normal characters and do not need to be matched explicitly.

-perm *mode*

The *mode* may be either symbolic (see *chmod*) or an octal number. If the mode is symbolic, a starting value of zero is assumed and the mode sets or clears permissions without regard to the process' file mode creation mask. If the mode is octal, only bits 07777 (*S_ISUID*|*S_ISGID*|*S_ISTXT*|*S_IRWXU*|*S_IRWXG*|*S_IRWXO*) of the file's mode bits participate in the comparison. If the mode is preceded by a dash ("-"), this primary evaluates to true if at least all of the bits in the mode are set in the file's mode bits. If the mode is not preceded by a dash, this primary evaluates to true if the bits in the mode exactly match the file's mode bits. Note that the first character of a symbolic mode may not be a dash ("-").

-print

This primary always evaluates to true. It prints the pathname of the current file to standard output. The expression is appended to the user specified expression if neither **-exec**, **-ls**, nor **-ok** is specified.

-prune

This primary always evaluates to true. It causes *find* to not descend into the current file. Note that the **-prune** primary has no effect if the **-d** option was specified.

-size *n*[**c**]

True, if the file's size (rounded up in 512-byte blocks) is *n*. If *n* is followed by a "c", then the primary is true if the file's size is *n* bytes.

-type *t*

True, if the file is of the specified type. Possible file types follow.

b	block special
c	character special
d	directory
f	regular file
l	symbolic link
p	FIFO
s	socket

-user *uname*

True, if the file belongs to the user *uname*. If *uname* is numeric and there is no such user name, then *uname* is treated as a user id.

All primaries which take a numeric argument allow the number to be preceded by a plus sign ("+") or a minus sign ("-"). A preceding plus sign means "more than n"; a preceding minus sign means "less than n"; and neither sign means "exactly n".

OPERATORS

The primaries may be combined using the following operators. The operators are listed in order of decreasing precedence.

(*expression***)**

This evaluates to true if the parenthesized expression evaluates to true.

!*expression*

This is the unary *NOT* operator. It evaluates to true if the expression is false.

expression **-and** *expression*
expression expression

The **-and** operator is the logical *AND* operator. As it is implied by the juxtaposition of two expressions, it does not need to be specified. The expression evaluates to true if both expressions are true. The second expression is not evaluated if the first expression is false.

expression **-or** *expression*

The **-or** operator is the logical *OR* operator. The expression evaluates to true if either the first or the second expression is true. The second expression is not evaluated if the first expression is true.

All operands and primaries must be separate arguments to *find*. Primaries which themselves take arguments expect each argument to be a separate argument to *find*.

EXAMPLES

The following examples are shown as given to the shell.

find / \! -name "*.c" -print

Prints out a list of all the files whose names do not end in ".c".

find / -newer ttt -user wnj -print

Prints out a list of all the files owned by user "wnj" that are newer than the file "ttt".

find / \! \(-newer ttt -user wnj \)

Prints out a list of all the files that are not both newer than "ttt" and owned by "wnj".

find / \(-newer ttt -or -user wnj \)

Prints out a list of all the files that are either owned by "wnj" or that are newer than "ttt".

STANDARDS

The *find* utility syntax is a superset of the syntax specified by the POSIX 1003.2 standard.

The **-s** and **-X** options and the **-inum** and **-ls** primaries are extensions to POSIX 1003.2.

Historically, the **-d**, **-s**, and **-x** options were implemented using the primaries "-depth", "-follow", and "-xdev". These primaries always evaluated to true. As they were really global variables that took effect before the traversal began, some legal expressions could have unexpected results. An example is the expression "-print -o -depth". As **-print** always evaluates to true, the standard order of evaluation implies that **-depth** would never be evaluated. This is not the case.

The operator "-or" was implemented as "-o", and the operator "-and" was implemented as "-a".

Historic implementations of the **-exec** and **-ok** primaries did not replace the string "{}" in the utility name or the utility arguments if it had preceding or following nonwhitespace characters. This version replaces it no matter where in the utility name or arguments it appears.

BUGS

The special characters used by *find* also are special characters to many shell programs. In particular, the characters "*", "[", "]", "?", "(", ")", "!", "\", and ";" may have to be escaped from the shell.

As there is no delimiter separating options and file names or file names and the *expression*, it is difficult to specify files named "-xdev" or "!". These problems are handled by the **-f** option and the *getopt*(3) "--" construct.

NAME

ln -- make links

SYNOPSIS

```
ln [-s] source_file target_file
ln [-s] source_file ...  target_directory
```

DESCRIPTION

The *ln* utility creates a new directory entry (linked file), which inherits the same modes as the orginal file. It is useful for maintaining multiple copies of a file in many places at once--without the "copies"; instead, a link 'points' to the original copy. There are two types of links: hard links and symbolic links. How a link 'points' to a file is one of the differences between a hard or symbolic link.

Option available:

-s

Create a symbolic link.

By default *ln* makes *hard* links. A hard link to a file is indistinguishable from the original directory entry; any changes to a file are effective independent of the name used to reference the file. Hard links may not refer to directories (unless the proper incantations are supplied) and may not span file systems.

A symbolic link contains the name of the file to which it is linked. The referenced file is used when an *open*(2) operation is performed on the link. A *stat*(2) on a symbolic link will return the linked-to file; an *lstat*(2) must be done to obtain information about the link. The *readlink*(2) call may be used to read the contents of a symbolic link. Symbolic links may span file systems and may refer to directories.

Given one or two arguments, *ln* creates a link to an existing file *source_file*. If *target_file* is given, the link has that name; *target_file* may also be a directory in which to place the link; otherwise it is placed in the current directory. If only the directory is specified, the link will be made to the last component of *source_file*.

Given more than two arguments, *ln* makes links in *target_directory* to all the named source files. The links made will have the same name as the files being linked to.

HISTORY
An *ln* command appeared in Sixth Edition AT&T UNIX.
NAME
ls -- list directory contents.

SYNOPSIS
```
ls [-ACFLRTacdfgiklqrstu1] file ...
```

DESCRIPTION
For each operand that names a *file* of a type other than directory, *ls* displays its name as well as any requested, associated information. For each operand that names a *file* of type directory, *ls* displays the names of files contained within that directory, as well as any requested, associated information.

If no operands are given, the contents of the current directory are displayed. If more than one operand is given, nondirectory operands are displayed first; directory and nondirectory operands are sorted separately and in lexicographical order.

The following options are available.

-A
: Lists all entries except for "." and "..". Always set for the superuser.

-C
: Forces multicolumn output; this is the default when output is to a terminal.

-F
: Displays a slash (/) immediately after each pathname that is a directory, an asterisk (*) after each that is executable, and an at sign (@) after each symbolic link.

-L
: If argument is a symbolic link, lists the file or directory the link references rather than the link itself.

-R
: Recursively lists subdirectories encountered.

-T
: Displays complete time information for the file, including month, day, hour, minute, second, and year.

-a
: Includes directory entries whose names begin with a dot (.).

-c
: Uses time when file status was last changed for sorting or printing.

-d
: Lists directories as plain files (not searched recursively).

-f
: Does not sort output.

