gawk — Command

Pattern-scanning and -processing language
gawk [*POSIX or GNU style options* **] -f** *program-file* **[--]** *file ...*
gawk [*POSIX or GNU style options* **] [--]** *program-text file ...*

gawk is the GNU Project's implementation of the AWK programming language. It conforms to the definition of the language in the POSIX Standard 1003.2 *Command Language and Utilities Standard*. This version in turn is based on the description in *The AWK Programming Language*, by Aho, Kernighan, and Weinberger, with the additional features defined in the System V Release 4 version of **awk**. **gawk** also provides some GNU-specific extensions.

The command line consists of options to **gawk** itself, the AWK program text (if not supplied via the options **-f** or **--file**), and values to be made available in the predefined AWK variables **ARGC** and **ARGV**.

Command-line Options

gawk options may be either the traditional POSIX one-letter options, or the GNU style long options. POSIX Standard-style options begin with a single '-', whereas GNU long options begin with "--". GNU-style long options are provided for both GNU-specific features and for POSIX mandated features. Other implementations of the AWK language are likely to only accept the traditional one-letter options.

Following the POSIX Standard, **gawk**-specific options are supplied via arguments to the **-W** option. Multiple **-W** options may be supplied, or multiple arguments may be supplied together if they are separated by commas, or enclosed in quotation marks and separated by white space. Case is ignored in arguments to the **-W** option. Each **-W** option has a corresponding GNU style long option, as detailed below.

gawk recognizes the following command-line options:

-F *fs*
--field-separator=*fs*
Use *fs* for the input field separator (the value of the predefined variable **FS**).

-v *variable***=***value*
--assign=*variable***=***value*
Assign *value* to *variable* before executing the program. *value* is available to the **BEGIN** block of an AWK program.

-f *program-file*
--file=*program-file*
Read the AWK program's source from file *program-file*, instead of from the first command-line argument. The **awk** command line can contain more than one **-f** or **--file** options.

-W compat
--compat
Run in compatibility mode. In compatibility mode, **gawk** behaves identically to UNIX **awk**; it recognizes none of the GNU-specific extensions are recognized. These extensions are described below.

-W copyleft
-W copyright
--copyleft
--copyright
Print the short version of the GNU copyright information message on the standard error.

-W help

-W usage
--help
--usage Print a relatively short summary of the available options on the standard error.

-W lint
--lint Provide warnings about constructs that are dubious or non-portable to other implementations of AWK.

-W posix
--posix This turns on compatibility mode, with the following additional restrictions:

- The '\x' escape sequences are not recognized.
- The synonym **func** for the keyword function is not recognized.
- The operators "**" and "**=" cannot be used in place of '^' and "^=".

-W source=*program-text*

--source=*program-text*
Use *program-text* as the AWK program's source code. This option allows the easy intermixing of library functions (used via the options **-f** and **--file**) with source code entered on the command line. It is intended primarily for medium to large AWK programs used in shell scripts. The **-W source=** form of this option uses the rest of the command line argument for *program-text*; no other options to **-W** will be recognized in the same argument.

-W version
--version
Print version information for this particular copy of **gawk** on the standard error. This is useful mainly for knowing if your copy of **gawk** is up to date with what the Free Software Foundation is distributing.

-- Signal the end of options. This is useful to allow further arguments to the AWK program itself to start with a '-'. This is mainly for consistency with the argument parsing convention used by most other POSIX Standard programs.

All other options are flagged as illegal and ignored.

AWK Program Execution

An AWK program consists of a sequence of pattern/action statements, plus optional function definitions:

```
pattern { action statements }
function name(parameter list) { statements }
```

gawk first reads the program source from the program file (or files) if specified, or from the first non-option argument on the command line. The option **-f** may be used multiple times on the command line. **gawk** reads the program text as if all the program-files had been concatenated. This is useful for building libraries of AWK functions, without having to include them in each new AWK program that uses them. To use a library function in a file from a program typed in on the command line, specify **/dev/tty** as one of the program files, type your program, and end it with a **<ctrl-D>**.

The environment variable **AWKPATH** specifies a search path to use when finding source files named with the option **-f**. If this variable does not exist, the default path is:

```
.:/usr/lib/awk:/usr/local/lib/awk
```

If a file name given to the **-f** option contains a '/' character, **gawk** does not perform a path search.

gawk executes AWK programs in the following order:

1. **gawk** compiles the program into an internal form.
2. All variable assignments specified via the **-v** option are performed.
3. **gawk** executes the code in the **BEGIN** block (or blocks), should there be any.
4. **gawk** then proceeds to read each file named in the **ARGV** array. If no files are named on the command line, **gawk** reads the standard input.

If a file name on the command line has the form *variable***=***value*, **gawk** treats it as a variable assignment, and assigns *value* to *variable*. (This happens after every **BEGIN** block has been run.) Command-line assignment of variables is most useful when you wish to assign values dynamically to the variables AWK uses to control how

LEXICON

input is broken into fields and records. It is also useful for controlling the state of program execution if multiple passes are needed over a single data file.

If the value of a particular element of **ARGV** is empty (""), **gawk** skips it.

For each line in the input, **gawk** tests to see if it matches any pattern in the AWK program. It tests the patterns in the order they occur in the program. For each pattern that the line matches, **awk** executes action associtaed with that pattern.

Finally, after all the input is exhausted, **gawk** executes the code in every **END** block.

Variables and Fields

AWK variables are dynamic: they come into existence when they are first used. Their values are floating-point numbers, strings, or both, depending upon how they are used. AWK also has one dimensional arrays: multiply dimensioned arrays can be simulated. Several pre-defined variables are set as a program runs; these are described as needed and summarized below.

Fields

As it reads a line of input, **gawk** splits that line into fields. The variable **FS** defines how fields are separated:

- If **FS** is a single character, fields are separated by that character.
- If **FS** is longer than one character, it must be a regular expression. In this case, the value of variable **IGNORECASE** (described below) also affects how fields are split. **FS** is a regular expression.
- In the special case that **FS** is a single space character, fields are separated by a number of space characters or tab characters.

If variable **FIELDWIDTHS** is set to a space-separated list of numbers, each field is expected to have a fixed width: **gawk** splits up the record using the specified widths, and ignores the value of **FS**. Assigning a new value to **FS** overrides the use of **FIELDWIDTHS**, and restores the default behavior.

Each field in the input line can be referenced by its position: **$1**, **$2**, and so on. $0 is the whole line.

The value of a field may be assigned to as well. Fields need not be referenced by constants. For example, the AWK expression

```
n = 5
print $n
```

prints the fifth field in the input line. The variable **NF** holds the total number of fields in the input line.

References to non-existent fields (i.e., fields after **$NF**) produce the null string. However, assigning to a nonexistent field (e.g., **$(NF+2) = 5**) increases the value of **NF**; creates any intervening fields, with the null string as the value of each; and causes the value of **$0** to be recomputed, with the fields being separated by the value of **OFS**.

Built-in Variables

The following variables are built into AWK:

ARGC The number of command-line arguments. Note that this does not include options to **gawk**, or the program source.

ARGIND
The index in **ARGV** of the file now being processed.

ARGV Array of command-line arguments. The array is indexed from through to **ARGC** minus one. Dynamically changing the contents of **ARGV** can control the files used for data.

CONVFMT
The conversion format for numbers — by default, “%.6g”.

ENVIRON
An array containing the values of the current environment. The array is indexed by the environment variables, each element being the value of that variable (e.g., **ENVIRON["HOME"]** might be **/u/arnold**). Changing this array does not affect the environment seen by programs which gawk spawns via redirection or the function **system()**. (This may change in a future version of **gawk**.)

ERRNO
If a system error occurs while performing redirection for **getline()**, during a read for **getline()**, or during a close, **ERRNO** contains a string describing the error.

FIELDWIDTHS
A white-space separated list of fieldwidths. When set, **gawk** parses the input into fields of fixed width, instead of using the value of the variable **FS** as the field separator. The fixed field-width facility is still experimental; expect the semantics to change as **gawk** evolves over time.

FILENAME
The name of the current input file. If no files are specified on the command line, the value of **FILENAME** is '-'. However, **FILENAME** is undefined within the **BEGIN** block.

FNR The number of the record within the current input file that is now being processed.

FS The input field separator. By default, this is a blank.

IGNORECASE
Tell **gawk**'s pattern-matching features to ignore the case when they compare text with a pattern. When **IGNORECASE** is set to a nonzero function, the following features of **gawk** are affected:

- Pattern-matching within rules
- Fieldsplitting with **FS**.
- Regular expression matching with '~' and "!~".
- The operation of the pre-defined **gawk** functions **gsub()**, **index()**, **match()**, **split()**, and **sub()**.

Thus, if **IGNORECASE** is not equal to zero, pattern

```
/aB/
```

matches all of the following:

```
ab
aB
Ab
AB
```

As with all AWK variables, the initial value of **IGNORECASE** is zero, so all regular expression operations are normally case-sensitive.

NF The number of fields in the current input record.

NR The total number of input records seen so far.

OFMT The output format for numbers — by default "%.6g".

OFS The output-field separator — by default a space character.

ORS The output-record separator — by default a newline.

RS The input record separator — by default a newline. **RS** is exceptional in that only the first character of its string value is used to separate records. (This will probably change in a future release of **gawk**.) If **RS** is set to the null string, then records are separated by blank lines. When **RS** is set to the null string, then the newline character always acts as a field separator, in addition to whatever value **FS** may have.

RSTART
The index of the first character matched by the **gawk** function **match()**: zero if no match.

RLENGTH
The length of the string matched by **match()**: -1 if no match.

SUBSEP
The character used to separate multiple subscripts in array elements — by default " 34".

Arrays

Arrays are subscripted with an expression between square brackets ('[' and ']'). If the expression is an expression

list (expr, expr ...)

LEXICON

then the array subscript is a string consisting of the concatenation of the (string) value of each expression, separated by the value of the variablen **SUBSEP**. This facility simulates multi-dimensional arrays. For example,

```
i = "A" ; j = "B" ; k = "C"
x[i, j, k] = "hello, world\n"
```

assigns the string

```
"hello, world\n"
```

to the element of the array x which is indexed by the string

```
"A\034B\034C".
```

All arrays in AWK are associative, i.e., indexed by string values.

The special operator in may be used in an **if** or **while** statement to see if an array has an index that consists of a particular value:

if (val in array)
print array[val]

If the array has multiple subscripts, use **(i, j)** in array.

You can also use the construct **in** within a **for** loop to iterate through all the elements of an array.

An element can be deleted from an array using the statement **delete**.

Variable Typing And Conversion

Variables and fields can be floating-point numbers, strings, or both. How the value of a variable is interpreted depends upon its context. If a variable or field is used in a numeric expression, **gawk** treats it as a number; if used as a string, **gawk** treats it as a string. To force a variable to be treated as a number, add zero to it; to force it to be treated as a string, concatenate it with the null string.

When a string must be converted to a number, the conversion is accomplished by the library function **atof()**. A number is converted to a string by using the value of **CONVFMT** as a format string for **sprintf()**, with the numeric value of the variable as the argument. However, even though all numbers in AWK are floating point, integral values are always converted as integers. Thus, given

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

the variable **b** has a value of 12, not 12.00.

gawk performs comparisons as follows:

- If two variables are numeric, they are compared numerically.
- If one value is numeric and the other has a string value that is a "numeric string," then comparisons are also done numerically.
- Otherwise, the numeric value is converted to a string and a string comparison is performed.

Two strings are compared, of course, as strings. According to the POSIX Standard, even if two strings are numeric strings, a numeric comparison is performed; however, this is clearly incorrect, and **gawk** does not do this.

Uninitialized variables have the numeric value zero and the string value "" (the null, or empty, string).

Patterns and Actions

AWK is a line-oriented language: the pattern comes first, and then the action. Action statements are enclosed in '{' and '}'. Either the pattern may be missing, or the action may be missing, but (of course) not both. If the pattern is missing, AWK executes the action for every line of input. A missing action is equivalent to

```
{ print }
```

which prints the entire line.

Comments begin with the character '#', and continue to the end of the line. Blank lines can be used to separate statements. Normally, a statement ends with a newline; however, this is not the case for lines ending in any of the following characters:

```
,     {     ?     :     &&    ||
```

Lines that end in one of the above characters have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a '\', in which case the newline will be ignored.

Multiple statements may be put on one line by separating them with a ';'. This applies to both the statements within the action part of a pattern/action pair (the usual case), and to the pattern/action statements themselves.

Patterns

AWK patterns may be one of the following:

```
BEGIN
END
/regular expression/
relational expression
pattern && pattern
pattern || pattern
pattern ? pattern : pattern
(pattern)
! pattern
pattern1, pattern2
```

BEGIN and **END** are two special patterns that are not tested against the input. The action parts of all **BEGIN** patterns are merged as if all the statements had been written in a single **BEGIN** block. They are executed before any of the input is read. Likewise, **gawk** merges all the **END** patterns and executes them when all the input is exhausted (or when an **exit** statement is executed). **BEGIN** and **END** patterns cannot be combined with other patterns in pattern expressions. **BEGIN** and **END** patterns must have action parts.

For

```
/regular expression/
```

patterns, the associated statement is executed for each input line that matches the regular expression. Regular expressions are the same as those described in the Lexicon entry for the shell **sh**, and are summarized below.

A relational expression may use any of the operators defined below in the section on actions. These generally test whether certain fields match certain regular expressions.

The operators **&&**, **||**, and **!** are logical AND, logical OR, and logical NOT, respectively, as in C. They do short-circuit evaluation, also as in C, and are used for combining more primitive pattern expressions. As in most languages, parentheses may be used to change the order of evaluation.

The operator **?:** is like the same operator in C. If the first pattern is true then the pattern used for testing is the second pattern, otherwise it is the third. Only one of the second and third patterns is evaluated.

The

pattern1, *pattern2*

form of an expression is called a "range pattern". It matches all input records starting with a line that matches *pattern1*, and continues until it reads a record that matches *pattern2*, inclusive. It does not combine with any other sort of pattern expression.

Regular Expressions

Regular expressions are the extended kind found in the shell **sh**. They are composed of characters, as follows:

c Match the non-meta-character *c*.

*****c* Match the literal character *c*.

. Match any character except newline.

^ Match the beginning of a line or a string.

$ Match the end of a line or a string.

[*abc...***]** Character class: Match any of the characters *abc...*.

LEXICON

[^*abc...***]**
Negated character class: Match any character except *abc...* and newline.

*r1***|***r2* Alternation: match either *r1* or *r2*.

r1r2 Concatenation: Match *r1*, then *r2*.

*r***+** Match one or more *r*'s.

*r** Match zero or more *r*'s.

*r***?** Match zero or one *r*'s.

(*r***)** Grouping: match *r*.

The escape sequences that are valid in string constants (see below) are also legal in regular expressions.

Actions

Action statements are enclosed in braces, '{' and '}'. Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and input/output statements available are patterned after those in C.

Operators

The following gives AWK's operators, in order of increasing precedence:

= += -=
*= /= %= ^= = (assignment)

Both absolute assignment (*var* **=** *value*) and operator-assignment (the other forms) are supported.

This has the form

expr1 **? expr2 :** *expr3*

If *expr1* is true, the value of the expression is *expr2*; otherwise it is *expr3*. Only one of *expr2* and *expr3* is evaluated.

| | — logical OR
&& — logical AND
~ — Regular expression match
!~ — Negated match
Do not use a constant regular expression (**/foo/**) on the left-hand side of a '~' or '!~'. Only use one on the right-hand side. The expression

/foo/ ~ *exp*

has the same meaning as:

(($0 ~ /foo/) ~ *exp***)**

This is usually not what was intended.

< >
<= >=
!=
== The regular relational operators.

<blank>
String concatenation.

+
- Addition and subtraction.

*
/
% Multiplication, division, and modulus.

+

-
! Unary plus, unary minus, and logical negation.

^ Exponentiation. The operator '**' may also be used, and '**=' for the assignment operator.

++
-- Increment and decrement, both prefix and suffix.

$ Field reference.

