AWK REFERENCE

CONTENTS

Arnold Robbins wrote this reference card. We thank Brian Kernighan and Michael Brennan who reviewed it.

OTHER FSF PRODUCTS:

Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301 USA
Phone: +1-617-542-5942
Fax (including Japan): +1-617-542-2652
E-mail: gnu@gnu.org
URL: https://www.gnu.org

Source Distributions on CD-ROM
Emacs, Make and GDB Manuals
Emacs and GDB References

DEFINITIONS

This card describes POSIX AWK, as well as three freely available **awk** implementations (see FTP/HTTP/GIT Information). Common extensions (in two or more versions) are printed in light blue. Features specific to just one version—usually GNU AWK (**gawk**)—are printed in dark blue. Exceptions and deprecated features are printed in red. Features mandated by POSIX are printed in black.

Several type faces are used to clarify the meaning:
- **`Courier Bold`** is used for computer input.
- *Times Italic* is used for emphasis, to indicate user input and for syntactic placeholders, such as *variable* or *action*.
- Times Roman is used for explanatory text.

number − a floating point number as in ANSI C, such as **`3`**, **`2.3`**, **`.4`**, **`1.4e2`** or **`4.1E5`**. Numbers may also be given in octal or hexadecimal: e.g., **`011`** or **`0x11`**.

escape sequences − a special sequence of characters beginning with a backslash, used to describe otherwise unprintable characters. (See Escape Sequences.)

string − a group of characters enclosed in double quotes. Strings may contain *escape sequences*.

regexp − a regular expression, either a regexp constant enclosed in forward slashes, or a dynamic regexp computed at run-time. Regexp constants may contain *escape sequences*.

strongly typed regexp − a regular expression constant with a leading **`@`**. E.g., **`@/stuff/`**. Such constants may be assigned to variables and passed to user-defined functions.

name − a variable, array or function name.

entry(*N*) − entry *entry* in section *N* of the Unix reference manual.

pattern − an expression describing an input record to be matched.

action − statements to execute when an input record is matched.

rule − a pattern-action pair, where the pattern or action may be missing.

COMMAND LINE ARGUMENTS (standard)

Command line arguments control setting the field separator, setting variables before the **`BEGIN`** rule runs, and the location of AWK program source code. Implementation-specific command line arguments change the behavior of the running interpreter.

`-F` *fs*	Use *fs* for the input field separator.
`-v` *var*`=`*val*	Assign the value *val* to the variable *var* before execution of the program begins. Such variable values are available to the **`BEGIN`** rule.
`-f` *prog-file*	Read the AWK program source from the file *prog-file*, instead of from the first command line argument. Multiple **`-f`** options may be used.
`--`	Signal the end of options.

BUG REPORTS

If you find a bug in this reference card, please report it via electronic mail to **`bug-gawk@gnu.org`**.

COMMAND LINE ARGUMENTS (gawk)

Long options may abbreviated as long as the abbreviation remains unique. You may use "**-W** *option*" for full POSIX compliance.

--assign *var=val* Same as **-v**.
--field-separator *fs* Same as **-F**.
--file *prog-file* Same as **-f**.

-b, **--characters-as-bytes**
Treat all input data as single-byte characters. Overridden by **--posix**.

-c, **--traditional**
Disable **gawk**-specific extensions.

-C, **--copyright**
Print the short GNU copyright information on **stdout**.

-d[*file*], **--dump-variables**[**=***file*]
Print a sorted list of global variables, their types and final values to *file* (default: **awkvars.out**).

-D[*file*], **--debug**[**=***file*]
Enable debugging of program. Optionally read stored commands from *file*.

-e '*text*', **--source** '*text*'
Use *text* as AWK program source code.

-E *file*, **--exec** *file*
Read program text from *file*. No other options are processed. Also disable command-line variable assignments. Useful with **#!**.

-g, **--gen-pot**
Process the program and print a GNU **gettext** format **.pot** file on **stdout**, containing the text of all strings that were marked for localization.

-h, **--help**
Print a short summary of the available options on **stdout**, then exit zero.

-i *file*, **--include** *file*
Include library AWK code in *file*. See Awk Program Execution.

-l *lib*, **--load** *lib*
Load dynamic extension *lib*. See Dynamic Extensions.

-L [*value*], **--lint**[**=***value*]
Warn about dubious or non-portable constructs. If *value* is **fatal**, lint warnings become fatal errors. If *value* is **invalid**, only issue warnings about things that are actually invalid (not fully implemented yet).

-M, **--bignum**
Enable arbitrary-precision arithmetic.

-n, **--non-decimal-data**
Recognize octal and hexadecimal values in input data. *Use this option with great caution!*

-N, **--use-lc-numeric**
Force use of the locale's decimal point character when parsing input data.

-o[*file*], **--pretty-print**[**=***file*]
Output a pretty printed version of the program to *file* (default: **awkprof.out**).