-g
: Includes the group ownership of the file in a long "l" output "lg". If the group is not a known group name, the numeric ID is printed.

-i
: For each file, prints the file's file serial number (inode number).

-k
: Modifies the **-s** option, causing the sizes to be reported in kilobytes.

-l

(The lowercase letter "ell.") Lists in long format. (See below.) If the output is to a terminal, a total sum for all the file sizes is output on a line before the long listing.

-q

Forces printing of nongraphic characters in file names as the character '?'; this is the default when output is to a terminal.

-r

Reverses the order of the sort to get reverse lexicographical order or the oldest entries first.

-s

Displays the number of file system bytes actually used by each file, in units of 512, where partial units are rounded up to the next integer value. If the output is to a terminal, a total sum for all the file sizes is output on a line before the listing.

-t

Sort by time modified (most recently modified first) before sorting the operands by lexicographical order.

-u

Uses time of last access, instead of last modification of the file for sorting "t" or printing "l".

-1

(The numeric digit "one.") Forces output to be one entry per line. This is the default when output is not to a terminal.

The **-1**, **-C**, and **-l** options all override each other; the last one specified determines the format used.
The **-c** and **-u** options override each other; the last one specified determines the file time used.

By default, *ls* lists one entry per line to standard output; the exceptions are to terminals or when the
-C option is specified.

File information is displayed with one or more *blank*s separating the information associated with the **-i**, **-s**, and **-l** options.

THE LONG FORMAT

If the **-l** option is given, the following information will be displayed: file mode, number of links, owner name, number of bytes in the file, abbreviated month, day-of-month file was last modified, hour file last modified, minute file last modified, and the pathname.

If the owner name is not a known user name, the numeric ID is displayed.

If the file is a character-special or block-special file, the major and minor device numbers for the file are displayed in the size field. If the file is a symbolic link, the pathname of the linked-to file is preceded
by "->".

The file mode printed under the **-l** option consists of the the entry type, owner permissions, and group permissions. The entry type character describes the type of file, as follows.

b Block special file.

c Character special file.
d Directory.
l Symbolic link.
s Socket link.
- Regular file.

The next three fields are three characters each: owner permissions, group permissions, and other permissions. Each field has three character positions.

If **r**, the file is readable; if **-**, it is not readable.
If **w**, the file is writable; if **-**, it is not writable.

The first of the following applies.

S If in the owner permissions, the file is not executable and set-user-ID mode is set. If in the group permissions, the file is not executable and set-group-ID mode is set.

s If in the owner permissions, the file is executable and set-user-ID mode is set. If in the group permissions, the file is executable and set-group-ID mode is set.

x The file is executable or the directory is searchable.

- The file is neither readable, writeable, exectutable, nor set-user-ID nor set-group-ID mode nor sticky. (See below.)

These next two apply only to the third character in the last group (other permissions).

T The sticky bit is set (mode **1000**) but is not executable nor contains search permission. (See *chmod*(1) or *sticky*(8).)

t The sticky bit is set (mode **1000**) and is searchable or executable. (See *chmod*(1) or *sticky*(8).)

The *ls* utility exits 0 on success, and >0 if an error occurs.

ENVIRONMENT

The following environment variables affect the execution of *ls*:

COLUMNS
If this variable contains a string representing a decimal integer, it is used as the column position width for displaying multiple- text-column output. The *ls* utility calculates how many pathname text columns to display based on the width provided. (See **-C**.)

HISTORY

An *ls* command appeared in Sixth Edition AT&T UNIX.

NAME

make -- maintain program dependencies

SYNOPSIS

`make [-eiknqrstv] [-D variable] [-d flags] [-f makefile]`
`[-I directory] [-j max_jobs] [variable=value ...] target ...`

DESCRIPTION

make is a program designed to simplify the maintenance of other programs. Its input is a list of specifications as to the files upon which programs and other files depend. If the file "makefile" exists, it is read for this list of specifications. If it does not exist, the file "Makefile" is read. If the file ".depend" exists, it is read. This manual page is intended as a reference document only.

The options follow.

-D *variable*

Defines *variable* to be 1, in the global context.

-d *flags*

Turns on debugging and specifies which portions of *make* are to print debugging information. *flags* is one or more of the following:

A	Prints all possible debugging information; equivalent to specifying all of the debugging flags.
a	Prints debugging information about archive searching and caching.
c	Prints debugging information about conditional evaluation.
d	Prints debugging information about directory searching and caching.
g1	Prints the input graph before making anything.
g2	Prints the input graph after making everything or before exiting on error.
j	Prints debugging information about running multiple shells.
m	Prints debugging information about making targets, including modification dates.
s	Prints debugging information about suffix-transformation rules.
t	Prints debugging information about target list maintenance.
v	Prints debugging information about variable assignment.
-e	Specifies that environmental variables override macro assignments within makefiles.
-f makefile	Specifies a makefile to read instead of the default "makefile" and "Makefile". If makefile is -, standard input is read. Multiple makefiles may be specified and are read in the order specified.

-I *directory* Specifies a directory in which to search for makefiles and included makefiles. The system makefile directory is automatically included as part of this list.

-i Ignores nonzero exit of shell commands in the makefile. Equivalent to specifying **-** before each command line in the makefile.

-j *max_jobs* Specifies the maximum number of jobs that *make* may have running at any one time.

-k Continues processing after errors are encountered but only on those targets that do not depend upon the target whose creation caused the error.

-n Displays the commands that would have been executed, but do not actually execute them.

-q Does not execute any commands but does exit 0 if the specified targets are up to date; otherwise, it specifies 1.

-r Does not use the built-in rules specified in the system makefile.

-s Does not echo any commands as they are executed. Equivalent to specifying _B@ before each command line in the makefile.

-t Rather than rebuilding a target as specified in the makefile, creates it or updates its modification time to make it appear up to date.

variable=value
Set the value of the variable *variable* to *value*.

There are six different types of lines in a makefile: file dependency specifications, shell commands, variable assignments, include statements, conditional directives, and comments.

In general, lines may be continued from one line to the next by ending them with a backslash ("\"). The trailing newline character and initial whitespace on the following line are compressed into a single space.

FILE DEPENDENCY SPECIFICATIONS

Dependency lines consist of one or more targets, an operator, and zero or more sources. This creates a relationship where the targets "depend" upon the sources and are usually created from them. The exact relationship between the target and the source is determined by the operator that separates them. The three operators are as follows:

:
A target is considered out of date if its modification time is less than those of any of its sources. Sources for a target accumulate over dependency lines when this operator is used. The target is removed if *make* is interrupted.

!

Targets are always re-created but not until all sources have been examined and re-created as necessary. Sources for a target accumulate over dependency lines when this operator is used. The target is removed if *make* is interrupted.

::

If no sources are specified, the target is always re-created. Otherwise, a target is considered out of date if any of its sources has been modified more recently than the target. Sources for a target do not accumulate over dependency lines when this operator is used. The target will not be removed if *make* is interrupted.

Targets and sources may contain the shell wildcard values "?", "*", "[]", and "{}". The values "?", "*", and "[]" may only be used as part of the final component of the target or source and must be used to describe existing files. The value "{}" need not necessarily be used to describe existing files. Expansion is in directory order not alphabetically as done in the shell.