Control Statements

The control statements are as follows:

if (*condition***)** *statement* **[else** *statement* **]**
while (*condition***)** *statement*
do *statement* **while (***condition***)**
for (*expr1***;** *expr2***;** *expr3***)** *statement*
for (*var in array***)** *statement*
break
continue
delete array[*index***]**
exit [*expression*]
{ *statements* }

I/O Statements

AWK recognizes the following input/output statements:

close(*filename***)**
Close file or pipe.

getline Set **$0** from next input record. This statement also sets the built-in variables **NF**, **NR**, and **FNR**.

getline <*file*
Set **$0** from next record of file. This statement also sets the built-in variable **NF**.

getline *var*
Set *var* from next input record. This statment also sets the built-in variables **NF** and **FNR**.

getline *var* **<***file*
Set *var* from next record of *file*.

next Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, each **END** block is executed.

next *file*
Stop processing the current input file. The next input record read comes from the next input file. **FILENAME** is updated, **FNR** is reset to one, and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, every **END** is executed.

print Print the current record.

print *expr-list*
Print each expression in *expr-list*.

print *expr-list* **>***file*
Print expressions on file.

printf *fmt***,** *expr-list*
Format and print.

printf *fmt***,** *expr-list* **>***file*
Format and print into *file*.

system(*cmd-line***)**
Execute the command *cmd-line*, and return its exit status.

Other input/output redirections are also allowed. For **print** and **printf**, **>>***file* appends output onto *file*, whereas a '**|**' command writes onto a pipe. Likewise, command **|getline pipes into getline**. **getline returns zero when it reads EOF, and -1 if an error occurs.**

LEXICON

The printf Statement

The AWK statement **printf** and the function **sprintf()** (see below) accept the following conversion specification formats:

%c An ASCII character. If the argument used for **%c** is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

%d A decimal number (the integer part).

%i Just like **%d**.

%e A floating-point number of the form **[-]***d***.***dddddd***E[+-]***dd*.

%f A floating-point number of the form **[-]***ddd***.***dddddd*.

%g Use 'e' or 'f' conversion, whichever is shorter, with nonsignificant zeros suppressed.

%o An unsigned octal number (again, an integer).

%s A character string.

%x An unsigned hexadecimal number (an integer).

%X Like **%x**, but using "ABCDEF" instead of "abcdef".

%% A single '%' character; no argument is converted.

There are optional, additional parameters that may lie between the '%' and the control letter:

- The expression should be left-justified within its field.

width The field should be padded to this width. If the number has a leading zero, then the field will be padded with zeroes; otherwise, it is padded with blanks.

.*prec* A number that indicates the maximum width of the string or digit to the right of the decimal point.

The dynamic width and precision capabilities of the ANSI C **printf()** routines are supported. A '*' in place of either the width or precision specification causes AWK to take its value from the argument list to **printf** or **sprintf()**.

Special File Names

When doing I/O redirection from either **print** or **printf** into a file, or via **getline** from a file, **gawk** recognizes certain special file names internally. These file names allow access to open file descriptors inherited from **gawk**'s parent process (usually the shell). Other special file names provide access information about the running **gawk** process. The file names are as follows:

/dev/pid
Reading this file returns the identfier of the current process, in decimal, terminated with a newline.

/dev/ppid
Reading this file returns the identifier of the current's process's parent, in decimal, terminated with a newline.

/dev/pgrpid
Reading this file returns the current process's group identifier, in decimal, terminated with a newline.

/dev/user
Reading this file returns a single record terminated with a newline. The fields are separated with blanks. **$1** is the value of the system call **getuid()**; **$2** is the value of the system call geteuid() ; **$3** is the value of the system call **getgid()**; and **$4** is the value of the system call **getegid()**. If there are any additional fields, they are the group identifiers returned by **getgroups()**.

/dev/stdin
The standard input.

/dev/stdout
The standard output.

/dev/stderr
The standard error output.

/dev/fd/*n*
The file associated with the open-file descriptor *n*.

These are particularly useful for error messages. For example, these files let you use the statement

```
print "You blew it!" > "/dev/stderr"
```

where otherwise you would have had to say:

```
print "You blew it!" | "cat 1>&2"
```

These file names may also be used on the command line to name data files.

Numeric Functions

AWK contains the following pre-defined arithmetic functions:

atan2(*y***, *x*)**
Return the arctangent of *y*/*x*, in radians.

cos(*expr***)**
Returns the cosine, in radians.

exp(*expr***)**
The exponential function.

int(*expr***)**
Truncate to integer.

log(*expr***)**
The natural-logarithm function.

rand() Returns a random number between zero and one.

sin(*expr***)**
Return the sine in radians.

sqrt(*expr***)**
The square-root function.

srand(*expr***)**
Use *expr* as a new seed for the random number generator. If no *expr* is provided, the time of day will be used. The return value is the previous seed for the random number generator.

String Functions

AWK has the following pre-defined string functions:

gsub(*r***, *s*, *t*)**
For each substring matching the regular expression *r* in the string *t*, substitute the string *s* and return the number of substitutions. If *t* is not supplied, use **$0**.

index(*s***, *t*)**
Return the index of the string *t* in the string *s*, or zero if *t* is not present.

length(s)
Return the length of the string *s*, or the length of **$0** if *s* is not supplied.

match(*s***, *r*)**
Return the position in *s* where the regular expression *r* occurs, or zero if *r* is not present, and set the values of **RSTART** and **RLENGTH**.

split(*s***, *a*, *r*)**
Split the string *s* into the array *a* on the regular expression *r*, and return the number of fields. If *r* is omitted, use **FS** instead.

sprintf(*fmt***, *expr-list*)**
Print *expr-list* according to *fmt*, and return the resulting string.

sub(*r***, *s*, *t*)**
Just like **gsub()**, but only the first matching substring is replaced.

LEXICON

substr(*s***, *i*, *n*)**
Return the *n*-character substring of *s* starting at *i*. If *n* is omitted, the rest of *s* is used.

tolower(*str***)**
Return a copy of the string *str*, with all the upper-case characters in *str* translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

toupper(*str***)**
Return a copy of the string *str*, with all the lower-case characters in *str* translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

Time Functions

Because one of the primary uses of AWK programs is processing log files that contain time stamp information, **gawk** provides the following two functions for obtaining time stamps and formatting them.

systime()
Return the current time of day as the number of seconds since 00:00:00 hours on January 1, 1970 GMT.

strftime(*format***, *timestamp*)**
Format *timestamp* according to the specification within *format*. *timestamp* should be of the same form as returned by **systime()**. If *timestamp* is missing, the current time of day is used. See the Lexicon entry for **strftime()** for the format conversions that are guaranteed to be available.

String Constants

String constants in AWK are sequences of characters enclosed between quotation marks "". Within a string, the following escape sequences are recognized:

****	Literal backslash
\a	The BEL character
\b	Backspace
\f	Form-feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	vertical tab.
\x*XX*	Character with hexadecimal value *XX*
*****OOO*	Character represented by octal digits *OOO*
*****c*	The literal character *c*

The escape sequences may also be used within constant regular expressions (e.g., **/[\t\f\n\r\v]/** matches whitespace characters).

Functions

AWK defines a function as follows:

*function name***(***parameter list***) { *statements* }**

AWK executes a function when it is called from within the action part of a regular *pattern***/***action* statement. The parameters supplied in the function call are used to instantiate the formal parameters declared within the function. Arrays are passed by reference, other variables are passed by value.

Because functions were not originally part of the AWK language, the provision for local variables is rather clumsy: they are declared as extra parameters in the parameter list. The convention is to separate local variables from real parameters by extra spaces in the parameter list. For example:

```
function f(p, q, a, b) { # a & b are local
      .....
}

/abc/ { ...
; f(1, 2) ; ...
}
```

The left parenthesis in a function call is required to immediately follow the function name, without any intervening white space. This is to avoid a syntactic ambiguity with the concatenation operator. This restriction does not apply to the built-in functions listed above.

Functions may call each other and may be recursive. Function parameters used as local variables are initialized to the null string and the number zero upon function invocation.

The word **func** may be used in place of **function**.

Examples

Print and sort the login names of every user on your system:

```
BEGIN { FS = ":" }
{ print $1 | "sort" }
```

Count lines in a file:

```
{ nlines++ }
END { print nlines }
```

Precede each line by its number in the file:

```
{ print FNR, $0 }
```

Concatenate and line number (a variation on a theme):

```
{ print NR, $0 }
```

Compatibility

A primary goal for **gawk** is compatibility with the POSIX Standard, as well as with the latest version of UNIX **awk**. To this end, **gawk** incorporates the following user-visible features that are not described in the AWK book, but are part of **awk** in System V Release 4, and are in the POSIX Standard:

- The option **-v** for assigning variables before program execution starts is new. The book indicates that command line variable assignment happens when awk would otherwise open the argument as a file, which is after the **BEGIN** block is executed. However, in earlier implementations, when such an assignment appeared before any file names, the assignment would happen before the **BEGIN** block was run. Applications came to depend on this "feature." When **awk** was changed to match its documentation, this option was added to accomodate applications that depended upon the old behavior. (This feature was agreed upon by both the AT&T and GNU developers.)
- The option **-W** for implementation specific features is from the POSIX Standard.
- When processing arguments, gawk uses the special option "--" to signal the end of arguments, and warns about, but otherwise ignores, undefined options.
- The AWK book does not define the return value of **srand()**. The System V Release 4 version of UNIX awk (and the POSIX Standard standard) has it return the seed it was using, to allow keeping track of random number sequences. Therefore, **srand()** in **gawk** also returns its current seed.
- Other new features include the following: use of multiple **-f** options (from MKS **awk**); the **ENVIRON** array; the escape sequences **\a** and **\v** (done originally in **gawk** and fed back into AT&T's); the built-in functions **tolower()** and **toupper()** (from AT&T); and the ANSI-C conversion specifications in **printf** (done first in AT&T's version).

GNU Extensions

gawk has some extensions to POSIX Standard **awk**. They are described in this section. All the extensions described here can be disabled by invoking **gawk** with the command-line option **-W compat**. The following features of **gawk** are not available in POSIX Standard **awk**:

- The escape sequence **\x**.
- The functions **systime()** and **strftime()**.
- The special-file names available for I/O redirection.
- The variables **ARGIND** and **ERRNO** are not special.
- The variable **IGNORECASE** and its side-effects are not available.
- The variable **FIELDWIDTHS** and fixed-width field splitting.

LEXICON

- No path search is performed for files named via the option **-f**. Therefore, the environmental variable **AWKPATH** is not special.
- The use of next file to abandon processing of the current input file.

The AWK book does not define the return value of the function **close()**. **gawk**'s **close()** returns the value from **fclose()** or **pclose()** when closing a file or pipe, respectively. When **gawk** is invoked with the option **-W compat**, if the *fs* argument to option **-F** is 't', then **FS** will be set to the tab character. Because this is a rather ugly special case, it is not the default behavior. This behavior also does not occur if **-Wposix** has been specified.

Historical Features

There are two features of historical AWK implementations that **gawk** supports. First, it is possible to call the **length()** built-in function not only with no argument, but even without parentheses! Thus

```
a = length
```

is the same as either of

```
a = length()
a = length($0)
```

This feature is marked as "deprecated" in the POSIX Standard standard, and **gawk** will issue a warning about its use if option **-Wlint** is specified on the command line.

The other feature is the use of the **continue** statement outside the body of a **while**, **for**, or **do** loop. Traditional AWK implementations have treated such usage as equivalent to the next statement. **gawk** supports this usage if **-Wposix** has not been specified.

See Also

awk, commands, Programming COHERENT
Introduction to the awk Language, tutorial.
Aho, Alfred V.; Kernighan, Brian W.; Weinberger, Peter J.: *The AWK Programming Language.* Englewood Cliffs, NJ, Addison-Wesley, Inc., 1988 (ISBN 0-201-07981-X).
The GAWK Manual, ed 0.15. Boston, The Free Software Foundation, 1993.

Notes

The option **-F** option is not necessary given the command line variable assignment feature; it remains only for backwards compatibility.

If your system actually has support for **/dev/fd** and the associated **/dev/stdin**, **/dev/stdout**, and **/dev/stderr** files, you may get different output from **gawk** than you would get on a system without those files. When **gawk** interprets these files internally, it synchronizes output to the standard output with output to **/dev/stdout**, while on a system with those files, the output is actually to different open files. *Caveat utilitor.*

This man page documents gawk, version 2.15. Please note that with this version, **gawk** no longer recognizes the command-line options **-c**, **-V**, **-C**, **-a**, and **-e** that had been recognized by version 2.11.

The original version of UNIX awk was designed and implemented by Alfred Aho, Peter Weinberger, and Brian Kernighan of AT&T Bell Laboratories. Brian Kernighan continues to maintain and enhance it.

Paul Rubin and Jay Fenlason, of the Free Software Foundation, wrote **gawk** to be compatible with the original version of awk distributed in UNIX version 7. John Woods contributed a number of bug fixes. David Trueman, with contributions from Arnold Robbins, made **gawk** compatible with the new version of UNIX awk.

Brian Kernighan of AT&T Bell Laboratories provided valuable assistance during testing and debugging. The authors thank him.

Finally, please note that **gawk** and its associated documentation (including this manual page) is protected by the Free Software Foundation's "copyleft". For details on your rights and obligations, see the file **COPYING** in the source code for **gawk**, which is available through the Mark Williams BBS and other public-domain systems.

gcd() — Multiple-Precision Mathematics (libmp)

Set variable to greatest common divisor
#include <mprec.h>
void gcd(*a, b, c***)**
mint **a, *b, *c;**

gcd() sets *c* to the greatest common divisor of *a* and *b*.

See Also

libmp

gcvt() — General Function (libc)

Convert floating-point numbers to strings
char *
gcvt(*d***,** *prec***,** *buffer***)**
double *d***; int** *prec***; char** **buffer***;*

gcvt() converts floating-point number *d* into a NUL-terminated string. Its operation resembles that of **printf()**'s operator **%g**.

Argument *prec* gives the precision of the string i.e., the number of numerals to the right of the decimal point. Unlike its cousins **ecvt()** and **fcvt()**, **gcvt()** uses a buffer that is defined by the caller. *buffer* must point to a buffer large enough to hold the result; 64 characters will always be sufficient. When generating its output, **gcvt()** mimics **fcvt()** if possible. Otherwise, it mimics **ecvt()**.

gcvt returns *buffer*.

Example

For an example of this function, see the entry for **ecvt()**.

See Also

libc

gdbm.h — Header File

Header file for GDBM routines
#include <gdbm.h>

Header file **<gdbm.h>** declares functions, data types, and global variables used by the GDBM set of routines:

gdbm_close() . . . . . Close a GDBM data base
gdbm_delete(). . . . . Delete a record from a GDBM data base
gdbm_exists() . . . . . Check whether a GDBM data base contains a given record
gdbm_fetch() . . . . . Retrieve a record from a GDBM data base
gdbm_firstkey(). . . . . Return the first record from a GDBM data base
gdbm_nextkey() . . . . . Return the next record from a GDBM data base
gdbm_open(). . . . . Open a GDBM data base
gdbm_reorganize() . . . . . Reorganize a GDBM data base
gdbm_setopt(). . . . . Set GDBM options
gdbm_store() . . . . . Add records to a GDBM data base
gdbm_strerror(). . . . . Translate a GDBM error code into text
gdbm_sync(). . . . . Flush buffered GDBM data into its data base

This header file also defines two structures that the GDBM routines use. The first, **datum**, defines the structure of a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

The other structure, **GDBM_FILE**, holds the information that the GDBM routines use to access a GDBM data base:

```
typedef struct {int dummy[10];} *GDBM_FILE;
```

Error codes are written into global variable **gdbm_errno**, and are defined in header file **<gdbmerrno.h>**.

See Also

Notes

For a statement of copyright and permissions on this header file, see the Lexicon entry for **libgdbm**.