-O, **--optimize**
Enable internal optimizations (default is on).

-p[*file*], **--profile**[**=***file*]
Send profiling data to *file* (default: **awkprof.out**). The profile contains execution counts in the left margin of each statement in the program.

-P, **--posix**
Disable common and GNU extensions.

-r, **--re-interval**
Enable *interval expressions*. (Needed with **-c**.)

COMMAND LINE ARGUMENTS (gawk)

-s, **--no-optimize**
Disable internal optimizations.

-S, **--sandbox**
Disable the **system()** function, input redirection with **getline**, output redirection with **print** and **printf**, and loading dynamic extensions.

-t, **--lint-old**
Warn about constructs that are not portable to the original version of Unix **awk**.

-V, **--version**
Print version info on **stdout** and exit zero.

Normally, if there is program text, unknown options are passed on to the AWK program in **ARGV** for processing. In compatibility mode, unknown options are flagged as invalid, but are otherwise ignored.

COMMAND LINE ARGUMENTS (mawk)

The following options are specific to **mawk**.

-W dump Print an assembly listing of the program to **stdout** and exit zero.
-W exec *file* Read program text from *file*. No other options are processed. Useful with **#!**.
-W interactive Unbuffer **stdout** and line buffer **stdin**. Lines are always records, ignoring **RS**.
-W posix_space **\n** separates fields when **RS = ""**.
-W sprintf=*num* Adjust the size of **mawk**'s internal **sprintf** buffer.
-W version Print version and copyright on **stdout**, limit information on **stderr**, and exit zero.

The options may be abbreviated using just the first letter, e.g., **-We**, **-Wv** and so on.

SIGNALS (gawk --profile)

gawk accepts two signals while profiling. **SIGUSR1** dumps a profile and function call stack to the profile file. It then continues to run. **SIGHUP** is similar, but exits.

LINES AND STATEMENTS

AWK is a line-oriented language. The pattern comes first, and then the action. Action statements are enclosed in **{** and **}**. Either the pattern or the action may be missing, but not both. If the pattern is missing, the action is executed for every input record. A missing action is equivalent to

```
{ print }
```

which prints the entire record.

Comments begin with the **#** character, and continue until the end of the line. Normally, statements end with a newline, but lines ending in a ",", **{**, **?**, **:**, **&&**, or **||**, are automatically continued. Lines ending in **do** or **else** also 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 is ignored. However, a "\" after a **#** is not special.

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.

AWK PROGRAM EXECUTION

AWK programs are a sequence of optional directives, pattern-action statements and optional function definitions.

```
@include "filename"
@load "filename"
pattern      { action statements }
function name(parameter list) { statements }
```

awk first reads the program source from the *prog-file*(s), if specified, from arguments to **--source**, or from the first non-option argument on the command line. The program text is read as if all the *prog-file*(s) and command line source texts had been concatenated.

gawk includes files named on **@include** lines. Nested includes are allowed. **gawk** loads extensions named on **@load** lines; see Dynamic Extensions.

AWK programs execute in the following order. First, all variable assignments specified via the **-v** option are performed. Next, **awk** executes the code in the **BEGIN** rules(s), if any, and then proceeds to read the files **1** through **ARGC - 1** in the **ARGV** array. If there are no files named on the command line, **awk** reads the standard input.

A command line argument of the form *var*=*val*, is treated as a variable assignment. The variable *var* is assigned the value *val*. (This happens after any **BEGIN** rule(s) have been run.)

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

For each input file, if a **BEGINFILE** rule exists, **gawk** executes the associated code before processing the contents of the file. Similarly, **gawk** executes the code associated with **ENDFILE** after processing the file.

For each record in the input, **awk** tests to see if it matches any *pattern* in the AWK program. For each pattern that the record matches, it executes the associated *action*. The patterns are tested in the order they occur in the program.

Finally, after all the input is exhausted, **awk** executes the code in the **END** rule(s), if any.

If a program only has a **BEGIN** rule, no input files are processed. If a program only has an **END** rule, the input is read.

VARIABLES

ARGC	Number of command line arguments.
ARGIND	Index in **ARGV** of current data file.
ARGV	Array of command line arguments. Indexed from zero to **ARGC** − 1. Dynamically changing the contents of **ARGV** can control the files used for data.
BINMODE	Controls "binary" mode for all file I/O. Values of 1, 2, or 3, indicate input, output, or all files, respectively, should use binary I/O. (Not Brian Kernighan's **awk**.) Applies only to non-POSIX systems. For **gawk**, string values of **"r"**, or **"w"** specify that input files, or output files, respectively, should use binary I/O. Use **"rw"** or **"wr"** for all files.