SHELL COMMANDS

Each target may have associated with it a series of shell commands, normally used to create the target. Each of the commands in this script **must** be preceded by a tab. While any target may appear on a dependency line, only one of these dependencies may be followed by a creation script, unless the "::" operator is used.

If the first or first two characters of the command line are "@" and/or "-", the command is treated specially. A "@" causes the command not to be echoed before it is executed. A "-" causes any nonzero exit status of the command line to be ignored.

VARIABLE ASSIGNMENTS

Variables in *make* are much like variables in the shell; and, by tradition, consist of all uppercase letters. The five operators that can be used to assign values to variables are as follows:

=

Assigns the value to the variable. Any previous value is overridden.

+=

Appends the value to the current value of the variable.

?=

Assigns the value to the variable if it is not already defined.

:=

Assigns with expansion, i.e., expands the value before assigning it to the variable. Normally, expansion is not done until the variable is referenced.

!=

Expands the value and passes it to the shell for execution and assigns the result to the variable. Any newlines in the result are replaced with spaces.

Any white-space before the assigned value is removed; if the value is being appended, a single space is inserted between the previous contents of the variable and the appended value.

Variables are expanded by surrounding the variable name with either curly braces ("{}") or parentheses ("()") and preceding it with a dollar sign ("$"). If the variable name contains only a single letter, the surrounding braces or parentheses are not required. This shorter form is not recommended.

Variable substitution occurs at two distinct times, depending upon where the variable is being used. Variables in dependency lines are expanded as the line is read. Variables in shell commands are expanded when the shell command is executed.

The four different classes of variables (in order of increasing precedence) are:

Environment variables
: Variables defined as part of *make*'s environment.

Global variables
: Variables defined in the makefile or in included makefiles.

Command line variables
: Variables defined as part of the command line.

Local variables
: Variables that are defined specific to a certain target. The seven local variables are as follows:

.ALLSRC	The list of all sources for this target; also known as ">".
.ARCHIVE	The name of the archive file.
.IMPSRC	The name/path of the source from which the target is to be transformed (the "implied" source); also known as "<".
.MEMBER	The name of the archive member.
.OODATE	The list of sources for this target that were deemed out of date; also known as "?".
.PREFIX	The file prefix of the file, containing only the file portion, no suffix or preceding directory components; also known as "*".
.TARGET	The name of the target; also known as "@". The shorter forms "@", "?", ">", and "*" are permitted for backward compatibility with historical makefiles and are not recommended. The six variables "@F", "@D", "<F", "<D", "*F", and "*D" are permitted for compatibility with AT&T System V makefiles and are not recommended.

Four of the local variables may be used in sources on dependency lines because they expand to the proper value for each target on the line. These variables are ".TARGET", ".PREFIX", ".ARCHIVE", and ".MEMBER".

In addition, *make* sets or knows about the following variables:

$
: A single dollar sign "$", i.e. "$$" expands to a single dollar sign.

.MAKE
: The name that *make* was executed with (*argv[0]*).

.CURDIR

A path to the directory where *make* was executed.

MAKEFLAGS

The environment variable "MAKEFLAGS" may contain anything that may be specified on *make*'s command line. Anything specified on *make*'s command line is appended to the "MAKEFLAGS" variable which is then entered into the environment for all programs which *make* executes.

Variable expansion may be modified to select or modify each word of the variable (where a "word" is white-space delimited sequence of characters). The general format of a variable expansion is:

```
{variable[:modifier[:...]]}
```

Each modifier begins with a colon and one of the following special characters. The colon may be escaped with a backslash ("\").

E

Replaces each word in the variable with its suffix.

H

Replaces each word in the variable with everything but the last component.

M *pattern*

Selects only those words that match the rest of the modifier. The standard shell wildcard characters ("*" and "?") may be used. The wildcard characters may be escaped with a backslash ("\").

N *pattern*

This is identical to **M**, but selects all words that do not match the rest of the modifier.

R

Replaces each word in the variable with everything but its suffix.

S /*old_pattern*/*new_pattern*/[**g**]

Modifies the first occurrence of *old_pattern* in each word to be replaced with *new_pattern*. If a **g** is appended to the last slash of the pattern, all occurrences in each word are replaced. If *old_pattern* begins with a caret ("^"), *old_pattern* is anchored at the beginning of each word. If *old_pattern* ends with a dollar sign ("$"), it is anchored at the end of each word. Inside *new_string*, an ampersand ("&") is replaced by *old_pattern*. Any character may be used as a delimiter for the parts of the modifier string. The anchoring, ampersand, and delimiter characters may be escaped with a backslash ("\").

Variable expansion occurs in the normal fashion inside both *old_string* and *new_string* with the single exception that a backslash is used to prevent the expansion of a dollar sign ("$"), not a preceding dollar sign as is usual.

T

Replaces each word in the variable with its last component.

old_string=new_string

This is the AT&T System V style variable substitution. It must be the last modifier specified. *old_string* is anchored at the end of each word, so only suffixes or entire words may be replaced.

INCLUDE STATEMENTS AND CONDITIONALS

Makefile inclusion and conditional structures reminiscent of the C programming language are provided in *make*. All such structures are identified by a line beginning with a single dot (".") character. Files are included by either ".include <*file*>" or ".include "*file*"". Variables between the angle brackets or double quotes are expanded to form the file name. If angle brackets are used, the included makefile is expected to be in the system makefile directory. If double quotes are used, the including makefile's directory and any directories specified using the **-I** option are searched before the system makefile directory.

Conditional expressions also are preceded by a single dot as the first chraracter of a line. The possible conditionals are listed below.

.undef *variable*
    Undefines the specified global variable. Only global variables may be undefined.
.if [**!**] *expression* [*operator expression ...*]
    Tests the value of an *expression*.
.ifdef [**!**] *variable* [*operator variable ...*]
    Tests the value of a *variable*.
.ifndef [**!**] *variable* [*operator variable ...*]
    Tests the value of a *variable*.
.ifmake [**!**] *target* [*operator target ...*]
    Tests the *target* being built.
.ifnmake [**!**] *target* [*operator target ...*]
    Tests the *target* being built.
.else
    Reverses the sense of the last conditional.
.elif [**!**] *expression* [*operator expression ...*]
    A combination of **.else** followed by **.if**.
.elifdef [**!**] *variable* [*operator variable ...*]
    A combination of **.else** followed by **.ifdef**.
.elifndef [**!**] *variable* [*operator variable ...*]
    A combination of **.else** followed by **.ifndef**.
.elifmake [**!**] *target* [*operator target ...*]
    A combination of **.else** followed by **.ifmake**.
.elifnmake [**!**] *target* [*operator target ...*]
    A combination of **.else** followed by **.ifnmake**.
.endif
    Ends the body of the conditional.

The *operator* may be any one of the following:

`||` `logical OR`

`&&` `Logical AND; of higher precedence than ||.`

As in C, *make* will only evaluate a conditional as far as is necessary to determine its value. Parentheses may be used to change the order of evaluation. The boolean operator "!" may be used to logically negate an entire conditional. It is of higher precendence than "&&".