LEXICON

gdbm_close() — GDBM Function (libgdbm)

Close a GDBM data base
#include <gdbm.h>
void gdbm_close (*database***)**
GDBM_FILE *database***;**

Function **gdbm_close()** closes the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

See Also

Notes

If *database* were opened into mode **GDBM_FAST**, **gdbm_close()** automatically calls **gdbm_sync()** to flush buffered data into the data base before it closes *database*.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_delete() — GDBM Function

Delete a record from a GDBM data base
#include <gdbm.h>
int gdbm_delete (*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_delete()** deletes a the record with *key* from the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

If all goes well, **gdbm_delete()** returns zero. It returns -1 if *database* did not contain a record with *key*, or if *database* were opened into read-only mode.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_exists() — GDBM Function (libgdbm)

Check whether a GDBM data base contains a given record
#include <gdbm.h>
int gdbm_exists(*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_exists()** checks whether the GDBM data base to which *database* points contains a record with the key to which *key* points. *database* must have been returned by a call to **gdbm_open()**.

If *database* contains *key*, **gdbm_exists()** returns a value other than zero; otherwise, it returns zero.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_fetch() — GDBM Function (libgdbm)

Retrieve a record from a GDBM data base
#include <gdbm.h>
datum gdbm_fetch (*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_fetch()** retrieves the record with *key* from the database to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

gdbm_fetch() returns the record that contains *key*. If *database* does not contains such a record, **gdbm_fetch()**

returns a record whose field **dptr** is set to NULL.

gdbm_fetch() calls **malloc()** to allocate the memory to hold the data it retrieves from *database*. It is your responsibility to free this memory; to do so, call **free()** and pass it field **dptr** in the record that **gdbm_fetch()** returns.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_firstkey() — GDBM Function (libgdbm)

Return the first record from a GDBM data base
#include <gdbm.h>
datum gdbm_firstkey(*database***)**
GDBM_FILE *database***;**

Function **gdbm_firstkey()** returns the first record from the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

gdbm_firstkey() returns the first record within *database*. Note that that the first record is dictated by the algorithm that the GDBM routines use to hash the keys within the data base, and so may not be what you expect. If *database* is empty, **gdbm_firstkey()** returns a record whose field **dptr** is set to NULL.

gdbm_firstkey() calls **malloc()** to allocate the memory to hold the data it retrieves from *database*. It is your responsibility to free this memory; to do so, call **free()** and pass it field **dptr** in the record that **gdbm_firstkey()** returns.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_nextkey() — GDBM Function (libgdbm)

Return the next record from a GDBM data base
#include <gdbm.h>
datum gdbm_nextkey(*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_nextkey()** retrieves the next record from the data base to which *database* points. If *database* contains no more records, it returns a record whose field **dptr** is set to NULL.

database must have been returned by a call to **gdbm_open()**. The call to **gdbm_nextkey()** must follow a call to **gdbm_firstkey()**.

Please note that **gdbm_nextkey()** returns records in the order dictated by the algorithm with which the GDBM routines hash the data base's keys. If called within a loop, it is guaranteed to retrieve every record within *database*, although the order in which the records are retrieved may not be what you expect.

gdbm_nextkey() calls **malloc()** to allocate the memory to hold the data it retrieves from *database*. It is your responsibility to free this memory; to do so, call **free()** and pass it field **dptr** in the record that **gdbm_nextkey()** returns.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_open() — GDBM Function (libgdbm)

Open a GDBM data base
#include <gdbm.h>
GDBM_FILE gdbm_open(*database***,** *block_size***,** *read_write***,** *mode***,** *bailout***)**
char **database***;*
int *block_size*, *read_write*, *mode*;
void (*bailout***)();**

Function **gdbm_open()** opens a GDBM data base. It takes the following parameters:

database
: This gives the complete name of the data base. Please note that a data base actually consists of two files: one, called *database*.**dir**, holds the hashed index; the other, called *database*.**pag**, holds the data. The GDBM routines manage the manipulation of these files; you need not worry about them yourself. (For more details on how GDBM works, see the Lexicon entry for **libgdbm**.)

block_size
: This gives the size of a single transfer from disk to memory. **gdbm_open()** ignores this parameter is unless *database* is new. The minimum size is 512. If you set *block_size* to less than 512, the GDBM routines use a block size of **BSIZE**. (This constant gives the size of a block under COHERENT; it is set in header file **<sys/buf.h>**.

read_write
: This parameter indicates whether you are opening the data base into read mode or write mode. If a process opens *database* only to read records within it, it is called a "reader". If, however, a process can also add records to *database*, remove record from it, or modify records within it, it is called a "writer". *database* can be opened by multiple readers simultaneously, or by a single writer; it cannot be opened by multiple writers simultaneously, or by a reader and a writer simultaneously. This rule prevents a writer from modifying a data base while it is being read, and so confusing the readers; and to prevent multiple writers from "clobbering" each other's changes.

  read_write can be one of the following values:

  GDBM_READER
  : The process opening *database* is a reader.

  GDBM_WRITER
  : The process opening *database* is a writer.

  GDBM_WRCREAT
  : The process opening *database* is a writer; if the data base *database* does not exist, create it.

  GDBM_NEWDB
  : The process opening *database* is a writer; create *database* as a new data base, regardless of whether it already exists.

  GDBM_FAST
  : If this constant is OR'd onto **GDBM_WRITER**, **GDBM_WRCREAT**, or **GDBM_NEWDB**, the GDBM routines write the data base without disk-file syncronization. This speeds writing to the data base; however, if the writer dies unexpectedly, some data may be lost. To flush buffered data to disk, call function **gdbm_sync()**.

mode This is a bitwise OR of the modes into which *database* is created. For a list of the flags that can be incorporated into this argument, see the Lexicon entry **stat.h**. **gdbm_open()** ignores this argument unless *read_write* is set to **GDBM_WRCREAT** or **GDBM_NEWDB**.

bailout This points to the function that **gdbm_open()** calls should a fatal error occur. This function must take only one argument, a string that holds an error message. If you set *bailout* to NULL, the GDBM routines use a default function.

If all goes well, **gdbm_open()** returns a pointer to a record of type **GDBM_FILE**. All other GDBM functions need this record to manipulate the data base in *database*. If an error occurs, **gdbm_open()** returns NULL and sets global variable **gdbm_errno** and **errno** to appropriate values. For information on interpreting the contents of **errno**, see the Lexicon entry for **errno.h**; for information on interpreting the contents of **gdbm_errno**, see the entry for **gdbmerrno.h**.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_reorganize() — GDBM Function (libgdbm)

Reorganize a GDBM data base
#include <gdbm.h>
int gdbm_reorganize(*database***)**
GDBM_FILE *database***;**

Function **gdbm_reorganize()** reorganizes the contents of the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

When you delete a record from a GDBM data base, the GDBM routines to not close up the space within the data base, because doing so would make the GDBM routines unacceptably slow. Thus, if you delete many records from within a data base, its file will be much larger than it need be. In this case, you should call **gdbm_reorganize()** to close up the "holes" in it.

gdbm_reorganize() returns zero if all went well. If something went wrong, it returns a value other than zero and sets the global variables **errno** and **gdbm_errno** to appropriate values. (For information on how to interpret the contents of these variables, see the Lexicon entries for **errno.h** and **gdbmerrno.h**).

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_setopt() — GDBM Function (libgdbm)

Set GDBM options
#include <gdbm.h>
int gdbm_setopt(*database***,** *option***,** *value***,** *size***)**
GDBM_FILE *database***;**
int *option***, ****value***,** *size***;**

Function **gdbm_setopt()** sets an option on an open GDBM data base. You should call **gdbm_setopt()** after you call **gdbm_open()**, but before you read the data base or write to it.

database points to the data base being manipulated; it must have been returned by a call to **gdbm_open()**.

value is the value to which *option* is being set. It is specified as a pointer to an integer.

option specifies the option to set, as follows:

GDBM_CACHESIZE
: Set the size of the internal bucket cache. This option may only be set once on each data base. Upon the first access to the data base, the GDBM routines by default set the cache size to 100. Set *value* to the size of the cache.

GDBM_FASTMODE
: Turn on or turn off fast mode of access. If fast mode is turned on, the GDBM routines do not synchronize disk updates with changes to the data base. This speeds modifications to the data base, but runs the risk of losing data should the "writer" process die unexpectedly. Set *value* to **TRUE** or **FALSE**.

size gives the size of the data to which *value* points.

For example, the following call sets a data base to use a cache of ten:

```
int value = 10;
ret = gdbm_setopt( dbf, GDBM_CACHESIZE, &value, sizeof(int));
```

If all goes well, **gdbm_setopt()** returns zero. If something goes wrong, it returns -1 and sets global variables **errno** and **gdbm_errno** to appropriate values. For information on how to interpret the contents of these variables, see the Lexicon entries **errno.h** and **gdbmerrno.h**.

See Also

Notes

The use of variables *value* and *size* may seem overly complex; however, this will permit the GDBM routines to recognize a larger range of options in the future.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_store() — GDBM Function (libgdbm)

Add records to a GDBM data base
#include <gdbm.h>
int gdbm_store (*database***,** *key***,** *content***,** *flag***)**
GDBM_FILE *database***;**
datum *key***,** *content***;**
int *flag***;**

Function **gdbm_store()** writes data into a GDBM data base.

database points to the data base into which data are written. It must have been returned by a call to **gdbm_open()**.

key gives the key for the record being written. *content* gives the data to be associated with *key*.

flag indicates how data should be written; it can be either of the following:

GDBM_INSERT
Insert only. If *database* already contains a record with *key*, generate an error.

GDBM_REPLACE
Update. If *database* already contains a record with *key*, replace it with with *contents*.

If all goes well, **gdbm_store()** returns zero. If *database* was opened into read-only mode, it returns -1. If *flag* is set to **GDBM_INSERT** and *database* already contains *key*, it returns one.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_strerror() — GDBM Function (libgdbm)

Translate a GDBM error code into text
#include <gdbm.h>
#include <gdbmerror.h>
char *gdbm_strerror(*errno***)**
gdbm_error *errno***;**

Function **gdbm_strerror()** converts a GDBM error code into an error message that can be read by a human being.

errno is the error code. This usually is the global variable **gdbm_errno**, which a GDBM routine sets should an error occur while manipulating a GDBM data base.

If an error occurs, **gdbm_strerror()** returns NULL. Otherwise, it returns a pointer to the string that holds the message.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_sync() — GDBM Function (libgdbm)

Flush buffered GDBM data into its data base
#include <gdbm.h>
void gdbm_sync(*database***)**
GDBM_FILE *database***;**

Function **gdbm_sync()** flushes buffered data into its data base. It is the GDBM analogue of the system call **sync()**. You should call this function periodically if you are writing data into a data base that had been opened with flag **GDBM_FAST**.

database points to the data base being manipulated. It must have been returned by a call to **gdbm_open()**.

gdbm_sync() does not return until all the buffers are flushed onto disk. **gdbm_close()** automatically calls **gdbm_sync()** to flush data-base buffers before it closes a GDBM data base.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbmerrno.h — Header File

Define error messages used by GDBM routines
#include <gdbmerrno.h>

Header file **<gdbmerrno.h>** defines the error codes that the GDBM routines can write into global variable **gdbm_errno**, as follows:

GDBM_NO_ERROR
: All is well.

GDBM_MALLOC_ERROR
: The GDBM routines call **malloc()** to allocate memory for each record that they retrieve from a data base. This message indicates that a call to **malloc()** failed.

GDBM_BLOCK_SIZE_ERROR
: You tried to set an illegal block size when you created a new data base.

GDBM_FILE_OPEN_ERROR
: A data-base file could not be opened, for whatever reason.

GDBM_FILE_WRITE_ERROR
: A process could not write into a data-base file. This probably indicates a problem with permissions.

GDBM_FILE_SEEK_ERROR
: A GDBM routine could not move a data-base file's seek pointer to a place where the data base's hash table indicates a given record was stored. The data base may well be corrupt; check this error seriously.

GDBM_FILE_READ_ERROR
: A process could not read a data-base file. This probably indicates a problem with permissions.

GDBM_BAD_MAGIC_NUMBER
: When the GDBM function **gdbm_open()** create a new data base, it stamps the file with a "magic number," which indicates that that file is, in fact, a GDBM data base. This error indicates that the file you're attempting to read is not a GDBM a data base.

GDBM_EMPTY_DATABASE
: The GDBM data base contains no data.

GDBM_CANT_BE_READER
: You failed in an attempt to open a GDBM data base into read mode. The data base may have already been opened into write mode.

GDBM_CANT_BE_WRITER
: You failed in an attempt to open a GDBM data base into write mode. The data base may have already been opened into write mode by another process.

GDBM_READER_CANT_DELETE
: You opened a GDBM data base into read mode, but then attempted to delete a record. This is illegal.

GDBM_READER_CANT_STORE
: You opened a GDBM data base into read mode, but then attempted to write a record into it. This is illegal.

GDBM_READER_CANT_REORGANIZE
: You opened a GDBM data base into read mode, but then attempted to reorganize it. This is illegal.

GDBM_UNKNOWN_UPDATE
: You attempted to update a record within a data base, but the data base does not contain a record with the given key.

GDBM_ITEM_NOT_FOUND
: You attempted to read a record from a data base, but the data base does not contain a record with the given key.

GDBM_REORGANIZE_FAILED
An attempted reorganization of a file failed. The data base may be corrupt.
GDBM_CANNOT_REPLACE
You attempted to write a new record into a data base, but the data base already contains a record with the given key.
GDBM_ILLEGAL_DATA
You attempted to write a record into a data base, but the record contains illegal data (e.g., the field **dptr** is NULL).
GDBM_OPT_ALREADY_SET
You called **gdbm_setopt()** to set an option on a data base, but that option is already set.
GDBM_OPT_ILLEGAL
You called **gdbm_setopt()** to set an option on a data base, but the requested option is illegal or unrecognized.

Function **gdbm_strerror()** translates a GDBM error code into a string that you can display.

See Also

Notes

For a statement of copyright and permissions on this header file, see the Lexicon entry for **libgdbm**.

getc() — STDIO Function (libc)

Read character from file stream
#include <stdio.h>
int getc(*fp***)**
FILE **fp***;*

getc() is a function that reads a character from the file stream *fp*, and returns an **int**.

Example

The following example creates a simple copy utility. It opens the first file named on the command line and copies its contents into the second file named on the command line.

```
#include <stdio.h>

void fatal(string)
char *string;
{
      printf("%s\n", string);
      exit (1);
}

main(argc, argv)
int argc; char *argv[];
{
      int foo;
      FILE *source, *dest;

      if (--argc != 2)
            fatal("Usage: copy [source][destination]");

      if ((source = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open source file");
      if ((dest = fopen(argv[2], "w")) == NULL)
            fatal("Cannot open destination file");

      while ((foo = getc(source)) != EOF)
            putc(foo, dest);
}
```

See Also

fgetc(), getchar(), libc, putc()
ANSI Standard, §7.9.7.5
POSIX Standard, §8.1

Diagnostics

getc() returns EOF at end of file or on read fatal.

Notes

Because **getc()** is a macro, arguments with side effects probably will not work as expected. Also, because **getc()** is a complex macro, its use in expressions of too great a complexity may cause unforeseen difficulties. Use of the function **fgetc()** may avoid this.

getchar() — STDIO Function (libc)

Read character from standard input
#include <stdio.h>
int getchar()

getchar() reads a character from the standard input. It is equivalent to **getc(stdin)**.

Example

The following example gets one or more characters from the keyboard, and echoes them on the screen.

```
#include <stdio.h>

main()
{
      int foo;
      while ((foo = getchar()) != EOF)
            putchar(foo);
}
```

See Also

getc(), libc, putchar()
ANSI Standard, §7.9.7.6
POSIX Standard, §8.1

Diagnostics

getchar() returns **EOF** at end of file or on read error.

If you wish to receive characters from the keyboard immediately, without waiting for the enter key, see the example in the entry for **pipe()**.

getcwd() — General Function (libc)

Get current working directory name
#include <unistd.h>
char ***getcwd(***_buffer_**,** _size_**)**
char **_buffer_**;**
int _size_**;**

The current working directory is the directory from which file-name searches commence when a path name does not begin with '/'. **getcwd()** returns the name of the current working directory. It is useful for processes like spoolers and daemons, which must generate full path names for files.

If *buffer* is not NULL, **getcwd()** writes the path of the current working directory into it. The expected path name must not be longer than two characters less than *size*. In this case, **getcwd()** returns *buffer*.