VARIABLES (continued)

CONVFMT	Conversion format for numbers, default value is **"%.6g"**.
ENVIRON	Array containing the current environment. It is indexed by the environment variable names, each element being the value of that variable.
ERRNO	String error value if a **getline** redirection or read fails, or if **close()** fails.
FIELDWIDTHS	Whitespace-separated list of field widths. Used to parse the input into fields of fixed width, instead of the value of **FS**. See Fields.
FILENAME	Name of the current input file. If no files given on the command line, **FILENAME** is "−". **FILENAME** is undefined inside the **BEGIN** rule (unless set by **getline**).
FNR	Record number in current input file.
FPAT	Regular expression describing field contents. Used to parse the input based on the fields instead of the field separator.
FS	Input field separator, a space by default (see Fields).
FUNCTAB	An array indexed by the names of all user-defined and extension functions.
IGNORECASE	If non-zero, all regular expression and string operations ignore case. Array subscripting is *not* affected. However, the **asort()** and **asorti()** function are affected.
LINT	Provides dynamic control of the **--lint** option from within an AWK program.
NF	Number of fields in the current input record.
NR	Total number of input records seen so far.
OFMT	Output format for numbers, **"%.6g"**, by default.
OFS	Output field separator, a space by default.
ORS	Output record separator, a newline by default.
PREC	The working precision of arbitrary precision floating-point numbers, 53 by default.
PROCINFO	Elements of this array provide access to information about the running AWK program. See *GAWK: Effective AWK Programming* for details.
RLENGTH	Length of the string matched by **match()**; −1 if no match.
ROUNDMODE	The rounding mode to use for arbitrary precision arithmetic, by default **"N"**.
RS	Input record separator, a newline by default (see Records).
RSTART	Index of the first character matched by **match()**; zero if no match.
RT	Record terminator. **gawk** sets **RT** to the input text that matched the character or regular expression specified by **RS**.
SUBSEP	Character(s) used to separate multiple subscripts in array elements, by default **"\034"**. (See Arrays).
SYMTAB	An array indexed by the names of all global variables and arrays. May be used to indirectly set variable and array values.
TEXTDOMAIN	The internationalization text domain, for finding the localized translations of the program's strings.

ARRAYS

An array subscript is an expression between square brackets (**[** and **]**). If the expression is a list (*expr*, *expr* ...), then the subscript is a string consisting of the concatenation of the (string) value of each expression, separated by the value of **SUBSEP**. This simulates multi-dimensional arrays. For example:

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

assigns **"hello, world\n"** to the element of the array **x** indexed by the string **"A\034B\034C"**. All arrays in AWK are associative, i.e., indexed by string values.

Use the special operator **in** in an **if** or **while** statement to see if a particular value is an array index.

```
if (val in array)
        print array[val]
```

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

Use the **in** construct in a **for** loop to iterate over all the elements of an array.

Use the **delete** statement to delete an element from an array. Specifying just the array name without a subscript in the **delete** statement deletes the entire contents of an array. You cannot use **delete** with **FUNCTAB** or **SYMTAB**.

gawk provides true multidimensional arrays. Such arrays need not be ''rectangular'' as in C or C++. For example:

```
a[1] = 5; a[2][1] = 6; a[2][2] = 7
```

EXPRESSIONS

Expressions are used as patterns, for controlling conditional action statements, and to produce parameter values when calling functions. Expressions may also be used as simple statements, particularly if they have side-effects such as assignment. Expressions mix *operands* and *operators*. Operands are constants, fields, variables, array elements, and the return values from function calls (both built-in and user-defined).

Regexp constants (*/pat/*), when used as simple expressions, i.e., not used on the right-hand side of **~** and **!~**, or as arguments to the **gensub()**, **gsub()**, **match()**, **patsplit()**, **split()**, and **sub()**, functions, mean **$0 ~ /pat/**.

The AWK operators, in order of decreasing precedence, are:

(...)	Grouping
$	Field reference
++ --	Increment and decrement, prefix and postfix
^ **	Exponentiation
+ - !	Unary plus, unary minus, and logical negation
*** / %**	Multiplication, division, and modulus
+ -	Addition and subtraction
space	String concatenation
< >	Less than, greater than
<= >=	Less than or equal, greater than or equal
== !=	Equal, not equal
~ !~	Regular expression match, negated match
in	Array membership
&&	Logical AND, short circuit
||	Logical OR, short circuit
?:	In-line conditional expression
= += -= *= /= %= ^= **=	Assignment operators

CONVERSIONS AND COMPARISONS

Variables and fields may be (floating point) numbers, strings or both. Context determines how a variable's value is interpreted. If used in a numeric expression, it will be treated as a number, if used as a string it will be treated as a string. Assigning a strongly typed regexp constant to a scalar makes it a regexp.

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.

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

When a string must be converted to a number, the conversion is accomplished using *strtod*(3). A number is converted to a string by using the value of **CONVFMT** as a format string for *sprintf*(3), 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.