The value of *expression* may be any of the following.

defined
: Takes a variable name as an argument and evaluates to true if the variable has been defined.

make
: Takes a target name as an argument and evaluates to true if the target was specified as part of *make*'s command line or was declared the default target (either implicitly or explicitly, see **.MAIN**) before the line containing the conditional.

empty
: Takes a variable, with possible modifiers, and evaluates to true if the expansion of the variable would result in an empty string.

exists
: Takes a file name as an argument and evaluates to true if the file exists. The file is searched for on the system search path (see **.PATH**).

target
: Takes a target name as an argument and evaluates to true if the target has been defined.

expression also may be an arithmetic or string comparison, with the left-hand side being a variable expansion. The standard C relational operators are all supported, and the usual number/base conversion is performed. Note, octal numbers are not supported. If the righthand value of a "==" or "!=" operator begins with a quotation mark ("") a string comparison is done between the expanded variable and the text between the quotation marks. If no relational operator is given, it is assumed that the expanded variable is being compared against 0.

When *make* is evaluating one of these conditional expressions and it encounters a word it does not recognize, either the "make" or "defined" expression is applied to it, depending upon the form of the conditional. If the form is **.ifdef** or **.ifndef**, the "defined" expression is applied. Similarly, if the form is **.ifmake** or **.ifnmake**, the "make" expression is applied.

If the conditional evaluates to true, the parsing of the makefile continues as before. If it evaluates to false, the following lines are skipped. In both cases this continues until a **.else** or **.endif** is found.

COMMENTS

Comments begin with a number sign ("#") character, anywhere but in a shell command line, and continue to the end of the line.

SPECIAL SOURCES

.IGNORE
: Ignores any errors from the commands associated with this target, exactly as if they all were preceded by a dash ("-").

.MAKE
: Executes the commands associated with this target, even if the **-n** or **-t** options were specified. Normally used to mark recursive *make*'s.

.NOTMAIN

Normally, *make* selects the first target it encounters as the default target to be built if no target was specified. This source prevents this target from being selected.

.OPTIONAL

If a target is marked with this attribute and *make* cannot determine how to create it, it will ignore this fact and assume that the file is not needed or already exists.

.PRECIOUS

When *make* is interrupted, it removes any partially made targets. This source prevents the target from being removed.

.SILENT

Does not echo any of the commands associated with this target, exactly as if they all were preceded by an at sign ("@").

.USE

Turns the target into *make*'s version of a macro. When the target is used as a source for another target, the other target acquires the commands, sources, and attributes (except for **.USE**) of the source. If the target already has commands, the **.USE** target's commands are appended to them.

SPECIAL TARGETS

Special targets may not be included with other targets; i.e., they must be the only target specified.

.BEGIN

Any command lines attached to this target are executed before anything else is done.

.DEFAULT

This is sort of a **.USE** rule for any target (that was used only as a source) that *make* cannot figure out any other way to create. Only the shell script is used. The **.IMPSRC** variable of a target that inherits **.DEFAULT**'s commands is set to the target's own name.

.END

Any command lines attached to this target are executed after everything else is done.

.IGNORE

Marks each of the sources with the **.IGNORE** attribute. If no sources are specified, this is the equivalent of specifying the **-i** option.

.INTERRUPT

If *make* is interrupted, the commands for this target will be executed.

.MAIN

If no target is specified when *make* is invoked, this target will be built.

.MAKEFLAGS

This target provides a way to specify flags for *make* when the makefile is used. The flags are as if typed to the shell, though the **-f** option will have no effect.

.PATH

The sources are directories that are to be searched for files not found in the current directory. If no sources are specified, any previously specified directories are deleted.

.PRECIOUS

Applies the **.PRECIOUS** attribute to any specified sources. If no sources are specified, the **.PRECIOUS** attribute is applied to every target in the file.

.SILENT

Applies the **.SILENT** attribute to any specified sources. If no sources are specified, the **.SILENT** attribute is applied to every command in the file.

.SUFFIXES

Each source specifies a suffix to *make*. If no sources are specified, any previously specified suffixes are deleted.

ENVIRONMENT
make utilizes the following environment variables, if they exist: MAKE, MAKEFLAGS,and MAKEOBJDIR.

FILES
.depend	list of dependencies
Makefile	list of dependencies
makefile	list of dependencies
sys.mk	system makefile
/usr/share/mk	system makefile directory

HISTORY
A *make* command appeared in Seventh Edition AT&T UNIX.

NAME
mkdir -- make directories

SYNOPSIS
```
mkdir [-p] directory_name ...
```

DESCRIPTION
mkdir creates the directories named as operands, in the order specified, using mode **0777** modified by the current *umask*.

The options follow.

-p

Creates intermediate directories as required. If this option is not specified, the full path prefix of each operand must already exist.

The user must have write permission in the parent directory.

mkdir exits 0 if successful and >0 if an error occurred.

STANDARDS
mkdir is POSIX 1003.2-compliant. This manual page is derived from the POSIX 1003.2 manual page.

NAME
mv -- move files

SYNOPSIS
```
mv [-f | -i] source target
mv [-f | -i] source ... directory
```

DESCRIPTION
In its first form, the *mv* utility renames the file named by the *source* operand to the destination path named by the *target* operand. This form is assumed when the last operand does not name an already existing directory.

In its second form, *mv* moves each file named by a *source* operand to a destination file in the existing directory named by the *directory* operand. The destination path for each operand is the pathname produced by the concatenation of the last operand, a slash, and the final pathname component of the named file.

The following options are available.

-f

Does not prompt for confirmation before overwriting the destination path. (The **-i** option is ignored if the **-f** option is specified.)

-i

Causes *mv* to write a prompt to standard error before moving a file that would overwrite an existing file. If the response from the standard input begins with the character "y'", the move is attempted.

It is an error for either the *source* operand or the destination path to specify a directory unless both do.
If the destination path does not have a mode that permits writing, *mv* prompts the user for confirmation as specified for the **-i** option.

As the *rename*(2) call does not work across file systems, *mv* uses *cp*(1) and *rm*(1) to accomplish the move. The effect is equivalent to:

```
rm -f destination_path && \
cp -pr source destination_path && \
rm -rf source
```

The *mv* utility exits 0 on success and >0 if an error occurs.

STANDARDS

The *mv* utility is expected to be POSIX 1003.2-compatible.

NAME

rm -- remove directory entries

SYNOPSIS

```
rm [-f | -i] [-d] [-R] [-r] file ...
```

DESCRIPTION

The *rm* utility attempts to remove the nondirectory type files specified on the command line. If the permissions of the file do not permit writing and the standard input device is a terminal, the user is prompted (on the standard error output) for confirmation.

The options are listed below.

-d

Attempts to remove directories as well as other types of files.

-f

Attempts to remove the files without prompting for confirmation, regardless of the file's permissions. If the file does not exist, does not display a diagnostic message nor modify the exit status to reflect an error. The **-f** option overrides any previous **-i** options.

-i

Requests confirmation before attempting to remove each file, regardless of the file's permissions or whether or not the standard input device is a terminal. The **-i** option overrides any previous
-f options.