If *buffer* is NULL, **getcwd() malloc()**'s *size* bytes. **getcwd()** returns a pointer to this block of memory. You can **free()** it later.

If you do not have permission to search all levels of the directory hierarchy above the current directory, **getcwd()** cannot obtain the directory name for you.

See Also

chdir(), libc, pwd, unistd.h
POSIX Standard §5.2.2

Diagnostics

getcwd() returns NULL and sets **errno** to an appropriate value if an error occurs. Possible errors include the following:

EPERM Could not read one of the parent directories.

EINVAL *size* is zero.

ENOMEM Memory could not be **malloc()**'d for the buffer.

ERANGE The path name is too long to fit into *size* minus two bytes.

Notes

If **getcwd()** fails, the working directory cannot be restored to its initial value.

getdents() — System Call (libc)

Read directory entries
#include <dirent.h>
int getdents (*fd***,** *buffer***,** *num***)**
int *fd***;**
char **buffer***;*
unsigned *num***;**

The COHERENT system call **getdents()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It reads an entry from a directory file and writes it into a structure of type **dirent**.

fd is the file descriptor for the directory file; it must be a file descriptor opened by a call to **open()** or **dup()**. *buffer* points to the area where **getdents()** writes its output. *num* gives the size of the area pointed to by *buffer*; **getdents()** returns no more than *num* bytes of information.

getdents() writes its output into a structure of type **dirent**, which is defined in the header file **dirent.h**. It has the following structure:

```
struct dirent {
      long d_ino;
      long d_off;
      unsigned short d_reclen;
      char d_name[1];
};
```

Field **d_name** is a NUL-terminated string of indefinite length. Because this structure does not have a fixed size, you must tell **getdents()** the maximum number of bytes it can output.

getdents() automatically increments the offset pointer associated with *fd* to point to the next entry within the directory file. This lets you within a loop to read the entire contents of a directory file.

If all goes well, **getdents()** returns the number of bytes it wrote into *buffer*. It returns zero if it has reached the end of the directory file. If something went wrong (for example, you tried to use it to read a file other than a directory file), it returns -1 and sets **errno** to an appropriate value.

See Also

dirent.h, closedir(), libc, opendir(), readdir(), rewinddir(), telldir()

Notes

This system call is designed to support directory-access library routines. It should not be called by user programs.

The COHERENT implementation of **getdents()** was written by D. Gwynn.

getdtablesize() — Sockets Function (libsocket)

Get the number of files a process can open
int getdtablesize()

Function **getdtablesize()** returns the number of file descriptors (and hence, the number of files) that a process can have open at any one time. It is meant to be an operating-system independent means of determining this value; under COHERENT, it returns the value of the manifest constant **OPEN_MAX**.

See Also

libsocket

getegid() — System Call (libc)

Get effective group identifier
#include <unistd.h>
getegid()

Every process has two different versions of its *group identifier*, called the *real* group identifier and the *effective* group identifier. The group identifiers determine eligibility to access files and use system privileges. Normally, these two identifiers are identical. However, for a *set group identifier* load module (see **exec**), the real group identifier is that of the group's current group, whereas the effective group identifier is that of the load module owner. This distinction allows system programs to use files which are protected from groups that invoke the program.

getegid() returns the effective group identifier.

See Also

access, exec, geteuid(), getgid(), getuid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.1

getenv() — General Function (libc)

Read environmental variable
#include <stdlib.h>
char ***getenv(***VARIABLE**) char** **VARIABLE**;**

A program may read variables from its *environment*. This allows the program to accept information that is specific to it. The environment consists of an array of strings, each having the form *VARIABLE*=*VALUE*. When called with the string *VARIABLE*, **getenv()** reads the environment, and returns a pointer to the string *VALUE*.

Example

This example prints the environmental variable **PATH**.

```
#include <stdio.h>
#include <stdlib.h>

main()
{
      char *env;
      extern char *getenv();

      if ((env = getenv("PATH")) == NULL) {
            printf("Sorry, cannot find PATH\n");
            exit(1);
      }
      printf("PATH = %s\n", env);
}
```

See Also

environmental variables, envp, exec, libc, putenv(), sh, stdlib.h
ANSI Standard, §7.10.4.4
POSIX Standard, §4.6.1

Diagnostics

When *VARIABLE* is not found or has no value, **getenv()** returns NULL.

geteuid() — System Call (libc)

Get effective user identifier
#include <unistd.h>
geteuid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

geteuid() returns the effective user identifier

Example

For an example of this call, see the entry for **getpwent()**.

See Also

access(), exec, getegid(), getgid(), getuid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.1

getgid() — System Call (libc)

Get real group identifier
#include <unistd.h>
getgid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

getgid() returns the real group id.

See Also

access(), exec, getegid(), geteuid(), getuid(), libc, login, setuid(), unistd.h
POSIX Standard §4.2.1

getgrent() — General Function (libc)

Get group file information
#include <grp.h>
struct group *getgrent();

getgrent() returns the next entry from file **/etc/group**. It returns NULL if an error occurs or if the end of file is encountered.

Files

/etc/group
<grp.h>

See Also

group, initgroups(), libc

Notes

All structures and information returned are in a static area internal to **getgrent()**. Therefore, information from a previous call is overwritten by each subsequent call.

getgrgid() — General Function (libc)

Get group file information, by group id
#include <grp.h>
struct group *getgrgid(*gid***);**
int *gid***;**

getgrgid() searches file **/etc/group** for the first entry with a numerical group id of *gid*. It returns a pointer to the entry if found; it returns NULL if an error occurs or if the end of file is encountered.

Files

/etc/group
<grp.h>

See Also

group, libc
POSIX Standard, §9.2.1

Notes

All structures and information returned are in a static area internal to **getgrgid()**. Therefore, information from a previous call is overwritten by each subsequent call.

getgrnam() — General Function (libc)

Get group file information, by group name
#include <grp.h>
struct group ***getgrnam(***gname***);**
char **gname***;*

getgrnam() searches file **/etc/group** for the first entry with a group name of *gname*. It returns a pointer to the entry for *gname* if it is found; it returns NULL for any error or if the end of the file is encountered.

Files

/etc/group
<grp.h>

See Also

group, libc
POSIX Standard, §9.2.1

Notes

All structures and information returned are in a static area internal to **getgrnam()**. Therefore, information from a previous call is overwritten by each subsequent call.

getgroups() — System Call (libc)

Read the supplemental group-access list
#include <unistd.h>
int getgroups(*gidsetsize***,** *grouplist***)**
int *gidsetsize***; gid_t** ***grouplist***;**

The "supplemental group-access list" is the list of group identifiers that are used in addition to the effective group identifier when determining the level of access that a process has to a file. **getgroups()** reads the identifiers from the current process's supplemental group-access list, and writes them into the array to which *grouplist* points.

grouplist has *gidsetsize* entries, and must be large enough to contain every entry from the list. The list cannot have more than **NGROUPS_MAX** entries. If *gidsetsize* equals zero, **getgroups()** returns the number of groups to which the calling process belongs without modifying the array to which *grouplist* points.

If all goes well, **getgroups()** returns the number of supplementary-group identifiers set for the calling process. It fails and returns -1 if *gidsetsize* is greater than zero but less than the number of supplementary-group identifiers set for the calling process, or if *grouplist* points to an illegal address. In the former instance, it sets **errno** to **EINVAL**; in the latter, it sets **errno** to **EFAULT**.

See Also

libc, setgroups(), unistd.h
POSIX Standard, §4.2.3

gethostbyaddr() — Sockets Function (libsocket)

Retrieve host information by address
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
struct hostent ***gethostbyaddr(***addr***,** *len***,** *type***)**
char ***host***;**
int *len***,** *type***;**

Function **gethostbyaddr()** interrogates file **/etc/hosts** and returns information about a given host on a network.

addr gives the address at which the host's Internet address resides in memory. *length* gives the number of characters in its name. *type* gives the type of address this is. If it is anything other than type **AF_INET**, **gethostbyaddr()** returns NULL.

If it could find information about the host in question, **gethostbyaddr()** returns the address in a instance of structure **hostent**, which is defined in header file **<netdb.h>**. If it could not, it returns **NULL**.

See Also

endhostent(), gethostbyname(), libsocket, sethostent()

Page 2

gethostbyname() — Sockets Function (libsocket)

Retrieve a host IP address by name

#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
struct hostent ***gethostbyname(***host***)**
char **host***;*

Function **gethostbyname()** interrogates file **/etc/hosts** for information about a host on a network. *host* gives the address where the name of the host resides in memory.

If it could find the address of *host*, **gethostbyname()** returns the address in a instance of structure **hostent**, which is defined in header file **<netdb.h>**. If it could not, or if *host* points to a spurious host name, it returns **NULL**.

See Also

endhostent(), gethostbyaddr(), libsocket, sethostent()

gethostname() — Sockets Function (libsocket)

Get the name of the local host

#include <sys/utsname.h>
int gethostname (*name***,** *length***)**
char **name***;*
int *length***;**

Function **gethostname()** reads the name of your local host.

name points to the chunk of memory into which **gethostname()** is to write the name of the local host. *length* gives the length of that chunk of memory.

gethostname() returns -1 if it could not read the name of the local host. Otherwise, it returns zero.

See Also

libsocket

Notes

name must point to enough memory to hold the name of your local host. If it does not, the behavior of this function is undefined (and probably unwelcome). *Caveat utilitor.*

getlogin() — General Function (libc)

Get login name
#include <unistd.h>
char *getlogin()

The name corresponding to the current user id is not always the same as the name under which a user logged into the COHERENT system. For example, the user may have issued a **su** command, or there may be several login names associated with a user id. **getlogin()** returns the login name found in the file **/etc/utmp.**

In cases where **getlogin()** fails to produce a result, **getpwuid()** (described in **getpwent()**) is normally used to determine the user name for a process.

Files

/etc/utmp login names

See Also

getpwent(), getuid(), libc, su, ttyname(), unistd.h, utmp.h, who
POSIX Standard, §4.2.4

Diagnostics

getlogin() returns NULL if the login name cannot be determined.

Notes

getlogin() stores the returned name in a static area that is destroyed by subsequent calls.

getmap — Command

De-archive Usenet map articles
/usr/lib/mail/getmap [-b *batchfile***] [-m** *mapdir***] [-n** *newsgroup***] [-u** *username***] [-w** *workdir***]**

The script **getmap** de-archives Usenet map articles. The articles must be in the form of a shell archive (or "shar" file). De-archived articles are copied into directory **/usr/spool/uumaps**.

getmap recognizes the following command-line arguments:

-b *batch*
De-archive *batch*, which is a shell archive of file names. If *batch* is '-', **getmap** reads the standard input. By default, **getmap** reads file **/usr/spool/uumaps/work/batch**.

-m *mapdir*
Copy articles into *mapdir*, instead of the default directory **/usr/spool/uumaps**.

-n *newsgroups*
Read articles from *newsgroup*.

-u *user* Mail errors to *user*. If *user* is '-', write errors to the standard output. By default, **getmap** mails errors to user **postmaster**.

-w *workdir*
Keep logs and batch files in *workdir*. By default, logs and batch files are kept in directory **/usr/spool/uumaps/work**.

See Also

commands, mail [overview]

getmsg() — System Call (libc)

Get the next message from a stream
#include <stropts.h>
int getmsg (*fd***,** *ctlptr***,** *dataptr***,** *flagsp***)**
int *fd***; struct strbuf ***ctlptr***,** *dataptr***; int ***flagsp***;**

getmsg() retrieves a message from a STREAMS file, and writes it into the buffer or buffers that you specify. The message must contain a data part, a control part, or both. **getmsg()** writes each part into its own buffer, as described below. The STREAMS module that generated the message defines the semantics of each part.

fd gives the file descriptor that references the stream whose message is being retrieved. *ctlptr* and *dataptr* each point to a structure of type **strbuf**, which contains the following members:

```
int maxlen; Maximum buffer length
int len;    Length of data
void  *buf; Pointer to buffer
```

ctlptr holds the message's control part, and *dataptr* its data part. **buf** points to the buffer into which the data or control information is to be written, and **maxlen** gives the maximum number of bytes the buffer can hold. **getmsg()** initializes **len** to the number of bytes of data or control information that it actually wrote into **buf**. It sets **len** to zero if the part in question has a length of zero; and it sets **len** to -1 if the message does not contain the part in question.

flagsp points to an integer that indicates the type of messages you can receive; this is discussed in detail below.

getmsg() has special behaviors, corresponding to the settings of *ctlptr* and *dataptr*, and of the structures to which they point:

- If either *ctlptr* or *dataptr* is NULL, or if **maxlen** equals -1, **getmsg()** does not process the corresponding part of the message. The message is left on the stream head's read queue.
- If *ctlptr* or *dataptr* is not NULL, but the message does not have a corresponding part, **getmsg()** sets **len** to -1.
- If **maxlen** equals zero and there is a zero-length control or data part, **getmsg()** removes the zero-length part from the read queue and sets **len** to zero If **maxlen** equals zero and the corresponding section contains more than zero bytes of information, **getmsg()** leaves that information on the read queue and sets **len** to zero.
- If **maxlen** is less than than the corresponding part of the message (the control part for *ctlptr* and the data part for *dataptr*), **getmsg()** retrieves **maxlen** bytes. It leaves the remainder of the message on the stream head's read queue and returns and a non-zero return value. Details are given below.

Flags

The following summarizes what flags are available, and what they mean.

- By default, **getmsg()** processes the first available message on the stream head's read queue. However, you can choose to retrieve only a high-priority message: just insert **RS_HIPRI** into the integer to which *flagsp* points. In this case, **getmsg()** processes the next message only if it is a high-priority message.
- If the integer to which *flagsp* points equals zero, **getmsg()** retrieves any message available on the stream head's read queue. In this case, if **getmsg()** retrieves a high-priority message, it sets to the integer to which *flagsp* points to **RS_HIPRI**; if the message does not have high priority, it sets that integer to zero.
- If flags **O_NDELAY** and **O_NONBLOCK** are not set as part of the global settings for *fd* (for details, see the Lexicon entry for **open()**), **getmsg()** blocks execution of your program until a message of the type specified by *flagsp* is available on the stream head's read queue. If either **O_NDELAY** or **O_NONBLOCK** has been set and a message of the specified type is not at the top of the queue, **getmsg()** fails and sets **errno** to **EAGAIN**.

If a hangup occurs on the stream from which messages are to be retrieved, **getmsg()** operates normally until the stream head's read queue is empty. Thereafter, it returns zero in the **len** fields of both *ctlptr* and *dataptr*.

Return Values

If all goes well, **getmsg()** returns a non-negative value. Zero indicates that a full message was read successfully.

MORECTL and **MOREDATA** indicate, respectively, that more control information or more data are awaiting retrieval; whereas **MORECTL | MOREDATA** indicates that more of both types information remain in the queue, to be retrieved by subsequent calls to **getmsg()**. However, if a message of higher priority has come into the stream head's read queue, the next call to **getmsg()** retrieves that higher-priority message and the information remaining from the partially retrieved message remains on the queue.

Errors

getmsg() fails if any of the following conditions are true:

- Either of the flags **O_NDELAY** or **O_NONBLOCK** is set but no message is available. **getmsg()** sets **errno** to **EAGAIN**.

- *fd* is not a valid file descriptor. **getmsg()** sets **errno** to **EBADF**.
- The next message in the read queue is not valie for **getmsg()** to read. **getmsg()** sets **errno** to **EBADMSG**.
- *ctlptr*, *dataptr*, or *flagsp* contains an illegal address. **getmsg()** sets **errno** to **EFAULT**.
- A signal was caught as **getmsg()** was executing. **getmsg()** sets **errno** to **EINTR**.
- *flagsp* holds an unrecognized value, or the stream referenced by *fd* is linked under a multiplexor. **getmsg()** sets **errno** to **EINVAL**.
- *fd* does not describe a stream. **getmsg()** sets **errno** to **ENOSTR**.

getmsg() also fails if the stream header receives a STREAMS error message before **getmsg()** tries to read it. **getmsg()** then returns the value in the STREAMS error message.