Comparisons are performed 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.

Note that string constants, such as **"57"**, are *not* numeric strings, they are string constants. The idea of ''numeric string'' only applies to fields, **getline** input, **FILENAME**, **ARGV** elements, **ENVIRON** elements and the elements of an array created by **split()** or **patsplit()** that are numeric strings. The basic idea is that *user input*, and only user input, that looks numeric, should be treated that way.

PATTERN ELEMENTS

AWK patterns may be one of the following.

```
BEGIN
END
BEGINFILE
ENDFILE
expression
pat1,pat2
```

BEGIN and **END** are special patterns that provide start-up and clean-up actions respectively. They must have actions. There can be multiple **BEGIN** and **END** rules; they are merged and executed as if there had just been one large rule. They may occur anywhere in a program, including different source files.

BEGINFILE and **ENDFILE** are special patterns that execute before the first record of each file and after the last record of each file, respectively. In the **BEGINFILE** rule, the **ERRNO** variable is non-null if there is a problem with the file; the rule should use **nextfile** to skip the file if desired. Otherwise **gawk** exits with its usual fatal error. The actions for multiple **BEGINFILE** and **ENDFILE** patterns are merged.

Expression patterns can be any expression, as described under Expressions.

The *pat1*,*pat2* pattern is called a *range pattern*. It matches all input records starting with a record that matches *pat1*, and continuing until a record that matches *pat2*, inclusive. It does not combine with any other pattern expression.

ACTION STATEMENTS

break
Break out of the nearest enclosing **switch** statement, or **do**, **for**, or **while** loop.

continue
Skip the rest of the loop body. Evaluate the *condition* part of the nearest enclosing **do** or **while** loop, or go to the *incr* part of a **for** loop.

delete *array*[*index*]
Delete element *index* from array *array*.

delete *array*
Delete all elements from array *array*.

do *statement* **while** (*condition*)
Execute *statement* while *condition* is true. The *statement* is always executed at least once.

exit [*expression*]
Terminate input record processing. Execute the **END** rule(s) if present. If present, *expression* becomes **awk**'s return value.

for (*init*; *cond*; *incr*) *statement*
Execute *init*. Evaluate *cond*. If it is true, execute *statement*. Execute *incr* before going back to the top to re-evaluate *cond*. Any of the three may be omitted. A missing *cond* is considered to be true.

for (*var* **in** *array*) *statement*
Execute *statement* once for each subscript in *array*, with *var* set to a different subscript each time through the loop.

if (*condition*) *statement1* [**else** *statement2*]
If *condition* is true, execute *statement1*, otherwise execute *statement2*. Each **else** matches the closest **if**.

next See Input Control.

nextfile See Input Control.

switch (*expression*) {
case *constant*|*regular expression*: *statement(s)*
default: *statement(s)*
}
Switch on *expression*, execute *case* if matched, default if not. The **default** label and associated statements are optional.

while (*condition*) *statement*
While *condition* is true, execute *statement*.

{ *statements* **}** .br A list of statements enclosed in braces can be used anywhere that a single statement would otherwise be used.

ESCAPE SEQUENCES

Within strings constants ("...") and regexp constants (/.../), escape sequences may be used to generate otherwise unprintable characters. This table lists the available escape sequences.

\a	alert (bell)	**\r**	carriage return
\b	backspace	**\t**	horizontal tab
\f	form feed	**\v**	vertical tab
\n	newline	****	backslash
*****ddd*	octal value *ddd*	**\x***hh*	hex value *hh*
\"	double quote	**\/**	forward slash

RECORDS

Normally, records are separated by newline characters. Assigning values to the built-in variable **RS** controls how records are separated. If **RS** is any single character, that character separates records. Otherwise, **RS** is a regular expression. (Not Brian Kernighan's **awk**.) Text in the input that matches this regular expression separates the record. **gawk** sets **RT** to the value of the input text that matched the regular expression. The value of **IGNORECASE** also affects how records are separated when **RS** is a regular expression. If **RS** is set to the null string, then records are separated by one or more empty lines. When **RS** is set to the null string, the newline character always acts as a field separator, in addition to whatever value **FS** may have. **mawk** does not apply exceptional rules to **FS** when **RS** is set to **""**.

FIELDS

As each input record is read, **awk** splits the record into *fields*, using the value of the **FS** variable as the field separator. If **FS** is a single character, fields are separated by that character. If **FS** is the null string, then each individual character becomes a separate field. Otherwise, **FS** is expected to be a full regular expression. In the special case that **FS** is a single space, fields are separated by runs of spaces and/or tabs and/or newlines. Leading and trailing whitespace are ignored. The value of **IGNORECASE** also affects how fields are split when **FS** is a regular expression.