-R

Attempts to remove the file hierarchy rooted in each file argument. The **-R** option implies the
-d option. If the **-i** option is specified, the user is prompted for confirmation before each directory's contents are processed (as well as before the attempt is made to remove the directory). If the user does not respond affirmatively, the file hierarchy rooted in that directory is skipped.

-r

Equivalent to **-R**.

The *rm* utility removes symbolic links, not the files referenced by the links.
It is an error to attempt to remove the files "." and "..".

The *rm* utility exits 0 if all of the named files or file hierarchies were removed or if the **-f** option was specified and all of the existing files or file hierarchies were removed. If an error occurs, *rm* exits with a value >0.

COMPATIBILITY

The *rm* utility differs from historical implementations in that the **-f** option only masks attempts to remove nonexistent files instead of masking a large variety of errors.

Also, historical *rm* implementations prompted on the standard output not the standard error output.

STANDARDS

The *rm* command is expected to be POSIX 1003.2-compatible.

NAME

rmdir -- remove directories

SYNOPSIS

```
rmdir directory ...
```

DESCRIPTION

The *rmdir* utility removes the directory entry specified by each *directory* argument, provided it is empty.
Arguments are processed in the order given. To remove both a parent directory and a subdirectory of that parent, the subdirectory must be specified first so that the parent directory is empty when *rmdir* tries to remove it.

The *rmdir* utility exits with one of the following values:

0 Each directory entry specified by a *directory* operand referred to an empty directory and was removed successfully.

>0 An error occurred.

STANDARDS

The *rmdir* function is expected to be POSIX 1003.2-compatible.

NAME

ash -- a shell

SYNOPSIS

`ash [-efIijnsxz] [+efIijnsxz] [-c command] [arg] ...`

COPYRIGHT

Copyright © 1989 by Kenneth Almquist.

DESCRIPTION

ash is a version of *sh*(1) with features similar to those of the System V shell. This manual page lists all the features of *ash* but concentrates upon the ones not in other shells.

Invocation

If the **-c** option is given, then the shell executes the specified shell command. The **-s** flag causes the shell to read commands from the standard input (after executing any command specified with the **-c** option). If neither the **-s** nor **-c** options are set, then the first *arg* is taken as the name of a file to read commands from. If this is impossible because there are no arguments following the options, then *ash* will set the **-s** flag and will read commands from the standard input.

The shell sets the initial value of the positional parameters from the *arg*s remaining after any *arg* used as the name of a file of commands is deleted.

The flags (other than **-c**) are set by preceding them with "-" and cleared by preceding them with "+"; see the *set* built-in command for a list of flags. If no value is specified for the **-i** flag, the **-s** flag is set and the standard input and output of the shell are connected to terminals; then, the **-i** flag will be set. If no value is specified for the **-j** flag, then the **-j** flag will be set if the **-i** flag is set.

When the shell is invoked with the **-c** option, it is good practice to include the **-i** flag if the command was entered interactively by a user. For compatibility with the System V shell, the **-i** option should come after the **-c** option.

If the first character of argument zero to the shell is "-", the shell is assumed to be a login shell; and the files **/etc/profile** and **.profile** are read if they exist. If the environment variable SHINIT is set upon entry to the shell, the commands in SHINIT are normally parsed and executed. SHINIT is not examined if the shell is a login shell or if it the shell is running a shell procedure. (A shell is considered to be running a shell procedure if neither the **-s** nor the **-c** options are set.)

Control Structures

A *list* is a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters. (This differs from the

System V shell, which, in most cases, requires a list to contain at least one command.) The commands in a list are executed in the order in which they are written. If a command is followed by an ampersand, the shell starts the command and immediately proceeds on to the next command; otherwise it waits for the command to terminate before proceeding to the next one.

"&&" and "||" are binary operators. "&&" executes the first command and then executes the second command if the exit status of the first command is zero. "||" is similar but executes the second command if the exit status of the first command is nonzero. "&&" and "||" both have the same priority.
The "|" operator is a binary operator that feeds the standard output of the first command into the standard input of the second command. The exit status of the "|" operator is the exit status of the second command. "|" has a higher priority than "||" or "&&".

An *if* command looks like the following example.

```
    if list
    then list
  [ elif list
      then list ] ...
  [ else list ]
    fi
```

A *while* command looks like the following example.

```
    while list
    do   list
    done
```

The two lists are executed repeatedly while the exit status of the first list is zero. The *until* command is similar; however, it has the word **until** in place of **while** and repeats until the exit status of the first list is zero.

The *for* command looks like the following example.

```
    for variable in word...
    do   list
    done
```

The words are expanded; and, then, the list is executed repeatedly with the variable set to each word in turn. **do** and **done** may be replaced with "{" and "}".

The *break* and *continue* commands look like the following example.

```
    break [ num ]
    continue [ num ]
```

break terminates the *num*'ths innermost **for** or **while** loops. **continue** continues with the next iteration of the *num*'ths innermost loop. These are implemented as built-in commands.

The *case* command looks like the following example.

```
case word in
pattern) list ;;
...
esac
```

The pattern can actually be one or more patterns (see **Patterns** below) separated by "|" characters. Commands may be grouped by writing either one of the following.

```
(list)
```

or

```
{ list; }
```

The first of these executes the commands in a subshell.

A function definition looks like the following example.

```
name ( ) command
```

A function definition is an executable statement; when executed it installs a function named *name* and returns an exit status of zero. The command is normally a list enclosed between "{" and "}".

Variables may be declared to be local to a function by using a *local* command. This should appear as the first statement of a function; it should looks like the following example.

```
local [ variable | - ] ...
```

local is implemented as a built-in command.

When a variable is made local, it inherits the initial value and exported and read-only flags from the variable with the same name in the surrounding scope, if there is one. Otherwise, the variable is initially unset. *ash* uses dynamic scoping so that, if you make the variable *x* local to function *f*, which then calls function *g*, references to the variable *x* made inside *g* will refer to the variable *x* declared inside *f*, not to the global variable named *x*.

The only special parameter that can be made local is "-". Making "-" local sets any shell options that are changed via the *set* command inside the function to be restored to their original values when the function returns.

The *return* command looks like the following example.

```
return [ exitstatus ]
```

It terminates the currently executing function. *return* is implemented as a built-in command.

Simple Commands
A simple command is a sequence of words. The execution of a simple command proceeds as follows. First, the leading words of the form "name=value" are stripped off and assigned to the environment of the command. Second, the words are expanded. Third, the first remaining word

is taken as the command name of the command that is located. Fourth, any redirections are performed. Fifth, the command is executed. We look at these operations in reverse order.

The execution of the command varies with the type of command. There are three types of commands: shell functions, built-in commands, and normal programs.

When a shell function is executed, all of the shell positional parameters (except $0, which remains unchanged) are set to the parameters to the shell function. The variables that are explicitly placed in the environment of the command (by placing assignments to them before the function name) are made local to the function and are set to values given. Then, the command given in the function definition is executed. The positional parameters are restored to their original values when the command completes.
Shell built-ins are executed internally to the shell, without spawning a new process.