See Also

libc, putmsg(), STREAMS, stropts.h

getnetbyaddr() — Sockets Function (libsocket)

Get a network entry by address
#include <netdb.h>
struct netent ***getnetbyaddr(***network***,** *type***)**
long *network***; int** *type***;**

getnetbyaddr() fetches a network entry. It opens and searches file **/etc/network**, which describes all entities on your local network, for the entry with *address*. **/etc/networks** must have been opened by function **setnetent()**. *type* is the type of network; at present, **getnetbyaddr()** recognizes only type **AF_INET**.

getnetbyaddr() returns a pointer to an object of type **netend**, which is defined in header file **<netdb.h>**:

```
struct netent {
      char *n_name;     /* official name of net */
      char **n_aliases; /* alias list */
      int n_addrtype;   /* net number type */
      unsigned long n_net;    /* net number */
};
```

The following describes the members:

n_name
The official name of the network.

n_aliases
This points to a zero-terminated list of alternate names for the network.

n_addrtype
The type of the network number returned; currently, only type **AF_INET** is recognized.

n_net The network's number. Network numbers are returned in the machine's byte order.

getnetent() returns a pointer to the **netent** structure it built. It returns NULL if something went wrong or if it cannot find an entry with *address*.

See Also

endnetent(), getnetent(), getnetbyname(), libsocket, netdb.h, setnetent()

getnetbyname() — Sockets Function (libsocket)

Get a network entry by address
#include <netdb.h>
struct netent ***getnetbyname(***name***)**
char **name***;*

getnetbyname() fetches a network entry. It opens and searches file **/etc/networks**, which describes all entities on your local network, for the entry with *name*. **/etc/networks** must have been opened by function **setnetent()**.

getnetbyname() returns a pointer to an object of type **netend**, which is defined in header file **<netdb.h>**:

```
struct netent {
      char *n_name;     /* official name of net */
      char **n_aliases; /* alias list */
      int n_addrtype;   /* net number type */
      unsigned long n_net;    /* net number */
};
```

The following describes the members:

n_name
The official name of the network.

n_aliases
This points to a zero-terminated list of alternate names for the network.

n_addrtype
The type of the network number returned; currently, only type **AF_INET** is recognized.

n_net The network's number. Network numbers are returned in the machine's byte order.

getnetent() returns a pointer to the **netent** structure it built. It returns NULL if something went wrong or if it cannot find an entry with *address*.

See Also

endnetent(), getnetent(), getnetbyaddr(), libsocket, netdb.h, setnetent()

getnetent() — Sockets Function

Fetch a network entry
#include <netdb.h>
struct netent *getnetent();

getnetent() fetches a network entry. It reads the next line of file **/etc/network**, which describes all entities on your local network; if necessary, it opens this file.

getnetent() returns a pointer to an object of type **netend**, which is defined in header file **<netdb.h>**:

```
struct netent {
      char *n_name;     /* official name of net */
      char **n_aliases; /* alias list */
      int n_addrtype;   /* net number type */
      unsigned long n_net;    /* net number */
};
```

The following describes the members:

n_name
The official name of the network.

n_aliases
This points to a zero-terminated list of alternate names for the network.

n_addrtype
The type of the network number returned; currently, only type **AF_INET** is recognized.

n_net The network's number. Network numbers are returned in the machine's byte order.

getnetent() returns a pointer to the **netent** structure it built. It returns NULL if something went wrong or if it has reached the end of **/etc/networks**. You must call function **endnetent()** to close **/etc/networks**.

See Also

getnetbyaddr(), getnetbyname(), endnetent(), libsocket, netdb.h, setnetent()

getopt() — General Function (libc)

Get option letter from argv
#include <unistd.h>
int getopt(*argc***,** *argv***,** *optstring***)**
int *argc***;**
char ***argv***;**
char **optstring***;**
extern char **optarg***;**
extern int *optind***;**

getopt() returns the next option letter in **argv** that matches a letter in *optstring*. *optstring* is a string of recognized option letters. If a letter is followed by a colon, the option must have an argument, which may or may not be separated from it by white space. *optarg* points to the start of the option argument on return from **getopt()**.

getopt() writes into *optind* the **argv** index of the next argument to be processed. Because *optind* is external, it is normally initialized to one automatically before the first call to **getopt()**.

When all options have been processed (i.e., up to the first non-option argument), **getopt()** returns EOF. The special option "--" may be used to delimit the end of the options: **getopt()** returns EOF and skip "--".

See Also

libc

Diagnostics

getopt() prints an error message and returns a question mark when it encounters an option letter not included in *optstring*.

Notes

It is not obvious how '-' standing alone should be treated. This version treats it as a non-option argument, which is not always right.

Option arguments are allowed to begin with '-'. This is reasonable, but reduces the amount of error checking possible.

getopt() returns the parsed letter option in the external **int optopt**, which is overwritten by each call to **getopt()**. When **getopt()** returns '?', it can be helpful to examine the contents of this variable.

getopts — Command

Parse command-line options
getopts *optstring name [opt]*

The command **getopts** parses a command's options and check their legality. *optstring* must contain the options letters that the command using **getopts** will recognize. If a letter is followed by a colon ':', that option must have an argument that is separated from it by whitespace.

Each time it is invoked, **getopts** places the next option into the shell variable *name* and the index of the next argument to be processed into the shell variable **OPTIND**, which is initialized by default to one. When an option requires an argument, **getopts** copies it into the shell variable **OPTARG**. If **getopts** encounters an error, it initializes variable *name* to **?**.

When it encounters the end of the options, **getopts** exits with non-zero status. The special option "--" can be used to delineate the end of options.

Example

The following example processes a command that takes options **a**, **b**, and **o**; the last option requires an argument:

```
while getopts abo: c
do
      case $c in
            a|b)  FLAGS=$FLAGS$c;;
            o)    OARG=$OPTARG;;
            \?)   echo $USAGE 1>&2
                  exit 2;;
      esac
done
shift OPTIND-1
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o"xxx z yy" file
cmd -a -b -o"xxx z yy" -- file
cmd -ab -o"xxx z yy" file
cmd -ab -o"xxx z yy" -- file
```

Note that no space is required between **-o** and its argument.

See Also

commands, getopt(), ksh

getpass() — General Function (libc)

Get password with prompting
char ***getpass(***prompt**)**
char **prompt**;**

getpass() first prints the *prompt*. Then it disables echoing of input characters on the terminal device (either the file **/dev/tty** or the standard input), reads a password from it, and restores echoing on the terminal. It returns the given password.

Files

/dev/tty

See Also

crypt(), libc, login, passwd, su

Notes

The password is stored in a static location that is overwritten by successive calls. This static buffer is 50 characters long; any password longer than that can cause problems of one sort or another.

getpeername() — Sockets Function (libsocket)

Get name of connected peer
int getpeername(*socket***,** *name***,** *namelen***)**
int *socket***,** ***namelen***; struct sockaddr** ***name***;**

getpeername() returns the name of the "peer socket" that is connected to *socket*.

name points to the space into which **getpeername()** writes the name of the peer. *namelen* points to an integer that gives the amount of space to which **name** points. **getpeername()** re-initializes it to the length, in bytes, of the peer name that it has written at *name*.

If all goes well, **getpeername()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **getpeername()** sets **errno**:

EBADF *socket* is not a valid descriptor.

ENOTSOCK
socket describes a file, not a socket.

ENOTCONN
socket is not connected.

ENOBUFS
The system lack resources to perform the operation.

EFAULT
name contains an illegal address.

See Also

accept(), bind(), getsockname(), libsocket, socket()

getpgrp() — System Call (libc)

Get process-group identifier
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp();

getpgrp() returns the identifier of the calling process's process group. It always succeeds.

See Also

libc, types.h, unistd.h
POSIX Standard, §4.3.1

getpid() — System Call (libc)

Get process identifier
#include <unistd.h>
getpid()

Every process has a unique number, called its *process id*. **fork()** returns the process id of a created child process to the parent process.

getpid() returns the process id of the requesting process. Typically a process uses **getpid()** to pass its process id to another process which wants to send it a signal, or to generate a unique temporary file name.

Example

For an example of using this system call in a C program, see **signal()**.

See Also

fork(), getppid(), kill, libc, mktemp, unistd.h
POSIX Standard, §4.1.1

getppid() — System Call (libc)

Get process identifier of parent process
#include <unistd.h>
getppid()

Every process has a unique number, called its *process id*. **fork()** returns the process id of a created child process to the parent process.

getppid() returns the process id of the requesting process's parent process. In this way, a wayward child process can discover the identity of its parent.

See Also

fork(), getpid(), kill, libc, mktemp, unistd.h
POSIX Standard, §4.1.1

getprotobyname() — Sockets Function (libsocket)

Get protocol entry by protocol name
#include <netdb.h>
struct protoent ***getprotobyname(***name***);**
char **name***;*

getprotobyname() searches file **/etc/protocols**, which holds information about all protocols recognized by your local network, for the protocol named *name*. **/etc/protocols** has to have been opened by a call to **setprotoent()**.

getprotobyname() returns a pointer to an object of type **protoent**, which is defined in header file **netdb.h**:

```
struct protoent {
      char *p_name;     /* official name of protocol */
      char **p_aliases; /* alias list */
      int p_proto;      /* protocol number */
};
```

The following details each member:

p_name
The official name of the protocol.

p_aliases
This points to a zero-terminated list of alternate names for the protocol.

p_proto
The number of the protocol.

getprotobyname() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endprotoent(), getprotobynumber(), getprotoent(), libsocket, netdb.h, setprotoent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

At present, only the Internet protocols are understood.

getprotobynumber() — Sockets Function (libsocket)

Get protocol entry by protocol number
#include <netdb.h>
struct protoent ***getprotobynumber(***protocol**);**
int *protocol***;**

getprotobynumber() searches file **/etc/protocols**, which holds information about all protocols recognized by your local network, for the protocol identified by *number*. **/etc/protocols** has to have been opened by a call to **setprotoent()**.

getprotobynumber() returns a pointer to an object of type **protoent**, which is defined in header file **netdb.h**:

```
struct protoent {
      char *p_name;     /* official name of protocol */
      char **p_aliases; /* alias list */
      int p_proto;      /* protocol number */
};
```

The following details each member:

p_name
The official name of the protocol.

p_aliases
This points to a zero-terminated list of alternate names for the protocol.

p_proto
The number of the protocol.

getprotobynumber() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endprotoent(), getprotobyname(), getprotoent(), libsocket, netdb.h, setprotoent()

Notes

This function uses a static data space. If your application needs to save these data it must copy them before any subsequent calls overwrite them.

At present, only the Internet protocols are understood.

getprotoent() — Sockets Function (libsocket)

Get protocol entry
#include <netdb.h>
struct protoent *getprotoent();

getprotoent() reads the next entry from file **/etc/protocols**, which holds information about all protocols recognized by your local network. If necessary, it opens the file. It returns a pointer to an object of type **protoent**, which is defined in header file **<netdb.h>**:

```
struct protoent {
      char *p_name;     /* official name of protocol */
      char **p_aliases; /* alias list */
      int p_proto;      /* protocol number */
};
```

The following details each member:

p_name
The official name of the protocol.

p_aliases
This points to a zero-terminated list of alternate names for the protocol.

p_proto
The number of the protocol.

To close **/etc/protocols**, call function **endprotoent()**.

getprotoent() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endprotoent(), getprotobyname(), getprotobynumber(), libsocket, netdb.h, setprotoent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

At present, only the Internet protocols are understood.

getpw() — General Function (libc)

Search password file
getpw(*uid, line***)**
short *uid***;**
char **line***;*

getpw() searches the password file **/etc/passwd** for the first entry with numerical user id *uid*. If found, *line* receives the corresponding line from the password file.

Files

/etc/passwd

See Also

getpwent(), getuid(), libc, passwd

Diagnostics

getpw() returns a nonzero value on error.

getpwent() — General Function (libc)

Get password file information
#include <pwd.h>
struct passwd *getpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For a description of this structure, see **pwd.h**.

getpwent() returns the next entry from **/etc/passwd**.

Example

The following example demonstrates **getpwent()**, **getpwnam()**, **getpwuid()**, **setpwent()**, and **endpwent()**.

```
#include    <pwd.h>
#include    <stdio.h>
#include    <unistd.h>

main()
{
        int euid,                       /* Effective user id */
            ruid;                       /* Real user id */
        struct passwd *pstp;
        int i;

        /* Print out all users and home directories */
        i = 0;
        setpwent();                     /* Rewind file /etc/passwd */
        while ((pstp = getpwent()) != NULL)
             printf("%d: user name is %s, home directory is %s.\n",
                     ++i, pstp->pw_name, pstp->pw_dir);

        /* Find real user name.
         * NOTE: functions getpwuid and getpwnam rewind /etc/passwd
         * by calling setpwent().
         */
        ruid = getuid();
        if ((pstp = getpwuid(ruid)) == NULL) {
                /* If this message appears, something's wrong */
                fprintf(stderr, "Cannot find user with id number %d\n", pstp);
                exit (EXIT_FAILURE);
        } else
                printf("User's real name is %s\n", pstp->pw_name);

        /* Find the user id for superuser root */
        ((pstp = getpwnam("root")) == NULL) ?
                fprintf(stderr, "Do you have user root on your system?\n") :
                printf("root id is  %d\n", pstp->pw_uid);

      /* Check if the effective process id is the superuser id.
       *
       * NOTE: if you wish to see how to enable the root
       * privileges, you can run this command:
       * cc pwfun.c
       * su root chown root pwfun
       * su root chmod 4511 pwfun
       */

        euid = geteuid();               /* Get effective user id. */
        printf("Process ");
        (euid == pstp->pw_uid) ? printf("has ") : printf("doesn't have ");
        printf("the root privileges\n");
        exit(EXIT_SUCCESS);
}
```

Files

/etc/passwd
pwd.h

See Also

endpwent(), getpwnam(), getpwuid(), libc, pwd.h, setpwent()

Diagnostics

getpwent() returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwent()**. Therefore, information from a previous call is overwritten by each subsequent call.

If your system has implemented shadow passwords, you must use the shadow-password routine **getspent()** to retrieve records that contain passwords. For details, see this function's entry in the Lexicon.

getpwnam() — General Function (libc)

Get password file information, by name
#include <pwd.h>
struct passwd *getpwnam(*uname***)**
char ***uname*;**

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For a description of this structure, see **pwd.h**.

getpwnam() attempts to find the first entry with a name of *uname*.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

libc
POSIX Standard, §9.2.2

Diagnostics

getpwnam() returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwnam()**. Therefore, information from a previous call is overwritten by each subsequent call.

If your system has implemented shadow passwords, you must use the shadow-password routine **getspnam()** to retrieve records that contain passwords. For details, see this function's entry in the Lexicon.

getpwuid() — General Function (libc)

Get password file information, by id
#include <pwd.h>
struct passwd *getpwuid(*uid***)**
int *uid***;**

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For more information on this structure, see **pwd.h**.

getpwuid() attempts to find the first entry with a numerical user id of *uid*.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

libc
POSIX Standard, §9.2.2

Diagnostics

getpwuid() returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwuid()**. Therefore, information from a previous call is overwritten by each subsequent call.

gets() — STDIO Function (libc)

Read string from standard input
#include <stdio.h>
char ***gets(***buffer**)**
char **buffer***;*

gets() reads characters from the standard input into a buffer pointed at by *buffer*. It stops reading as soon as it detects a newline character or EOF. **gets()** discards the newline or EOF, appends NUL onto the string it has built, and returns another copy of *buffer*.

Example

The following example uses **gets()** to get a string from the console; the string is echoed twice to demonstrate what **gets()** returns.

```
#include <stdio.h>

main()
{
      char buffer[80];

      printf("Type something: ");

      fflush(stdout);
      printf("%s\n%s\n", gets(buffer), buffer);
}
```

See Also

buffer, fgets(), getc(), libc
ANSI Standard, §7.9.7.7
POSIX Standard, §8.1

Diagnostics

gets() returns NULL if an error occurs or if EOF is seen before any characters are read.

Notes

gets() stops reading the input string as soon as it detects a newline character. If a previous input routine left a newline character in the standard input buffer, **gets()** will read it and immediately stop accepting characters; to the user, it will appear as if **gets()** is not working at all.