If the **FIELDWIDTHS** variable is set to a space-separated list of numbers, each field is expected to have a fixed width, and **gawk** splits up the record using the specified widths. Each field width may optionally be preceded by a colon-separated value specifying the number of characters to skip before the field starts. The value of **FS** is ignored. Assigning a new value to **FS** or **FPAT** overrides the use of **FIELDWIDTHS**. and restores the default behavior.

Similarly, if the **FPAT** variable is set to a string representing a regular expression, each field is made up of text that matches that regular expression. In this case, the regular expression describes the fields themselves, instead of the text that separates the fields. Assigning a new value to **FS** or **FIELDWIDTHS** overrides the use of **FPAT**.

Each field in the input record may be referenced by its position: **$1**, **$2** and so on. **$0** is the whole record. Fields may also be assigned new values.

The variable **NF** is set to the total number of fields in the input record.

References to non-existent fields (i.e., fields after **$NF**) produce the null string. However, assigning to a non-existent field (e.g., **$(NF+2) = 5**) increases the value of **NF**, creates any intervening fields with the null string as their value, and causes the value of **$0** to be recomputed with the fields being separated by the value of **OFS**. References to negative numbered fields cause a fatal error. Decreasing the value of **NF** causes the trailing fields to be lost (not Brian Kernighan's **awk**).

HISTORICAL FEATURES (gawk)

It is possible to call the **length()** built-in function not only with no argument, but even without parentheses. Doing so, however, is poor practice, and **gawk** issues a warning about its use if **--lint** is specified on the command line.

REGULAR EXPRESSIONS

Regular expressions are the extended kind originally defined by **egrep**. **gawk** supports additional GNU operators. A *word-constituent* character is a letter, digit, or underscore (_).

Summary of Regular Expressions
In Decreasing Precedence

(*r***)**	regular expression (for grouping)
c	if non-special character, matches itself
*****c	turn off special meaning of *c*
^	beginning of string (note: *not* line)
$	end of string (note: *not* line)
.	any single character, including newline
[...**]**	any one character in ... or range
[^...**]**	any one character not in ... or range
\y	word boundary
\B	middle of a word
\<	beginning of a word
\>	end of a word
\s	any whitespace character
\S	any non-whitespace character
\w	any word-constituent character
\W	any non-word-constituent character
\`	beginning of a string
\'	end of a string
*r******	zero or more occurrences of *r*
*r***+**	one or more occurrences of *r*
*r***?**	zero or one occurrences of *r*
*r***{***n***,***m***}**	*n* to *m* occurrences of *r* (POSIX: see note below)
*r1***|***r2*	*r1* or *r2*

The *r***{***n***,***m***}** notation is called an *interval expression*. Not supported by **mawk** or Brian Kernighan's **awk**.

In regular expressions, within character ranges (**[**...**]**), the notation **[[:***class***:]]** defines character classes:

alnum	alphanumeric	**lower**	lowercase
alpha	alphabetic	**print**	printable
blank	space or tab	**punct**	punctuation
cntrl	control	**space**	whitespace
digit	decimal	**upper**	uppercase
graph	non-spaces	**xdigit**	hexadecimal

ENVIRONMENT VARIABLES (gawk)

The environment variable **AWKPATH** specifies a search path to use when finding source files named with the **-f** option. The default path is **".:/usr/local/share/awk"**. If a file name given to the **-f** option contains a "/" character, no path search is performed.

The variable **AWKLIBPATH** specifies the search path for dynamic extensions to use with **@load** and the **-l** option.

For socket communication, **GAWK_SOCK_RETRIES** controls the number of connection retries, and **GAWK_MSEC_SLEEP** controls the interval between retries. The interval is in milliseconds. On systems that do not support *usleep*(3), the value is rounded up to an integral number of seconds.

The value of **GAWK_READ_TIMEOUT** specifies the time, in milliseconds, for **gawk** to wait for input before returning with an error.

If **POSIXLY_CORRECT** exists then **gawk** behaves exactly as if the **--posix** option had been given.

LOCALIZATION (gawk)

There are several steps involved in producing and running a localizable **awk** program.

1. Add a **BEGIN** action to assign a value to the **TEXTDOMAIN** variable to set the text domain for your program.

```
BEGIN { TEXTDOMAIN = "myprog" }
```

This allows **gawk** to find the **.gmo** file associated with your program. Without this step, **gawk** uses the **messages** text domain, which probably won't work.

2. Mark all strings that should be translated with leading underscores.

3. Use the **bindtextdomain()**, **dcgettext()**, and/or **dcngettext()** functions in your program, as appropriate.

4. Run

```
gawk --gen-pot -f myprog.awk > myprog.pot
```

to generate a **.pot** file for your program.

5. Provide appropriate translations, and build and install a corresponding **.gmo** file.

The internationalization features are described in full detail in *GAWK: Effective AWK Programming*.