When a normal program is executed, the shell runs the program, passing the parameters and the environment to the program. If the program is a shell procedure, the shell will interpret the program in a subshell. The shell will reinitialize itself in this case, so that the effect will be as if a new shell had been invoked to handle the shell procedure, except that the location of commands located in the parent shell will be remembered by the child. If the program is a file beginning with "#!", the remainder of the first line specifies an interpreter for the program. In this case, the shell (or the operating system, under Berkeley UNIX) will run the interpreter. The arguments to the interpreter will consist of any arguments given on the first line of the program, followed by the name of the program, followed by the arguments passed to the program.

Redirection
Input/output redirections can be intermixed with the words in a simple command and can be placed following any of the other commands. When redirection occurs, the shell saves the old values of the file descriptors and restores them when the command completes. The "<", ">", and ">>" redirections open a file for input, output, and appending, respectively. The "<&digit" and ">&digit" makes the input or output a duplicate of the file descriptor numbered by the digit. If a minus sign is used in place of a digit, the standard input or the standard output is closed.

The "<< word" redirection takes input from a *here* document. As the shell encounters "<<" redirections, it collects them. The next time it encounters an unescaped newline, it reads the documents in turn. The word following the "<<" specifies the contents of the line that terminates the document. If none of the quoting methods (', "", or \) are used to enter the word, then the document is treated like a word inside double quotes: "$" and backquote are expanded and backslash can be used to escape these and to continue long lines. The word cannot contain any variable or command substitutions, and its length (after quoting) must be in the range of 1 to 79 characters. If "<<-" is used in place of "<<", then leading tabs are deleted from the lines of the document. (This is to allow you to do indent shell procedures containing here documents in a natural fashion.)

Any of the preceding redirection operators may be preceded by a single digit specifying the file descriptor to be redirected. There cannot be any white space between the digit and the redirection operator.

Path Search

When locating a command, the shell first looks to see if it has a shell function by that name. Then, if PATH does not contain an entry for "%builtin", it looks for a built-in command by that name. Finally, it searches each entry in PATH in turn for the command.

The value of the PATH variable should be a series of entries separated by colons. Each entry consists of a directory name, or a directory name followed by a flag beginning with a percent sign. The current directory should be indicated by an empty directory name.

If no percent sign is present, then the entry causes the shell to search for the command in the specified directory. If the flag is "%builtin", then the list of shell built-in commands is searched. If the flag is "%func", then the directory is searched for a file that is read as input to the shell. This file should define a function whose name is the name of the command being searched for.

Command names containing a slash are simply executed without performing any of the above searches.

The Environment
The environment of a command is a set of name/value pairs. When the shell is invoked, it reads these names and values, sets the shell variables with these names to the corresponding values, and marks the variables as exported. The *export* command can be used to mark additional variables as exported.

The environment of a command is constructed by constructing name/value pairs from all the exported shell variables, and then by modifying this set by the assignments that precede the command, if any.

Expansion
The process of evaluating words when a shell procedure is executed is called *expansion*. Expansion consists of four steps: variable substitution, command substitution, word splitting, and file name generation. If a word is the expression following the word **case** in a case statement, the file name that follows a redirection symbol, or an assignment to the environment of a command, then the word cannot be split into multiple words. In these cases, the last two steps of the expansion process are omitted.

Command Substitution
ash accepts two syntaxes for command substitution, as listed below.

```
    'list'
```
and

```
    $(list)
```

Either of these may be included in a word. During the command substitution process, the command (syntactically a *list*) will be executed, and anything that the command writes to the standard output will be captured by the shell. The final newline (if any) of the output will be deleted; the rest of the output will be substituted for the command in the word.

Word Splitting
When the value of a variable or the output of a command is substituted, the resulting text is subject to word splitting, unless the dollar sign introducing the variable or backquotes containing

the text were enclosed in double quotes. In addition, "$@" is subject to a special type of splitting, even in the presence of double quotes.

ash uses two different splitting algorithms. The normal approach, which is intended for splitting text separated by white space, is used if the first character of the shell variable IFS is a space. Otherwise, an alternative experimental algorithm, which is useful for splitting (possibly empty) fields separated by a separator character, is used.

When performing splitting, the shell scans the replacement text looking for a character (when IFS does not begin with a space) or a sequence of characters (when IFS does begin with a space), deletes the character or sequence of characters, and spits the word into two strings at that point. When IFS begins with a space, the shell deletes either of the strings if they are null. As a special case, if the word containing the replacement text is the null string, the word is deleted.

The variable "$@" is special in two ways. First, splitting takes place between the positional parameters, even if the text is enclosed in double quotes. Second, if the word containing the replacement text is the null string and there are no positional parameters, then the word is deleted. The result of these rules is that "$@" is equivalent to "$1" "$2" ... "$*n*", where *n* is the number of positional parameters. (Note that this differs from the System V shell. The System V documentation claims that "$@" behaves this way; in fact on the System V shell "$@" is equivalent to "" when there are no positional paramteters.)

File Name Generation
Unless the **-f** flag is set, file name generation is performed after word splitting is complete. Each word is viewed as a series of patterns, separated by slashes. The process of expansion replaces the word with the names of all existing files whose names can be formed by replacing each pattern with a string that matches the specified pattern. There are two restrictions on this: first, a pattern cannot match a string containing a slash; and, second, a pattern cannot match a string starting with a period unless the first character of the pattern is a period.

If a word fails to match any files and the **-z** flag is not set, then the word will be left unchanged (except that the meta characters will be converted to normal characters). If the **-z** flag is set, then the word is only left unchanged if none of the patterns contain a character that can match anything other than itself. Otherwise, the **-z** flag forces the word to be replaced with the names of the files that it matches, even if there are zero names.

Patterns
A *pattern* consists of normal characters, which match themselves, and meta characters. The meta characters are "!", "*", "?", and "[". These characters lose there special meanings if they are quoted. When command or variable substitution is performed and the dollar sign or back quotes are not double quoted, the value of the variable or the output of the command is scanned for these characters and they are turned into meta characters.

Two exclamation points at the beginning of a pattern function as a "not" operator, causing the pattern to match any string that the remainder of the pattern does *not* match. Other occurrences of exclamation points in a pattern match exclamation points. Two exclamation points are required, rather than one, to decrease the incompatibility with the System V shell (which does not treat exclamation points specially).
An asterisk ("*") matches any string of characters. A question mark matches any single character. A left bracket ("[") introduces a character class. The end of the character class is

indicated by a "]"; if the "]" is missing, then the "[" matches a "[" rather than introducing a character class. A character class matches any of the characters between the square brackets. A range of characters may be specified using a minus sign. The character class may be complemented by making an exclamation point of the first character of the character class.

To include a "]" in a character class, make it the first character listed (after the "!", if any). To include a minus sign, make it the first or last character listed.

The /u Directory
By convention, the name "/u/user" refers to the home directory of the specified user. There are good reasons why this feature should be supported by the file system (using a feature such as symbolic links) rather than by the shell, but *ash* is capable of performing this mapping if the file system does not. If the mapping is done by *ash*, setting the **-f** flag will turn it off.