For example, if **getchar()** is followed by **gets()**, the first character **gets()** will receive is the newline character left behind by **getchar()**. A simple statement will remedy this:

```
      while (getchar() != '\n')
            ;
```

This throws away the newline character left behind by **getchar()**; **gets()** will now work correctly.

getservbyname() — Sockets Function (libsocket)

Get a service entry by name
#include <netdb.h>
struct servent ***getservbyname(***name**,** *protocol);*
char ***name***,** ****protocol***;*

Function **getservbyname()** searches file **/etc/services**, which describes the services offered by TCP/IP on your local network, for the services offered by *name*. If *protocol* is not NULL, the search must also match the protocol it names. **/etc/services** must first have been opened by a call to **setservent()**.

getservbyname() returns a pointer to a structure of type **servent**, which is defined in header file **<netdb.h>**:

```
struct servent {
      char *s_name;     /* official name of service */
      char **s_aliases; /* alias list */
      int s_port; /* port service resides at */
      char *s_proto;    /* protocol to use */
};
```

The following details each member:

s_name
The official name of the service.

s_aliases
This points to a zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto
The name of the protocol to use when contacting the service.

To close **/etc/services**, call function **endservent()**.

getservbyname() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endservent(), getservent(), getservbyport(), libsocket, netdb.h, setservent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

getservbyport() — Sockets Function (libsocket)

Get a service entry by port number
#include <netdb.h>
struct servent ***getservbyport(***port**,** *protocol);*
int *port***; char** **protocol***;*

Function **getservbyport()** searches file **/etc/services**, which describes the services offered by TCP/IP on your local network, for the services offered by *port*. If *protocol* is not NULL, the search must also match the protocol it names. **/etc/services** must first have been opened by a call to **setservent()**.

getservbyport() returns a pointer to a structure of type **servent**, which is defined in header file **<netdb.h>**:

```
struct servent {
      char *s_name;     /* official name of service */
      char **s_aliases; /* alias list */
      int s_port; /* port service resides at */
      char *s_proto;    /* protocol to use */
};
```

The following details each member:

s_name
The official name of the service.

s_aliases
This points to a zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto
The name of the protocol to use when contacting the service.

To close **/etc/services**, call function **endservent()**.

getservbyport() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endservent(), getservbyname(), getservent(), libsocket, netdb.h, setservent(),

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

getservent() — Sockets Function (libsocket)

Get a service entry
#include <netdb.h>
struct servent *getservent();

Function **getservent()** reads the next entry from file **/etc/services**, which describes the services offered by TCP/IP on your local network. If necessary, it opens the file. It returns a pointer to a structure of type **servent**, which is defined in header file **<netdb.h>**:

```
struct servent {
      char *s_name;     /* official name of service */
      char **s_aliases; /* alias list */
      int s_port; /* port service resides at */
      char *s_proto;    /* protocol to use */
};
```

The following details each member:

s_name
The official name of the service.

s_aliases
This points to a zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto
The name of the protocol to use when contacting the service.

To close **/etc/services**, call function **endservent()**.

getservent() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endservent(), getservbyname(), getservbyport(), libsocket, netdb.h, setservent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

getsockname() — Sockets Function (libsocket)

Get the name of a socket
int getsockname(*socket***,** *name***,** *namelen***)**
int *socket*, ***namelen*; struct sockaddr ***name*;**

Function **getsockname()** returns the current name that is bound to *socket*.

socket is a file descriptor that identifies the socket in question. *name* points to a space into which **getsockname()** can write the socket name. *namelen* points to an integer that holds the number of bytes to which *name* points. **getsockname()** re-initializes this integer to the number of bytes in the name that it writes at address *name*.

If all goes well, **getsockname()** returns zero. If a problem occurs, it returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **getsockname()** sets **errno**:

EBADF *socket* is not a valid file descriptor.

ENOTSOCK
socket identifies a file, not a socket.

ENOBUFS
The system lacks sufficient resources to perform the operation.

EFAULT
name contains an illegal address.

See Also

bind(), libsocket, socket()

getsockopt() — Sockets Function (libsocket)

Read a socket option
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(*socket***,** *level***,** *option***,** *buffer***,** *length***)**
int *socket***,** *level***,** *option***;**
char **buffer***;*
int **length***;*

Function **getsockopt()** reads the options that are set on a socket.

socket gives the identifier of the socket, as returned by the function **socket()**.

level gives the level at which the options are set. To retrieve options set on the socket level, set *level* to **SOL_SOCKET** whereas to retrieve options set the TCP level, set *level* to the number of the TCP protocol.

option gives the number of the option whose setting interests you. For a list of options that are recognized at the socket level, see header file **<sys/socket.h>**. Options at other levels are set by their respective protocols.

buffer gives the address of the buffer into which the retrieve information will be written. *length* gives the address of an integer that gives the length of *buffer*, in bytes. If **getsockopt()** succeeds in retrieving the value of the requested option, it writes the option into *buffer* and re-initializes the **int** to which *length* points to give the length of the material it wrote into *buffer*.

If all goes well, **getsockopt()** returns zero. If something goes wrong, it returns -1 and sets **errno** to one of the following values:

EBADF *socket* does not identify a valid socket.

ENOMEM
The available user memory was insufficient to complete the operation.

ENOPROTOOPT
option gives an unknown option.

ENOTSOCK
socket identifies a file, not a socket.

See Also

libsocket, setsockopt()

getspent() — General Function (libc)

Get a shadow-password record
#include <shadow.h>
struct spwd *getspent()

The COHERENT system has four routines that search the file **/etc/shadow**, which contains the password of every user of the system. **getspent()** returns a record from this file. If a program has already read entries from **/etc/shadow**, **getspent()** returns the next entry; otherwise, it returns the first entry.

If an error occurs, **getspent()** returns NULL. Otherwise, it returns the address of an object with the structure **spwd** which is defined in header file **<shadow.h>**. For a description of this structure, see the Lexicon entry for **shadow.h**.

See Also

endspent(), libc, setspent(), shadow, shadow.h

Notes

All structures and information returned are in static areas internal to **getspent()**. Therefore, information from a previous call is overwritten by each subsequent call.

getspnam() — General Function (libc)

Get a shadow-password record, by user name
#include <shadow.h>
struct spwd ***getspnam(***uname**)**
char ***uname*;**

The COHERENT system has four routines that search the shadow-password file **/etc/shadow**, which contains the password of every user of your system. **getspnam()** returns the first entry for the user with a given login identifier. *uname* points to the login identifier of the user whose password you wish to retrieve.

If an error occurs, **getspnam()** returns NULL. Otherwise, it returns the address of an object with the structure **spwd**, which is defined in the header file **<shadow.h>**. For a description of this structure, see the Lexicon entry for **shadow.h**.

Files

/etc/shadow
shadow.h

See Also

getspent(), libc, shadow, shadow.h
POSIX Standard, §9.2.2

Notes

All structures and information returned are in static areas internal to **getspnam()**. Therefore, information from a previous call is overwritten by each subsequent call.

gettimeofday() — Sockets Function (libsocket)

Berkeley time function
#include <sys/time.h>
#include <time.h>
void gettimeofday (*timeval***,** *zone***)**
struct timeval ***timeval*;**
char ***zone*;**

Function **gettimeofday()** writes the current system time (i.e., the number of seconds since January 1, 1970 GMT) into *timeval***->tv_sec**. It also initializes field *timeval***->tv_usec** to zero.

gettimeofday() ignores argument *zone*. It returns nothing.

See Also

libsocket, time [overview]

getty — System Administration

Terminal initialization
/etc/getty *type*

The initialization process **init** invokes **getty** for each device indicated in the file **/etc/ttys**. **getty** tries to read a user name from the terminal which is the standard input, adapting its mode settings accordingly. Then **getty** invokes **login** with the name read. This process may set delays, mapping of upper to lower case, speed, and whether the terminal normally uses carriage return or linefeed to terminate input.

If the terminal baud rate is wrong, the login message printed by **getty** will appear garbled. If the specified *type* indicates variable speeds, as described below, hitting BREAK will try the next speed.

init passes the third character in a line of the file **/etc/ttys** as the *type* argument to **getty**. *type* conveys information about the terminal port. An upper-case letter in the range **A** to **S** specifies a hard-wired baud rate, as indicated in the header file **<sgtty.h>**. Other characters specify a range of speeds suitable to a dial-in modem. The following variable-speed settings are recognized:

0 Cycles through speeds 300, 1200, 150, and 110 baud, in that order. This is a good default setting for dial-in ports.

- Teletype model 33, fixed at 110 baud.

1 Teletype model 37, fixed at 150 baud.

2 9600 baud with delays (e.g., Tektronix 4104).

3 Cycles between 2400, 1200, and 300 baud. This is used with 2400-bps modems.

4 DECwriter (LA36) with delays.

5 Like **3**, but starts at 300 baud.

getty recognizes the following fixed-speed settings, for hard-wired terminals:

A 50 baud
B 75 baud
C 110 baud
D 134 baud
E 150 baud
F 200 baud
G 300 baud
H 600 baud
I 1200 baud
J 1800 baud
K 2000 baud
L 2400 baud
M 3600 baud
N 4800 baud
O 7200 baud
P 9600 baud
Q 19200 baud
R EXT
S EXT

Files

/etc/tty
<sgtty.h>

See Also

Administering COHERENT, init, ioctl(), login, sgtty.h, stty, ttys

getuid() — System Call (libc)

Get real user identifier
#include <unistd.h>
int getuid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec()**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

getuid() returns the real user id.

Example

For an example of this call, see the entry for **getpwent()**.

See Also

access(), exec, getegid(), geteuid(), getgid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.1

getutent() — General Function (libc)

Read an entry from a login logging file
#include <utmp.h>
struct utmp *getutent()

getutent() reads the next entry from the file that holds login information. If the file is not open, **getutent()** opens it. By default, **getutent()** reads file **/etc/utmp**. To change this, call **utmpname()**.

getutent() returns the address of the record it has read from the login file. This object has type **utmp**, which is defined in header file **<utmp.h>**; for a detailed description of this structure, see the Lexicon entry for **utmp.h**. **getutent()** returns NULL if it cannot open the login file, or when it attempts to read past the end of the file.

See Also

libc, utmp.h

Notes

getutent() writes its **utmp** record into a static portion of memory, which it overwrites the next time it is called. Therefore, if you wish to save **utmp** record, you must copy it into a portion of memory that you define before you again call **getutent()**.

getutid() — General Function (libc)

Find a record in login logging file by login identifier
#include <utmp.h>
struct utmp *getutid(*id***)**
struct utmp *id;

Function **getutid()** searches a login file for a record with a given type, or for a user with a given login identifier.

id is the address of an object type **utmp**, which is a structure whose fields describe a login event. (For a detailed description of this structure, see the Lexicon entry for **utmp.h**). Before you call **getutid()**, initialize *id*'s fields as follows:

- Set field *id***.ut_type** to the type of record you wish to retrieve. The type can be one of the following:

EMPTY	An empty entry
RUN_LVL	Run level
BOOT_TIME	Boot time
OLD_TIME	
NEW_TIME	
INIT_PROCESS	Process spawned by **init**
LOGIN_PROCESS	A **getty** waiting for a login
USER_PROCESS	A user process
DEAD_PROCESS	
ACCOUNTING	

- If you initialize field *id***.ut_type** to **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, initialize field *id***.ut_id** to the identifier of the user whose login event you are seeking. Note that this must be the identifier as set by **/etc/init**, *not* the login identifier that the user types to log into your system.

If you initialize field **ut_type** to **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, **getutid()** seeks the first record that matches both the type and the identifier that you set in **id**. If you initialize field **ut_type** to any other type, it seeks the first record that matches the type you requested.

If it finds a record that matches your request, **getutid()** copies it into a static portion of memory and returns the address of that memory. It returns NULL if it fails to find a record of the type you requested, or if it cannot open the login file.

By default, **getutid()** reads records from **/etc/utmp**. If you wish to read records from another file, call **utmpname()** before you call **getutid()**.

See Also

libc, utmp.h

getutline() — General Function (libc)

Find a record in login logging file by device
#include <utmp.h>
struct utmp ***getutline(***line***)**
struct utmp **line***;*

Function **getutline()** seeks a record for a login that occurred for a given line.

line points to an object of type **utmp**, which is a structure whose fields describe a login event. (For a detailed description of this structure, see the Lexicon entry for **utmp.h**.) Before you call **getutline()**, you must initialize field *line***.ut_line** to the name of the device that interests you.

If it finds a record that matches your request, **getutline()** copies it into a static portion of memory and returns the address of that memory. It returns NULL if it fails to find a record for the device you named, or if it cannot open the login file.

By default, **getutline()** reads records from **/etc/utmp**. If you wish to read records from another file, call **utmpname()** before you call **getutid()**.

See Also

libc, utmp.h

getw() — STDIO Function (libc)

Read word from file stream
#include <stdio.h>
int getw(*fp***) FILE** **fp***;*

getw() reads a word (an **int**) from the file stream *fp*.

getw() differs from **getc()** in that **getw()** gets and returns an **int**, whereas **getc()** returns either a **char** promoted to an **int**, or EOF. To detect EOF while using **getw()**, you must use **feof()**.

See Also

canon, getc(), libc

Notes

getw() returns EOF on errors.

getw() assumes that the bytes of the word it receives are in the natural byte ordering of the machine. This means that such files might not be portable between machines.

GMT — Definition

GMT is an abbreviation of Greenwich Mean Time, the time recorded at the Greenwich Observatory in England, where by international convention the Earth's zero meridian is fixed.

By definition, COHERENT fixes system time in GMT. It calculates local time as an offset of GMT; for example, the time zone for Chicago is six hours (360 minutes) behind Greenwich, so the local time for Chicago is calculated by subtracting 360 minutes from GMT.

See Also

gmtime(), localtime, Programming COHERENT, time, time.h, TIMEZONE

Notes

The ANSI Standard replaces GMT with UTC (*universel temps coordonne* or universal coordinated time) for C programming. The change is mainly one of terminology rather than substance, as some signatories to international conventions object to naming the standard for global time after a suburb of London.

Under international convention, there are two UTC standards: UTC1 is based on solar time and is identical to current GMT; and UTC2, which uses atomic clocks that are corrected by comparison with pulsars. These standards drift apart as the earth's rotation slows; thus, "leap seconds" are inserted periodically into UTC1 to bridge the difference.

gmtime() — Time Function (libc)

Convert system time to calendar structure
#include <time.h>
#include <sys/types.h>
tm *gmtime(*timep***)**
time_t ***timep*;**

gmtime() converts the internal time from seconds since midnight January 1, 1970 GMT, into fields that give integer years since 1900, the month, day of the month, the hour, the minute, the second, the day of the week, and yearday. It returns a pointer to the structure **tm**, which defines these fields, and which is itself defined in the header file **time.h**. Unlike its cousin, **localtime()**, **gmtime()** returns Greenwich Mean Time (GMT).

Example

For an example of how to use this function, see **asctime()**.

See Also

GMT, libc, localtime(), time [overview], TIMEZONE
ANSI Standard, §7.12.3.3
POSIX Standard, §8.1

Notes

gmtime() returns a pointer to a statically allocated data area that is overwritten by successive calls.

gnucpio — Command

Archiving/backup utility
Copy-in mode: **cpio {-o|--create} [-0acvABLV] [-C** *bytes***] [-H** *format***] [-M** *message***] [-O [[***user***@]***host***:]***archive***] [-F [[***user***@]***host***:]***archive***]** [--file=[[*user*@]*host*:]*archive*] **[--format=***format***] [--message=***message***] [--null] [--reset-access-time] [--verbose] [--dot] [--append] [--block-size=blocks] [--dereference] [--io-size=***bytes***] [--version] <** *name-list* **[>** *archive***]**

Copy-out mode: **cpio {-i|--extract} [-bcdfmnrtsuvBSV] [-C** *bytes***] [-E** *file***] [-H** *format***] [-M** *message***] [-R [***user***][:.][***group***]] [-I [[***user***@]***host***:]***archive***] [-F [[***user***@]***host***:]***archive***] [--file=[[***user***@]***host***:]***archive***] [--make-directories] [--nonmatching] [--preserve-modification-time] [--numeric-uid-gid] [--rename] [--list] [--swap-bytes] [--swap] [--dot] [--unconditional] [--verbose] [--block-size=***blocks***] [--swap-halfwords] [--io-size=***bytes***] [--pattern-file=***file***] [--format=***format***] [--owner=[***user***][:.][***group***]] [--no-preserve-owner] [--message=***message***] [--version] [***pattern...***] [<** *archive***]**

Copy-through mode: **cpio {-p|--pass-through} [-0adlmuvLV] [-R [***user***][:.][***group***]] [--null] [--reset-access-time] [--make-directories] [--link] [--preserve-modification-time] [--unconditional] [--verbose] [--dot] [--dereference] [--owner=[***user***][:.][***group***]] [--no-preserve-owner] [--version]** *destination-directory* **<** *name-list*

gnucpio is the GNU version of the archive utility **cpio**. It copies files into or out of a **cpio** or **tar** archive, which is a file that contains other files plus information about them, such as their pathname, owner, timestamps, and access permissions. The archive can be another file on the disk, a magnetic tape, or a pipe.

gnucpio has three operating modes.