SPECIAL FILENAMES

All three **awk** implementations recognize certain special filenames internally when doing I/O redirection from either **print** or **printf** into a file or via **getline** from a file. These filenames provide access to open file descriptors inherited from the parent process. They may also be used on the command line to name data files. The filenames are:

"-"	standard input
/dev/stdin	standard input
/dev/stdout	standard output
/dev/stderr	standard error output

The following names are specific to **gawk**.

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

/inet/tcp/*lport***/***rhost***/***rport*
/inet4/tcp/*lport***/***rhost***/***rport*
/inet6/tcp/*lport***/***rhost***/***rport*
Files for TCP/IP connections on local port *lport* to remote host *rhost* on remote port *rport*. Use a port of **0** to have the system pick a port. Use **/inet4** to force an IPv4 connection, and **/inet6** to force an IPv6 connection. Plain **/inet** uses the system default (probably IPv4). Usable only with the **|&** two-way I/O operator.

/inet/udp/*lport***/***rhost***/***rport*
/inet4/udp/*lport***/***rhost***/***rport*
/inet6/udp/*lport***/***rhost***/***rport*
Similar, but use UDP/IP instead of TCP/IP.

INPUT CONTROL

getline	Set **$0** from next record; set **NF**, **NR**, **FNR**.
getline < *file*	Set **$0** from next record of *file*; set **NF**.
getline *v*	Set *v* from next input record; set **NR**, **FNR**.
getline *v* **<** *file*	Set *v* from next record of *file*.
cmd **| getline**	Pipe into **getline**; set **$0**, **NF**.
cmd **| getline** *v*	Pipe into **getline**; set *v*.

cmd **|& getline**
Coprocess pipe into **getline**; set **$0**, **NF**.

cmd **|& getline** *v*
Coprocess pipe into **getline**; set *v*.

next
Stop processing the current input record. Read next input record and start over with the first pattern in the program. Upon end of the input data, execute any **END** rule(s).

nextfile
Stop processing the current input file. The next input record comes from the next input file. Update **FILENAME** and **ARGIND**, reset **FNR** to 1, and start over with the first pattern. At end of file, execute any **ENDFILE** and **END** rule(s).

getline returns 1 on success, zero on end of file, and −1 on an error. For retryable I/O, **getline** returns −2. All versions set **RT**. Upon an error, **ERRNO** describes the problem.

OUTPUT CONTROL

fflush([*file*]**)**
Flush any buffers associated with the open output file or pipe *file*. If no *file*, or if *file* is null, then flush all open output files and pipes.

print
Print the current record. Terminate output record with **ORS**.

print *expr-list*
Print expressions. Each expression is separated by the value of **OFS**. Terminate the output record with **ORS**.

printf *fmt*, *expr-list*
Format and print (see Printf Formats).

system(*cmd***)**
Execute the command *cmd*, and return the exit status (may not be available on non-POSIX systems).

I/O redirections may be used with both **print** and **printf**.

print "hello" > *file*
Print data to *file*. The first time the file is written to, it is truncated. Subsequent commands append data.

print "hello" >> *file*
Append data to *file*. The previous contents of *file* are not lost.

print "hello" | *cmd*
Print data down a pipeline to *cmd*.

print "hello" |& *cmd*
Print data down a pipeline to coprocess *cmd*.

CLOSING REDIRECTIONS

close(*file***)**
Close input or output file, pipe or coprocess.

close(*command*, *how***)**
Close one end of coprocess pipe. Use **"to"** for the write end, or **"from"** for the read end.

On success, **close()** returns zero for a file, or the exit status for a process. It returns −1 if *file* was never opened, or if there was a system problem. **ERRNO** describes the error.

PRINTF FORMATS

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

%c	An ASCII character
%d, **%i**	A decimal number (the integer part)
%e	A floating point number of the form **[−]d.dddddde[+−]dd**
%E	Like **%e**, but use **E** instead of **e**
%f	A floating point number of the form **[−]ddd.dddddd**
%F	Like **%f**, but use capital letters for infinity and not-a-number values.
%g	Use **%e** or **%f**, whichever is shorter, with nonsignificant zeros suppressed
%G	Like **%g**, but use **%E** instead of **%e**
%o	An unsigned octal integer
%u	An unsigned decimal integer
%s	A character string
%x	An unsigned hexadecimal integer
%X	Like **%x**, but use **ABCDEF** for 10–15
%%	A literal **%**; no argument is converted

Optional, additional parameters may lie between the **%** and the control letter:

*count***$**	Use the *count*'th argument at this point in the formatting (a *positional specifier*). For use in translated versions of format strings, not in the original text of an AWK program.
−	Left-justify the expression within its field.
space	For numeric conversions, prefix positive values with a space and negative values with a minus sign.
+	Use before the *width* modifier to always supply a sign for numeric conversions, even if the data to be formatted is positive. The **+** overrides the space modifier.
#	Use an "alternate form" for some control letters:
%o	Supply a leading zero.
%x, **%X**	Supply a leading **0x** or **0X** for a nonzero result.
%e, **%E**, **%f**	The result always has a decimal point.
%g, **%G**	Trailing zeros are not removed.
0	Pad output with zeros instead of spaces. This applies only to the numeric output formats. Only has an effect when the field width is wider than the value to be printed.
'	Use the locale's thousands separator and decimal point characters.
width	Pad the field to this width. The field is normally padded with spaces. If the **0** flag has been used, pad with zeros.
.*prec*	Precision. The meaning of the *prec* varies by control letter:
%d, **%o**, **%i**, **%u**, **%x**, **%X**	The minimum number of digits to print.
%e, **%E**, **%f**	The number of digits to print to the right of the decimal point.
%g, **%G**	The maximum number of significant digits.
%s	The maximum number of characters to print.

Use a ***** in place of either the *width* or *prec* specifications to take their values from the **printf** or **sprintf()** argument list. Use *****n**$** to use positional specifiers with a dynamic width or precision.

USER-DEFINED FUNCTIONS

Functions in AWK are defined as follows:

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

Functions execute when they are called from within expressions in either patterns or actions. Actual parameters supplied in the function call instantiate the formal parameters declared in the function. Arrays are passed by reference, other variables are passed by value.

Declare local variables 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:

```
# a and b are local
function f(p, q,     a, b)
{
        .....
}
/abc/    { ... ; f(1, 2) ; ... }
```

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

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.

Functions may be called indirectly. To do this, assign the name of the function to be called, as a string, to a variable. Then use the variable as if it were the name of a function, prefixed with an ''at'' sign, like so:

```
function myfunc()
{
        print "myfunc called"
}
{
        the_func = "myfunc"
        @the_func()
}
```

Use **return** to return a value from a function. The return value is undefined if no value is provided, or if the function returns by ''falling off'' the end.

The word **func** may be used in place of **function**. This usage is deprecated.

NUMERIC FUNCTIONS

atan2(*y***, *x*)**	The arctangent of *y*/*x* in radians.
cos(*expr***)**	The cosine of *expr*, which is in radians.
exp(*expr***)**	The exponential function ($e \hat{} x$).
int(*expr***)**	Truncate to integer.
log(*expr***)**	The natural logarithm function (base *e*).
rand()	A random number *N* such that $0 \le N < 1$.
sin(*expr***)**	The sine of *expr*, which is in radians.
sqrt(*expr***)**	The square root of *expr*.
srand([*expr***])**	Use *expr* as the new seed for the random number generator. If no *expr*, use the time of day. Return the previous seed.

STRING FUNCTIONS

asort(*s* [**,** *d* [**,** *comp*]]**)**
Sort the source array *s*, replacing the indices with numeric values 1 through *n* (the number of elements in the array), and return the number of elements. If destination *d* is supplied, copy *s* to *d*, sort *d*, and leave *s* unchanged. Use *comp* to compare indices and elements.

asorti(*s* [**,** *d* [**,** *comp*]]**)**
Like **asort()**, but sort on the indices, not the values. The original values are thrown array, so provide a second array to preserve the first.

gensub(*r***,** *s***,** *h* [**,** *t*]**)**
Search the target string *t* for matches of the regular expression *r*. If *h* is a string beginning with **g** or **G**, replace all matches of *r* with *s*. Otherwise, *h* is a number indicating which match of *r* to replace. If *t* is not supplied, use **$0** instead. Within the replacement text *s*, the sequence *****n*, where *n* is a digit from 1 to 9, indicates just the text that matched the *n*th parenthesized subexpression. The sequence **\0** represents the entire matched text, as does the character **&**. Unlike **sub()** and **gsub()**, the function returns the modified string; the original target string is *not* changed.

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**. An **&** in the replacement text is replaced with the text that was actually matched. Use **\&** to get a literal **&**. See *GAWK: Effective AWK Programming* for a fuller discussion of the rules for **&**'s and backslashes in the replacement text of **gensub()**, **sub()** and **gsub()**.

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. With an array argument, return the number of elements in the array.

match(*s***,** *r* [**,** *a*]**)**
Return the position in *s* where the regular expression *r* occurs, or zero if *r* is not present, and set the values of variables **RSTART** and **RLENGTH**. If *a* is supplied, the text matching all of *r* is placed in *a***[0]**. If there were parenthesized subexpressions, the matching texts are placed in *a***[1]**, *a***[2]**, and so on. Subscripts **a[***n***, "start"]**, and **a[***n***, "length"]** provide the starting index in the string and length, respectively, of each matching substring.

patsplit(*s***,** *a* [**,** *r* [**,** *seps*]] **)**
Split the string *s* into the array *a* and the array *seps* of separator strings using the regular expression *r*, and return the number of fields. Element values are the portions of *s* that matched *r*. The value of *seps*[*i*] is the separator that appeared in front of *a*[*i*+1]. If *r* is omitted, use **FPAT** instead. Clear the arrays *a* and *seps* first. Splitting behaves identically to field splitting with **FPAT**.

split(*s***,** *a* [**,** *r* [**,** *seps*]] **)**
Split the string *s* into the array *a* and the array *seps* of separator strings using the regular expression *r*, and return the number of fields. If *r* is omitted, use **FS** instead. Clear the arrays *a* and *seps* first. Splitting behaves identically to field splitting. (See Fields.)