Character Set
ash silently discards NUL characters. Any other character will be handled correctly by *ash*, including characters with the high-order bit set.

Job Names and Job Control
The term *job* refers to a process created by a shell command or, in the case of a pipeline, to the set of processes in the pipeline. The ways to refer to a job follow.

```
%number %string %% process_id
```

The first form identifies a job by its job number. When a command is run, *ash* assigns it a job number (the lowest unused number is assigned). The second form identifies a job by giving a prefix of the command used to create the job. The prefix must be unique. If there is only one job, then the null prefix will identify the job, so you can refer to the job by writing "%". The third form refers to the *current job*. The current job is the last job to be stopped while it was in the foreground. (See the next paragraph.) The last form identifies a job by giving the process id of the last process in the job.

If the operating system that *ash* is running on supports job control, *ash* will allow you to use it. In this case, typing the suspend character (typically ^Z) while running a command will return you to *ash* and will make the suspended command the current job. You can then continue the job in the background by typing *bg*, or you can continue it in the foreground by typing *fg*.

ATTY
If the shell variable ATTY is set and the shell variable TERM is not set to "emacs", then *ash* generates appropriate escape sequences to talk to *atty*(1).

Exit Statuses
By tradition, an exit status of zero means that a command has succeeded, and a nonzero exit status indicates that the command failed. This is better than no convention at all; but, in practice, it is extremely useful to allow commands that succeed to use the exit status to return information to the caller. A variety of better conventions have been proposed, but none of them has met with universal approval. The convention used by *ash* and all the programs included in the *ash* distribution follows.

0	Success

1	Alternate success
2	Failure
129-...	Command terminated by a signal

The *alternate success* return is used by commands to indicate various conditions that are not errors but which can, with a little imagination, be conceived of as less successful than plain success. For example, *test* returns 1 when the tested condition is false and *getopts* returns 1 when there are no more options. Because this convention is not used universally, the **-e** option of *ash* causes the shell to exit when a command returns 1, even though that contradicts the convention described here.

When a command is terminated by a signal, it uses 128 plus the signal number as the exit code for the command.

Built-in Commands
This concluding section lists the built-in commands that are built in because they need to perform some operation that cannot be performed by a separate process. In addition to these, there are several other commands (*catf*, *echo*, *expr*, *line*, *nlecho*, *test*, ":", and *true*) that can optionally be compiled into the shell. The built-in commands described below that accept options use the System V Release 2 *getopt*(3) syntax.

bg [*job*] ...
Continues the specified jobs (or the current job if no jobs are given) in the background. This command is only available on systems with Berkeley job control.

bltin *command arg* ...
Executes the specified built-in command. (This is useful when you have a shell function with the same name as a built-in command.)

cd [*directory*]
Switchs to the specified directory (default $HOME). If an entry for CDPATH appears in the environment of the cd command or the shell variable CDPATH is set and the directory name does not begin with a slash, then the directories listed in CDPATH will be searched for the specified directory. The format of CDPATH is the same as that of PATH. In an interactive shell, the cd command will print out the name of the directory that it actually switched to if this is different from the name that the user gave. These may be different either because the CDPATH mechanism was used or because a symbolic link was crossed.

. *file*
The commands in the specified file are read and executed by the shell. A path search is not done to find the file because the directories in PATH generally contain files that are intended to be executed, not read.

eval *string* ...
The strings are parsed as shell commands and executed. (This differs from the System V shell, which concatenates the arguments (separated by spaces) and parses the result as a single command.)

exec [*command arg* ...]

Unless *command* is omitted, the shell process is replaced with the specified program (which must be a real program, not a shell built-in or function). Any redirections on the exec command are marked as permanent so that they are not undone when the exec command finishes. If the command is not found, the exec command causes the shell to exit.

exit [*exitstatus*]
Terminates the shell process. If *exitstatus* is given, it is used as the exit status of the shell; otherwise, the exit status of the preceding command is used.

export *name ...*
The specified names are exported so that they will appear in the environment of subsequent commands. The only way to unexport a variable is to unset it. *ash* allows the value of a variable to be set at the same time it is exported by writing the following.

```
export name=value
```

With no arguments, the export command lists the names of all exported variables.

fg [*job*]
Moves the specified job or the current job to the foreground. This command is only available on systems with Berkeley job control.

getopts *optstring var*
The System V *getopts*(1) command.

hash [**-rv**] *command ...*
The shell maintains a hash table that remembers the locations of commands. With no arguments whatsoever, the hash command prints out the contents of this table. Entries that have not been looked at since the last *cd* command are marked with an asterisk; it is possible for these entries to be invalid.

With arguments, the hash command removes the specified commands from the hash table (unless they are functions) and then locates them. With the **-v** option, *hash* prints the locations of the commands as it finds them. The **-r** option causes the *hash* command to delete all the entries in the hash table except for functions.

jobid [*job*]
Prints the process id's of the processes in the job. If the job argument is omitted, uses the current job.

jobs
This command lists out all the background processes that are children of the current shell process.

lc [*function-name*]
The function name is defined to execute the last command entered. If the function name is omitted, the last command executed is executed again. This command only works if the **-i** flag is set.

pwd

Prints the current directory. The built-in command may differ from the program of the same name because the built-in command remembers what the current directory is rather than recomputing it each time. This makes it faster. However, if the current directory is renamed, the built-in version of pwd will continue to print the old name for the directory.

read [**-p** *prompt*] [**-e**] *variable ...*

The prompt is printed if the **-p** option is specified and the standard input is a terminal. Then, a line is read from the standard input. The trailing newline is deleted from the line, and the line is split (as described in the section on word splitting above) and the pieces are assigned to the variables in order. If there are more pieces than variables, the remaining pieces (along with the characters in IFS that separated them) are assigned to the last variable. If there are more variables than pieces, the remaining variables are assigned the null string.

The **-e** option causes any backslashes in the input to be treated specially. If a backslash is followed by a newline, the backslash and the newline will be deleted. If a backslash is followed by any other character, the backslash will be deleted and the following character will be treated as though it were not in IFS, even if it is.

readonly *name ...*

The specified names are marked as read only, so that they cannot be subsequently modified nor unset. *ash* allows the value of a variable to be set at the same time it is marked read-only by writing the following.

```
readonly name=value
```

With no arguments, the read-only command lists the names of all read-only variables.

set [{ -*options* | +*options* | **--** }] [*arg ...*]

The *set* command performs three different functions.

With no arguments, it lists the values of all shell variables.

If options are given, it sets the specified option flags, or clears them if the option flags are introduced with a + rather than a **-**. Only the first argument to *set* can contain options. The possible options are listed below.

-e Causes the shell to exit when a command terminates with a nonzero exit status, except when the exit status of the command is explicitly tested. The exit status of a command is considered to be explicitly tested if the command is used to control an *if*, *elif*, *while*, or *until* or if the command is the left-hand operand of an "&&" or "||" operator.

-f Turns off file name generation.

-I Causes the shell to ignore end-of file-conditions. (This does not apply when the shell is a script source using the "." command.) The shell will, in fact, exit if it gets 50 EOF's in a row.