Copy-out Mode
: **gnucpio** copies files into an archive. It reads a list of file names, one per line, from the standard input, and writes the archive onto the standard output.

Copy-in Mode
: **gnucpio** copies files from an archive or lists the archive's contents. It reads the archive from the standard input. Any non-option command-line arguments are shell wild-card patterns; only files in the archive whose names match one or more of those patterns are copied from the archive. Unlike in the shell, an initial '.' in a file name does match a wildcard at the start of a pattern, and a '/' in a file name can match wildcards. If the command line contains no pattern, **gnucpio** extracts all files.

Copy-pass Mode
: **gnucpio** copies files from one directory tree to another. This combines the copy-out and copy-in steps without actually using an archive. It reads the list of files to copy from the standard input; the directory into which it copies them is given as a non-option argument.

gnucpio supports the following archive formats: binary, old ASCII, new ASCII, **crc**, old **tar**, and POSIX.1 **tar**. The binary format is obsolete because it encodes information about the files in a way that is not portable between different machine architectures. The old ASCII format is portable between different machine architectures, but should not be used on file systems with more than 65536 i-nodes. The new ASCII format is portable between different machine architectures and can be used on any size file system, but is not supported by all versions of **cpio**; currently, it is only supported by GNU and UNIX System V R4. The **crc** format resembles the new ASCII format, but also contains a checksum for each file that **gnucpio** calculates when creating an archive and verifies when the file is extracted from the archive.

tar format is provided for compatability with the command **tar**. It can not be used to archive a file whose name exceeds 100 characters, and cannot be used to archive block or character devices. The POSIX.1 **tar** format can not be used to archive a file whose name exceeds 255 characters (less unless it has a '/' in just the right place).

By default, **gnucpio** creates binary archives, for compatibility with older **cpio** programs. When extracting from archives, **gnucpio** automatically recognizes the kind of archive it is reading, and can read archives created on machines with a different byte-order.

Options

gnucpio recognizes the following command-line options. Not every option applies to every mode. You can prefix the long-named options with an '+' as well as with an '--', for compatibility with previous releases. Eventually, support for '+' will be removed, because it is incompatible with the POSIX Standard.

-0
--null In copy-out and copy-pass modes, read a list of file names terminated by a null character instead of a newline. This permits **gnucpio** to archive files whose names contain newlines.

-a
--reset-access-time
Reset the access times of files after reading them, so that it does not look like they have just been read.

-A
--append
Append to an existing archive. Only works in copy-out mode. The archive must be a disk file specified with the options **-O** or **-F**.

-b
--swap In copy-in mode, swap both halfwords of words and bytes of halfwords in the data. Equivalent to the option **-sS**. Use this option to convert 32-bit integers between big-endian and little-endian machines.

-B Set the I/O block size to 5,120 bytes. Initially, the block size is 512 bytes.

--block-size=*blocks*
Set the block size to *blocks*×512 bytes.

-c Use the old portable (ASCII) archive format.

-C *size*

--io-size=*size*
Set the I/O block size to *size* bytes.

-d
--make-directories
Create leading directories where needed.

-E file
--pattern-file=*file*
In copy-in mode, read from *file* additional patterns that specify file names to extract or list. **gnucpio** treats the lines of *file* as if they had been non-option arguments to **gnucpio**.

-f
--nonmatching
Copy only the files that do *not* match any of the given patterns.

-F

LEXICON

--file=*archive*
Read to or write from *archive* instead of the standard input or output. When you use this option, you do not have to specify the output device for each volume of a multi-volume backup.

--force-local
With options **-F**, **-I**, or **-O**, take the archive file name to be a local file even if it contains a colon (which ordinarily names a remote host).

-H *format*
--format=*format*
Use archive format *format*. The valid formats are listed below; **gnucpio** also recognizes these names if given in capital letters. The default in copy-in mode is to detect automatically the archive format, and in copy-out mode is **bin**.

bin The obsolete binary format.

odc The old (POSIX.1) portable format.

newc The new (SVR4) portable format, which supports file systems that have more than 65536 i-nodes.

crc The new (SVR4) portable format with a checksum added.

tar The old **tar** format.

ustar The POSIX.1 **tar** format. Also recognizes GNU **tar** archives, which are similar but not identical.

-i
--extract
Run in copy-in mode.

-I *archive*
Archive file name to use instead of standard input.

-k This option exists only for compatibility with other versions of **cpio**. It is ignored.

-l
--link Whenever possible, link files instead of copying them.

-L
--dereference
Dereference symbolic links — that is, copy the files that they point to instead of copying the links.

-m
--preserve-modification-time
Retain previous file-modification times when creating files.

-M *message*
--message=*message*
Print *message* when **gnucpio** reaches the end of a volume of the back-up medium (such as a tape or a floppy disk), to prompt the user to insert a new volume. If *message* contains the string **%d**, **gnucpio** replaces that string with the number of the current volume (starting at one).

-n
--numeric-uid-gid
In the verbose table of contents listing, show the numeric UID and GID instead of translating them into names.

--no-preserve-owner
In copy-in and copy-pass modes, do not change the ownership of the files: leave them owned by the user who extracts them. This is the default for non-root users, so that users on System-V UNIX do not inadvertantly give away files.

-o
--create
Run in copy-out mode.

-O *archive*
Write output into *archive* instead of to the standard output.

-p
--pass-through
Run in copy-pass mode.

-r
--rename
Interactively rename files.

-R [*user***][:.][***group***]**
--owner [*user***][:.][***group***]**
In copy-out and copy-pass modes, set the ownership of all files created to *user* and *group*. Either the user or the group, or both, must be present. If the group is omitted but the ':' or '.' separator is given, **gnucpio** uses the user's login group. Only the super-user can change files' ownership.

-s
--swap-bytes
In copy-in mode, swap the bytes of each halfword (pair of bytes) in the files.

-S
--swap-halfwords
In copy-in mode, swap the halfwords of each word (four bytes) in the files.

-t
--list Print a table of contents of the input.

-u
--unconditional
Replace all files, without asking whether to replace existing newer files with older files.

-v
--verbose
List the files processed. When used with the option **-t**, give a listing that resembles the output of the command **ls -l**. In a verbose table of contents of a **ustar** archive, user and group names in the archive that do not exist on the local system are replaced by the names that correspond locally to the numeric UID and GID stored in the archive.

-V --dot
Print a '.' for each file processed.

--version
Print the number of the version of **gnucpio** that you are now running, and exit.

Examples

The following command copies all files and directories listed by the command **find** and copies them into the archive **newfile.cpio**:

```
find . -print | cpio -oc > ../newfile.cpio
```

The following command reads the **cpio** archive **newfile.cpio** and extracts all files whose names match the patterns **memo/al** or **memo/b***:

```
cpio -icdv "memo/al" "memo/b*" <../newfile.cpio
```

Note that the **-d** option forces **cpio** to create the sub-directory **memo** and write the files into it. Otherwise, the files would have been written into the current directory. Option **-v** causes **cpio** to display each file name as it is extracted from the archive.

The following commands perform a multi-volume backup of all files on mounted filesystem **/v** to the character-special (i.e., "raw") floppy device **/dev/rfha0**:

```
su root
cd /v
find . -print | cpio -ocv >/dev/rfha0
```

If the **cpio** archive exceeds one floppy disk, you will be prompted to insert another.

See Also

commands, cpio, gtar

LEXICON

Notes

COHERENT does not yet support networking. The above descriptions of host addressing do not yet apply.

gnucpio is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.

goto — C Keyword

Unconditionally jump within a function

A **goto** command jumps to the area of the program introduced by a label. A program can **goto** only within a function; to jump across function boundaries, you must use the functions **setjmp()** and **longjmp()**.

In the context of C programming, the most common use for **goto** is to exit from a control block or go to the top of a control block. It is used most often to write "ripcord" routines, i.e., routines that are executed when a major error occurs too deeply within a function for the program to disentangle itself correctly. Note that in most instances, **goto** is a bad solution to a problem that can be better solved by structured programming.

Example

The following example demonstrates how to use **goto**.

```
#include <stdio.h>

main()
{
      char line[80];

getline:
      printf("Enter line: ");
      fflush(stdout);
      gets(line);

/* a series of tests often is best done with goto's */
      if (*line == 'x') {
            printf("Bad line\n");
            goto getline;
      } else if (*line == 'y') {
            printf("Try again\n");
            goto getline;
      } else if (*line == 'q')
            goto goodbye;
      else
            goto getline;


goodbye:
      printf("Goodbye.\n");
      exit(0);
}
```

See Also

C keywords
ANSI Standard, §7.6.6.1

Notes

The C Programming Language describes **goto** as "infinitely-abusable": *caveat utilitor*.

grep — Command

Pattern search

grep searches each *file* for occurrences of the *pattern* (sometimes called a regular expression). If no *file* is specified, **grep** searches the standard input. The *pattern* is given in the same manner as to **ed**. Normally, **grep** prints each line matching the *pattern.*

grep recognizes the following command-line options:

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print the count of matching lines rather than the lines.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contain a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-i Ignore case when matching letters in *pattern*. For example, an 'a' in *pattern* matches either 'a' or 'A' in *file*; likewise, an 'A' in *pattern* matches either 'a' or 'A'.

-l Print the name of each file containing matching lines rather than the lines.

-n The line number in the file accompanies each line printed.

-s Suppress all output, just return status.

-v Print a line if the pattern is *not* found in the line.

-x Print the line only if it is exactly the same as the pattern; treat wildcards in the pattern as plain text.

-y Lower-case letters in *pattern* match only upper-case letters within the input lines.

Limits

The COHERENT implementation of **grep** sets the following limits on input and output:

Characters per input record	512
Characters per output record	512
Characters per field	512

See Also

cgrep, commands, ed, egrep, zgrep

Diagnostics

grep returns an exit status of zero for success, one for no matches, two for error.

Notes

cgrep is a version of **grep** that is optimized for handling C-style expressions.

egrep is an extended and faster version of **grep**.

group — System Administration

Define groups of users

The group file **/etc/group** describes the user groups that have been defined on your COHERENT system. This allows users to control the access that members of their group have to certain files. **/etc/group** contains the information to map any ASCII group name to the corresponding numerical group identifier, and vice versa. It also contains, in ASCII, the names of the members of each group. This information is used by, among others, the command **newgrp**.

Each group has an entry in the file **/etc/group** one line per entry. Each line consists of four colon-separated ASCII fields, as follows:

group_name **:** *password* **:** *group_number* **:** *member***[,***member***...]**

Passwords are encrypted with **crypt**, so the group file is generally readable.

The COHERENT system has five system calls that manipulate **/etc/group**, as follows:

endgrent() Close **/etc/group**.

getgrent() Return the next entry from **/etc/group**.

getgrnam() Return the first entry with a given group name.

getgrgid() Return the first entry with a given group identifier.

setgrent() Rewind **/etc/group**, so that searches can begin again from the beginning of the file.

The calls **getgrent()**, **getgrid()**, and **getgrnam()** each return a pointer to structure **group**, which the header file **grp.h** defines as follows:

```
struct group {
        char    *gr_name;       /* Group name */
        char    *gr_passwd;     /* Group password */
        int     gr_gid;         /* Numeric group id */
        char    **gr_mem;       /* Group members */
};
```

A user can belong to more than one group. His "main" group, however, is the one that is named is in his entry in the file **/etc/passwd**. When a user creates a file, that file by default is "owned" by the user's main group.

For example, consider user **joe**, who has the following entry in **/etc/passwd**:

```
joe:*:10:5:Joe Smith:/usr/joe:/usr/bin/ksh
```

The fourth field, which in this example has the value **5**, gives the number of the user's main group. (For details on what the other fields mean, see the Lexicon entry for **passwd**.) Looking in **/etc/group**, we see the following entry for group 5:

```
user::5:
```

Thus, whenever **joe** creates a file, by default it will be "owned" by group **user**. Any member of group **user** will be granted that file's group-level permissions on that file.

A user can use the command **chmod** to change the group-level permissions on any file he owns. The superuser **root** can use the command **chgrp** to changes the group ownership for any file. For details on how to use these commands, see their entries in the Lexicon.

Files

/etc/group

See Also

Administering COHERENT, **chgrp(), chmod, chown, endgrent(), getgrent(), getgrgid(), getgrnam(), grp.h, newgrp, passwd, setgrend()**

Notes

At present the group password field cannot be set directly (no command similar to **passwd** exists for groups). One alternative is to set the password in the **/etc/passwd** file for a user with the **passwd** command, then transcribe the password into the group file manually.

grp.h — Header File

Declare group structure
#include <grp.h>

The header file **grp.h** declares the structure **group**, which is composed as follows:

```
struct group (
        char *gr_name;          /* group name */
        char *gr_passwd;        /* group password */
        int  gr_gid;            /* numeric group id */
        char **gr_mem;          /* group members */
};
```

This structure holds information about the group to which a given user belongs, as defined in the file **/etc/group**. It is used by the functions **endgrent()**, **getgrent()**, **getgrgid()**, **getgrnam()**, and **setgrent()**.

See Also

header files
POSIX Standard, §9.2.1

gtar — Command

Archiving/backup utility
gtar *options*

gtar is the GNU version of the archiving utility **tar**. It copies files into or out of a **tar** archive, reads the contents of a **tar** archive, and replaces files within an archive. It can also perform additional tasks such as compressing files as they are added to an archive, or uncompressing them as they are read out.

gtar works in either of two modes:

Copy-in Mode
: **gtar** copies files from an archive or lists the archive's contents. By default, it reads the archive from the standard input; you can also use the option **-f** (described below) to name the file or device that holds the archive you want read.

  gtar regards any non-option argument as a shell wild-card pattern; and it copies from the archive only those files whose names match one or more of those patterns. Unlike the shell, an initial '.' in a file name matches a wildcard at the start of a pattern, and a '/' in a file name can match a wildcard. If the command line contains no pattern, **gtar** extracts all files.

Copy-out Mode
: **gtar** copies files into an archive. By default, **gtar** reads a list of file names, one per line, from the standard input. However, if the command line contains non-option arguments, **gtar** regards each as a shell wild-card pattern that names one or more files to copy into the archive. If an argument names a directory, then **gtar** recursively copies all files within that directory into the archive.

  By default, **gtar** writes its newly built archive to the standard output. However, you can use the option **-f** (described below) to name the file or device into **gtar** writes the new archive.

  gtar normally writes into the local directory all files that it reads from an archive. If files were archived using absolute path names, **gtar** by default drops the leading '/' from the path name; to suppress this behavior, use the option **-P**, described below. If a file being extracted resides within a directory that does not exist in the current directory, **gtar** will create that directory. **gtar** will fail, of course, if you do not have write permission in the current directory.

Options

gtar recognizes the following options. Please note that not every option applies to both modes.

Please note, too, that some options have more than one name. Every option has a multi-character name that begins with with two hyphens **--**; some commonly used options also have a one-character name that begins with a single hyphen. This convention may appear clumsy, but it does permit option names to have hyphens embedded within them.

The following command-line options govern the mode in which **gtar** works:

-A
--catenate
--concatenate
: Append files onto an archive.