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

STRING FUNCTIONS (continued)

strtonum(*s***)**
Examine *s*, and return its numeric value. If *s* begins with a leading **0**, treat it as an octal number. If *s* begins with a leading **0x** or **0X**, treat *s* as a hexadecimal number. Otherwise, treat the number as decimal.

sub(*r***, *s* [, *t*])**
Just like **gsub()**, but replace only the first matching substring.

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

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

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

TIME FUNCTIONS

gawk and **mawk** provide the following functions for obtaining time stamps and formatting them.

mktime(*datespec* [, *utc-flag*]**)**
Convert *datespec* into a time stamp of the same form as returned by **systime()** and return it. The *datespec* is a string of the form **"***YYYY MM DD HH MM SS*[*DST*]**"**. If *utc-flag* is present and is non-zero or non-null, the result is in UTC, otherwise it is in local time.

strftime([*format* [, *timestamp*[, *utc-flag*]]]**)**
Format *timestamp* according to the specification in *format*. The *timestamp* should be of the same form as returned by **systime()**. If *utc-flag* is present and is non-zero or non-null, the result is in UTC, otherwise it is in local time. If *timestamp* is missing, use the current time of day. If *format* is missing, use **PROCINFO["strftime"]**. The default value is equivalent to the output of *date*(1).

systime()
Return the current time of day as the number of seconds since the Epoch.

BIT MANIPULATION FUNCTIONS (gawk)

gawk provides the following bit manipulation functions.

and(*v1*, *v2* [, ...]**)**
Return the bitwise AND of the arguments.

compl(*val***)**
Return the bitwise complement of *val*.

lshift(*val*, *count***)**
Return the value of *val*, shifted left by *count* bits.

or(*v1*, *v2* [, ...]**)**
Return the bitwise OR of the arguments.

rshift(*val*, *count***)**
Return the value of *val*, shifted right by *count* bits.

xor(*v1*, *v2* [, ...]**)**
Return the bitwise XOR of the arguments.

DYNAMIC EXTENSIONS (gawk)

@load "*extension***"**
Dynamically load the named *extension*. This adds new built-in functions to **gawk**. The extension is loaded during the parsing of the program. See the manual for details.

TYPE FUNCTIONS (gawk)

isarray(*x***)**
Return true if *x* is an array, false otherwise.

typeof(*x***)**
Return a string indicating the type of *x*.

INTERNATIONALIZATION (gawk)

gawk provides the following functions for runtime message translation.

bindtextdomain(*directory* [, *domain*]**)**
Specify the directory where **gawk** looks for the **.gmo** files, in case they will not or cannot be placed in the "standard" locations (e.g., during testing). Return the directory where *domain* is "bound."

The default *domain* is the value of **TEXTDOMAIN**. When *directory* is the null string (**""**), **bindtextdomain()** returns the current binding for the given *domain*.

dcgettext(*string* [, *domain* [, *category*]]**)**
Return the translation of *string* in text domain *domain* for locale category *category*. The default value for *domain* is the current value of **TEXTDOMAIN**. The default value for *category* is **"LC_MESSAGES"**.

If you supply a value for *category*, it must be a string equal to one of the known locale categories. You must also supply a text domain. Use **TEXTDOMAIN** to use the current domain.

dcngettext(*string1*, *string2*, *number* [, *dom* [, *cat*]]**)**
Return the plural form used for *number* of the translation of *string1* and *string2* in text domain *dom* for locale category *cat*. The default value for *dom* is the current value of **TEXTDOMAIN**. The default for *cat* is **"LC_MESSAGES"**.

If you supply a value for *cat*, it must be a string equal to one of the known locale categories. You must also supply a text domain. Use **TEXTDOMAIN** to use the current domain.

FTP/HTTP/GIT INFORMATION

Host: **ftp.gnu.org**
File: **/gnu/gawk/gawk-4.2.1.tar.gz**
GNU **awk** (**gawk**). There may be a later version.

git clone git://github.com/onetrueawk/awk
Brian Kernighan's **awk**. This version requires an ANSI C compiler; GCC (the GNU Compiler Collection) works well.

Host: **invisible-island.net**
File: **/mawk/mawk.tar.gz**
Michael Brennan's **mawk**. Thomas Dickey now maintains it.

COPYING PERMISSIONS