-i Makes the shell interactive. This causes the shell to prompt for input, to trap interrupts, to ignore quit and terminate signals, and to return to the main command loop rather than exiting upon error.

-j Turns on Berkeley job control on systems that support it. When the shell starts up, the
-j is set by default if the **-i** flag is set.

-n Causes the shell to read commands but not to execute them. (This is marginally useful for checking the syntax of scripts.)

-s If this flag is set when the shell starts up, the shell reads commands from its standard input. The shell does not examine the value of this flag at any other time.

-x If this flag is set, the shell will print out each command before executing it.

-z If this flag is set, the file name generation process may generate zero files. If it is not set, then a pattern that does not match any files will be replaced by a quoted version of the pattern.

The third use of the set command is to set the values of the shell's positional parameters to the specified *arg*s. To change the positional parameters without changing any options, use "**--**" as the first argument to *set*. If no *args* are present, the set command will leave the value of the positional parameters unchanged. Therefore, to set the positional parameters to a set of values that may be empty, execute the command

```
shift $#
```

first to clear out the old values of the positional parameters.

setvar *variable value*
Assigns *value* to *variable*. (In general, it is better to write *variable=value* rather than using *setvar*. *setvar* is intended to be used in functions that assign values to variables whose names are passed as parameters.)

shift [*n*]
Shift the positional parameters *n* times. A shift sets the value of $1 to the value of $2, the value of $2 to the value of $3, and so on, decreasing the value of $# by one. If there are zero positional parameters, shifting does not do anything.

trap [*action*] *signal* ...
Causes the shell to parse and execute *action* when any of the specified signals are received. The signals are specified by signal number. *action* may be null or omitted; the former causes the specified signal to be ignored and the latter causes the default action to be taken. When the shell forks off a subshell, it resets trapped (but not ignored) signals to the default action. The trap command has no effect upon signals that were ignored upon entry to the shell.

umask [*mask*]

Sets the value of umask to the specified octal value. If the argument is omitted, the umask value is printed.

unset *name ...*
The specified variables and functions are unset and unexported. If a given name corresponds to both a variable and a function, both the variable and the function are unset.

wait [*job*]
Waits for the specified job to complete and returns the exit status of the last process in the job. If the argument is omitted, waits for all jobs to complete and then returns an exit status of zero.

EXAMPLES

The following function redefines the *cd* command:

```
cd() {
    if bltin cd "$@"
    then if test -f .enter
    then .  .enter
    else return 0
    fi
    fi
}
```

This function causes the file ".enter" to be read when you enter a directory, if it exists. The *bltin* command is used to access the real *cd* command. The "return 0" ensures that the function will return an exit status of zero if it successfully changes to a directory that does not contain a ".enter" file. Redefining existing commands is not always a good idea, but this example shows that you can do it if you want to.
The suspend function distributed with *ash* looks like the following example.

```
  # Copyright (C) 1989 by Kenneth Almquist.  All rights
reserved.
  # This file is part of ash, which is distributed under the
terms
  # specified by the Ash General Public License.

  suspend() {
      local -
      set +j
      kill -TSTP 0
}
```

This turns off job control and then sends a stop signal to the current process group, which suspends the shell. (When job control is turned on, the shell ignores the TSTP signal.) Job control will be turned back on when the function returns because "-" is local to the function. As an example of what *not* to do, consider an earlier version of *suspend*, as illustrated below.

```
  suspend() {
      suspend_flag=$-
      set +j
      kill -TSTP 0
      set -$suspend_flag
  }
```

There are two problems with this. First, *suspend_flag* is a global variable rather than a local one, which will cause problems in the (unlikely) circumstance that the user is using that variable for some other purpose. Second, consider what happens if shell received an interrupt signal after it executes the first *set* command but before it executes the second one. The interrupt signal will abort the shell function, so that the second *set* command will never be executed and job control will be left off. The first version of *suspend* avoids this problem by turning job control off only in a local copy of the shell options. The local copy of the shell options is discarded when the function is terminated, no matter how it is terminated.

HINTS
Shell variables can be used to provide abbreviations for things that you type frequently. For example, if you set export h=$HOME in your.profile so that you can type the name of your home directory simply by typing "$h".

When writing shell procedures, try not to make assumptions about what is imported from the environment. Explicitly unset or initialize all variables, rather than assuming they will be unset. If you use *cd*, it is a good idea to unset CDPATH.

People sometimes use "<&-" or ">&-" to provide no input to a command or to discard the output of a command. A better way to do this is to redirect the input or output of the command to **/dev/null**.
Word splitting and file name generation are performed by default, and you must explicitly use double quotes to suppress it. This is backwards, but you can learn to live with it. Just get in the habit of writing double quotes around variable and command substitutions, and omit them only when you really want word splitting and file-name generation. If you want word splitting but not file-name generation, use the
-f option.

AUTHORS
Kenneth Almquist

BUGS
When command substitution occurs inside a here document, the commands inside the here document are run with their standard input closed. For example, the following will not work because the standard input of the *line* command will be closed when the command is run.

```
cat <<-!
Line 1: $(line)
Line 2: $(line)
!
```

Unsetting a function that is currently being executed may cause strange behavior.

The shell syntax allows a here document to be terminated by an end-of-file as well as by a line containing the terminator word that follows the "<<". What this means is that, if you mistype the terminator line, the shell will silently swallow up the rest of your shell script and stick it in the here document.

NAME

touch -- update last modification date of a file

SYNOPSIS

`touch [-c] [-f] file ...`

DESCRIPTION

The *touch* utility changes the modification and or access times of the given *file* operands.

Available options:

-c

Does not create a specified file if it does not exist. Does not write any diagnostic messages concerning this condition.

-f

Attempts to force the *touch* in spite of read and write permissions on a *file*.

HISTORY

A *touch* command appeared in Seventh Edition AT&T UNIX.

NAME

wc -- word, line, and byte count

SYNOPSIS

`wc [-c] [-l] [-w] [file ...]`

DESCRIPTION

The *wc* utility reads one or more input text *file*s, and, by default, writes the number of lines, words, and bytes contained in each input *file* to the standard output. If more than one input *file* is specified, a line of cumulative counts for all named *file*s is output on a separate line following the last file count. *wc* considers a word to be a maximal string of characters, delimited by white space.

The following options are available.

-c

The number of bytes in each input *file* is written to the standard output.

-l

The number of lines in each input *file* is written to the standard output.

-w

The number of words in each input *file* is written to the standard output.

When an option is specified, *wc* only reports the information requested by that option. The default action is equivalent to all the flags (**-lwc**) having been specified.

The following operands are available.

```
     file A pathname of an input file.
```

If no *file* names are specified, the standard input is used and a file name is not output. The resulting output is one line of the requested count(s), with the cumulative sum of all data read in via standard input.

By default, the standard output contains a line for each input *file* of the form:

```
lines words bytes file_name
```

The counts for lines, words, and bytes are integers separated by spaces. The ordering of the display of the number of lines, words, and/or bytes is the order in which the options were specified.
The *wc* utility exits 0 on success and >0 if an error occurs.

STANDARDS
The *wc* function conforms to POSIX 1003.2.