-c
--create
: Create a new archive.

-d
--diff
--compare
: Find the differences between the files in an archive and the identically named files in the file system. This is very useful in verifying that a new archive was built correctly.

--delete
: Delete files from the archive. Do not for use this option with an archive that is on a magnetic tape.

-r

--append
Replace files within an archive. If a file does not exist within an archive, append it onto the archive.

-t
--list List the contents of an archive.

-u
--update
Append a file onto an archive only if it is younger than the identically named file within the archive.

--use-compress-program
Specify the compression program to use. By default, **gtar** invokes **gzip** to compress files.

-x
--extract
--get Extract files from the archive.

The following options modify other aspects of **gtar**'s behavior:

--atime-preserve
Do not change the access times on files, whether copying into or out of an archive.

-b *N*
--block-size *N*
Use a block size of *N*×512 bytes. By default, **gtar** uses an *N* of 20 — that is, a block size of ten kilobytes.

-B
--read-full-blocks
Tell **gtar** to reblock as it reads. This is required for reading pipes under Berkeley UNIX release 4.2, and does not apply to COHERENT.

--block-compress [compress | gzip]
Block the output of the compression program for tapes. You must name one of the compression options to use: either **compress** or **gzip**.

-C *directory*
--directory *directory*
Change to *directory*.

--checkpoint
Print directory names while reading the archive.

--exclude *file*
Do not include *file* when archiving or de-archiving files. *file* can be a regular expression.

-f *file*
--file *file*
Read the input from, or write the output to, *file*. *file* can name an ordinary file or a device. File name '-' indicates the standard input or standard output (depending upon whether an archive is being read or written). When this option is not used, **gtar** by default reads from the standard input and writes to the standard output.

--force-local
The archive file is local even if its name contains a colon. **gtar** usually interprets a file name that contains a colon as naming a file on a remote system that is connected via a network.

-F *script*
--info-script *script*
--new-volume-script *script*
At the end of each tape (or disk), run *script*. Note that this option implies that you are also using option **-M**.

-G [*file ...***]**
--incremental
Create, list, or extract every *file* that is in an archive written in the format of the old GNU incremental backup. If no *file* is named, all **gtar** extracts all files.

-g
--listed-incremental
Create, list, or extract files that are in an archive written in the format of the new GNU incremental backup. create/list/extract new GNU-format incremental backup

-i
--ignore-zeros
Ignore blocks of zeros in archive.

--ignore-failed-read
gtar normally exits with non-zero status when it encounters an unreadable file. With this option, **gtar** ignores the unreadable file and continues to work.

-k
--keep-old-files
If a file being extracted from an archive has an identically named analogue in the file system, **gtar** normally overwrites the file in the file system with the file withdrawn from the archive. This option tells **gtar** to rename the file that is in the file system, rather than overwrite it.

-K *file*
--starting-file *file*
Keep option: begin work with *file* in the archive.

-l
--one-file-system
Stay in the local file system when creating an archive.

-L *N*
--tape-length *N*
Change tapes after writing *N*×1,024 bytes. **gtar** normally reads or writes until it reaches the end of the medium, then prompts for the name of the next device. This option, of course, normally does not apply to archives being written to or read from disk.

-m
--modification-time
Do not extract file modified time.

-M
--multi-volume
Create, list, or extract a multi-volume archive. You can use this option with multiple **-f** options. **gtar** uses the output devices in sequence, then wraps around to the beginning. This lets you, say, write output to two different tape drives or floppy-disk drives; you can loading blank media into one while **gtar** is writing to the other. Note that if you are using this option to create an archive, be *very* careful to label disks or tapes correctly to note the order in which they were written.

-N *date*
--after-date *date*
--newer *date*
Only store files newer than *date*.

-o
--old-archive
--portability
Write a V7-format archive, rather than an ANSI-format archive.

-O
--to-stdout
Write files to the standard output.

-p
--same-permissions
--preserve-permissions
Preserve the permissions that the file had originally.

-P

--absolute-paths
Do not strip leading '/'s from file names.

--preserve
This option is identical to **-p** plus **-s**.

-R
--record-number
Show record number within archive with each message.

--remove-files
Remove files after adding them to the archive.

-s
--same-order
--preserve-order
Sort the list of names to extract to match their order within the archive.

--same-owner
Create extracted files with the same ownership they had within the archive.

-S
--sparse
Handle sparse files efficiently. For a description of what a *sparse file* is, see the Lexicon entry for **chsize()**.

--show-omitted-dirs
Print the names of directories omitted from the archive.

-T file
--files-from *file*
Read from *file* the names of all files to archive or extract.

--null Modify option **-T** so that it reads null-terminated names. This option disables option **-C**.

--totals Print the number of bytes written with option **-c**.

--use-compress-program *program*
Filter the archive through *program*. Note that *program* must accept option **-d**.

-v
--verbose
Write the names of all files archived or extracted. When you also use the option **-f**, **gtar** writes the names to the standard output; however, when you do not use **-f**, it writes them to the standard error.

-V *name*
--label *name*
Name the archive *name*. When used with the option **--extract**, *name* can be a regular expression.

--version
Print the version of **gtar** that you are using.

--volno-file *file*
Read from *file* the volume number used when prompting the user. Note that **gtar** does not use the contents of *file* when it records volume identifiers on the archive.

-w
--interactive
--confirmation
Ask the user to confirm every action.

-W
--verify Attempt to verify the archive after writing it.

-X *file*
--exclude-from *file*
Do *not* archive or de-archive all of the files named in *file*.

-Z
--compress
--uncompress
Filter files being archived or de-archived through **compress**.

-z
--gzip
--ungzip
Filter files being archived or de-archived through **gzip**.

Examples

The first example archives **piggy**, into archive **piggy.tar**:

```
gtar -cf piggy.tar piggy
```

To simultaneously compress **piggy** with the utility **gzip**, use the command:

```
gtar -czf piggy.gtz piggy
```

Note that the suffix **.gtz** is used by convention to mark archives whose contents are compressed. This is not required, but it is a good idea to use this or some similar suffix to mark compressed archives: if you do not remember to use the **-z** option to de-archive a compress archive, **gtar** will fail. So, to extract file **piggy** from its compressed archive, use the command:

```
gtar -xzf piggy.gtz piggy
```

The **-z** is recommended: it speeds archiving of large files or file systems, and increases their accuracy — because the archives are smaller, there are fewer opportunities for errors to occur.

To write an archive onto a device, use the option **-f** to name that device instead of a file. You must, of course, have write permission on that device. If you are writing onto a floppy disk, the disk must have been formatted with the command **fdformat**, but does not need to have a COHERENT file system on it; in fact, **gtar** will overwrite all file-system information that may reside on a disk. For example, to write file **piggy** onto a high-density, 5.25-inch, formatted floppy disk in drive 0, use the following command:

```
gtar -czf /dev/fha0 piggy
```

To copy **piggy** back from this archive, use the command:

```
gtar -xzf /dev/fha0
```

As noted above, you must remember to use the **-z** option to de-archive files from a compressed archive.

As noted above, if you name a directory on **gtar**'s command line, **gtar** will archive or de-archive that directory and all files that it contains, including its sub-directories and their contents. For example, to archive all of your personal files, use the command:

```
gtar -cvzf backup.gtz $HOME
```

The option **-v** tells **gtar** to name every file that it is copying into its archive. Note, too, that **gtar** is smart enough not to copy an archive into itself, so you can execute the above command while still within your home directory.

The following backs up your personal files onto a high-density, 3.5-inch disk in drive 0:

```
gtar -cvzf /dev/fva0 $HOME
```

NB, if you are backing up a directory that will require more than one floppy disk, you should consider using the utility **cpio** instead: it is somewhat easier to use when you are handling multiple-volume archives.

To copy directory **src** to the SCSI tape device with SCSI identifier 2, use the command:

```
tar cvzf /dev/rStp2 src
```

To archive **src** to a tape and then confirm it, use the command

```
tar cvzf /dev/rStp2 src ; tar dvzf /dev/rStp2 src
```

Note that this can be time consuming, but will confirm the integrity of backups of vital files. To restore **src** from its tape, use the command:

```
tar xvzf /dev/rStp2
```

LEXICON

gtar by default saves files with their original ownerships and permissions; however, when it restores files, it may modify them. To restore files with their original permissions, use the option **-p**. For example, to restore **src** and restore the original ownership and permissions of its files, use the command:

```
tar xvpzf /dev/rStp2
```

See Also

commands, compression, gnucpio, tar
POSIX Standard, §10.1.1

Notes

COHERENT does not yet support networking. The above descriptions of host addressing do not yet apply.

gtar is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.

gtty() — System Call (libc)

Device-dependent control
#include <sgtty.h>
int gtty(*fd, sgp***)**
int *fd***; struct sgttyb** **sgp***;*

gtty() gets attributes of a terminal. It is shorthand notation for **ioctl** calls with a *command* argument of **TIOCGETP**.

Example

For examples of this system call, see **pipe()** and **stty()**.

See Also

exec, libc, ioctl(), open(), read(), sgtty.h, stty(), write()

guess — Command

Extraordinarily amusing guessing game
/usr/games/guess

The COHERENT game **guess** plays a guessing game with you. When you first invoke it, it will ask you to think of an object. As you go through the guessing game, it will ask you for questions by which that object can be distinguished from other objects. **guess** gets "smarter" over time (assuming you don't lie to it), so it over time develops a fighting chance of actually guessing something.

See Also

commands

Notes

guess is not for the impatient.

gunzip — Command

GNU utility to uncompress files
gunzip [-cfhLrtvV] [*file ...* **]**

gunzip is the GNU command that uncompresses each *file* named on its command line.

Whenever possible, **gunzip** replaces each *file* whose name ends with **.z** or **.Z** (and which begins with the correct magic number) with an uncompressed file without the original suffix. **gunzip** also recognizes the special extensions **.tgz** and **.taz** as shorthands for **.tar.z** or **.tar.Z**.

gunzip can currently decompress files created by the COHERENT commands **gzip** or **compress**, or by the UNIX commands **zip** or **pack**. It automatically detects the format by which the file is compressed and applies the correct algorithm to uncompress it.

When uncompressing the formats used by **gzip** and **zip**, **gunzip** checks a 32-bit CRC. For files compressed by **pack**, **gunzip** checks the uncompressed length.

The format used by **compress** was not designed to allow consistency checks. However, **gunzip** can sometimes

detect a corrupted **.Z** file. If you get an error when uncompressing a **.Z** file, do not assume that the **.Z** file is correct simply because the COHERENT command **uncompress** does not complain. This generally means that most implementations of **uncompress** do not check their input, and happily generate garbage output.

Command-Line Options

gunzip recognizes the following command-line options:

-c Write output to standard output, and do not change the original *file*. If the command line names more than one *file*, **gzip** writes to the standard output a sequence of independently compressed members. To obtain better compression, concatenate the *file*s before compressing them.

-f force compression or decompression, even if *file* has multiple links or the corresponding file already exists. Without this option, and when not running in the background, **gzip** prompts to verify whether it should overwrite an existing file.

-h Help: display a screenful of information on how to run this program.

-L Display the **gzip** license.

-r Recurse: if a *file* is a directory, compress or uncompress all files within it.

-t Test: check the integrity of a compressed file.

-v Verbose: display the name and percentage reduction for each *file* as it is compressed.

-V Display the version of this command, and the options by which it was compiled.

See Also

commands, compress, compression, gzip, unpack

Diagnostics

gunzip returns zero if all went well. It returns one if an error occurred and it returns two if it had to issue a warning message.

gunzip can issue the following warning messages:

*file***: not in gzip format**
A *file* named on the command line was not compressed.

The compressed file has been damaged. If the data were compressed by the program **compress**, they can be recovered up to the point of damage by using the program **zcat** to concatenate the file into another file.

*file***: compressed with** *XX* **bits, can only handle** *YY* **bits**
file was compressed by a program that could deal with more bits than the decompress code on this machine. Recompress the file with **gzip**, which compresses better and uses less memory.

*file***: already has z suffix -- no change**
file has the suffix **.z** or **.Z**; therefore, **gunzip** assumes that it is compressed already.

file **already exists; do you wish to overwrite (y or n)?**
Respond 'y' if you want the output file to be replaced; 'n' if not.

gunzip: corrupt input
gunzip detected a **SIGSEGV** violation, which usually means that the input file has been corrupted.

Notes

gzip is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.

gzip — Command

GNU utility to compress files
gzip [-cdfhLrtvV19] [*file ...* **]**

The command **gzip** is the GNU command that compresses *file* named on its command line. It will only attempt to compress regular files.

Whenever possible, **gzip** replaces each *file* with one that has the suffix **.gz**, while preserving its ownership and times of last access and last modification. If the name of *file* is longer than 12 characters (which prevents **gzip**

from attaching the suffix **.gz**), **gzip** truncates it and keeps its original name within the compressed file.

If its command line names no *file*, **gzip** compresses what it reads from from the standard input, and writes it to the standard output.

To restore a compressed file, filter it thorugh the command **gunzip**.

gzip uses the Lempel-Ziv algorithm to perform compression. Under most circumstances, this algorithm compresses files more tightly than do most other commonly used techniques, such as the LZW algorithm, Huffman coding, or adaptive Huffman coding. The amount of compression obtained depends upon the size of the input and the distribution of common substrings; in general, it reduces text by 60% to 70%.

gzip always compresses its input, even if the compressed file is slightly larger than the original. The worst-case expansion is a few bytes for the **gzip** file header, plus five bytes for every 32-kilobyte block.

Command-Line Options

gzip recognizes the following command-line options:

-c Write output to standard output, and do not change the original *file*. If the command line names more than one *file*, **gzip** writes to the standard output a sequence of independently compressed members. To obtain better compression, concatenate the *file*s before compressing them.

-d Decompress each *file*.

-f force compression or decompression, even if *file* has multiple links or the corresponding file already exists. Without this option, and when not running in the background, **gzip** prompts to verify whether it should overwrite an existing file.

-h Help: display a screenful of information on how to run this program.

-L Display the **gzip** license.

-q Suppress all warning messages.

-r Recurse: if a *file* is a directory, compress or uncompress all files within it.

-t Test: check the integrity of a compressed file.

-v Verbose: display the name and percentage reduction for each *file* as it is compressed.

-V Display the version of this command, and the options by which it was compiled.

1-9 Regulate the speed of compression, on a scale of from **1** to **9**. **1** performs the fastest but most superficial compression, whereas **9** performs the slowest but most thorough compression. **-fast** is a synonym for **-1**, whereas **-best** is a synonym for **-9**. The default compression level is **-5**.

Advanced Usage

You can concatenate multiple compressed files. In this case, **gunzip** extracts all members at once. For example:

```
gzip -c file1 > foo.gz
gzip -c file2 >> foo.gz
gunzip -c foo
```

is equivalent to:

```
cat file1 file2
```

In case of damage to one member of a **.gz** file, other members can still be recovered (if the damaged member is removed). However, you can get better compression by compressing all members at once:

```
cat file1 file2 | gzip > foo.gz
```

compresses better than:

```
gzip -c file1 file2 > foo.gz
```

If you want to recompress concatenated files to get better compression, type:

```
zcat old.gz | gzip > new.gz
```

Environment

gzip reads the environment variable **GZIP** for its default options. It interprets these options first; you can override them by setting other options on the **gzip** command line.

See Also

commands, compress, compression, gunzip, uncompress, unpack, zcmp, zdiff, zforce, zgrep, zmore, znew

Diagnostics

If all went well, **gzip** returns zero upon exiting. If an error occurred, it returns one; if it issued a warning message, it returns two.

gzip can issue the following warning messages:

Usage: gzip [-cdfhLrtvV19] [file ...]
The **gzip** command line contained an option that **gzip** does not recognize.

*file***: already has z suffix -- no change**
file already has the suffix **.gz**; therefore, **gzip** assumes that it already is compressed.

file **not a regular file or directory: ignored**
A *file* is not a regular file or directory. **gzip** does not attempt to compress devices, pipes, or other special files.

file **has** *XX* **other links: unchanged**
file has more than one link. By default, **gzip** does not compress a file that has multiple links.

Notes

gzip is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.