NAME

pawk − pattern-directed scanning and processing language, with profiling support and other extensions

SYNOPSIS

[env **PROFILE**=*[1 | 2 | pathname]*]

pawk [**−dieharder**] [**−help**] [**−hex**] [**−ieee**] [**−profile** *1 | 2 | pathname*] [**−safe**]

[**−seed** *seed*] [**−version**]

[**−no-dieharder**] [**−no-hex**] [**−no-ieee**] [**−no-profile**] [**−no-safe**]

[**−F** *fs* | **−csv**] [**−d** *level*] [**−v** *var=value*]

[*'prog'* | **−f** *progfile*] [**−−**] [*input-file(s)*]

Long option names may be prefixed with either a single or double hyphen, and may be abbreviated to any unique leading prefix.

pawk may be built and installed under the names **pawk32**, **pawk64**, **pawk80**, and **pawk128**, corresponding to the working precisions of the IEEE 754 binary32, binary64, binary80, and binary128 formats. They provide roughly 7, 15, 19, and 34 decimal digits, respectively.

Some sites may also have **pawkd32**, **pawkd64**, and **pawkd128**, which use IEEE 754 decimal32, decimal64, and decimal128 arithmetic, exactly 7, 16, and 34 decimal digits, respectively. A later section, DECIMAL FLOATING-POINT ARITHMETIC, describes some of its features, and additional useful *awk* functions.

The program **pawk** has the precision of **pawk64**, but may not offer extended arithmetic and function support.

DESCRIPTION

pawk is an enhancement of *awk*(1) with optional profiling support, and additional command line options. Apart from profiling, timing functions, and possibly, its random number streams, in processing with default options, it behaves just like *awk*.

Awk scans each input *file* for lines that match any of a set of patterns specified literally in *prog* or in one or more files specified as **−f** *progfile*. With each pattern there can be an associated action that will be performed when a line of a *file* matches the pattern. Each line is matched against the pattern portion of every pattern−action statement; the associated action is performed for each matched pattern. The file name **−** means the standard input. Any *file* of the form *var=value* is treated as an assignment, not a filename, and is executed at the time it would have been opened if it were a filename. The option **−v** followed by *var=value* is an assignment to be done before *prog* is executed; any number of **−v** options may be present. The **−F** *fs* option defines the input field separator to be the regular expression *fs*.

The **−csv** option causes *awk* to process records using (more or less) standard comma-separated values (CSV) format.

The **−d** *level* option sets an internal flag to write debugging information on *stdout*. This option is primarily of interest to *awk* developers.

The **−dieharder** option sends a continuous stream of random bits to *stdout* for testing, or for other uses; see the later section, RANDOM NUMBER NOTES.

The **−help** option prints a usage summary on *stdout* and exits immediately.

The **−hex** option *enables* recognition in both code and data of C/C++-style hexadecimal integers (like *0x2026abcd*), and C99-style hexadecimal floating-point values (like *0x2026.abcdp−16*). They are not defined in the standard *awk* language, but this option

adds them to it. They are *essential* when you need bit-precise numbers in programs and data.

The **−ieee** option *enables* full IEEE 754 arithmetic behavior, and also the **−hex** option. This permits division by zero to produce a signed Infinity, or a NaN if the numerator is zero; preserves the sign of zero in arithmetic operations; allows gradual underflow to subnormals (if the host CPU supports it: most do); and supports overflow to signed Infinity. In standard *pawk*, the default is **−no-ieee** and **−no-hex**, but in a *pawk* build with extended function support, both are enabled.

The **−profile** option causes an execution profile to be written on completion. The following argument is *1* for output to *stdout*, *2* for output to *stderr*, and otherwise, is a pathname to the output file. It may alternatively be specified by the **PROFILE** environment variable. No profile report is written if neither the option nor the variable is specified.

The **−safe** option prevents process creation, such as with command pipelines, or *system*(cmd).

The **−seed** *seed* option immediately calls *srand*(seed) internally to initialize the random-number generator, replacing the startup default.

The **−version** option prints a version number, and build information, on *stdout* and exits immediately.

The **−no-dieharder** [default] option turns off any previous random bitstream request.

The **−no-hex** [default] option turns off recognition of hexadecimal constants.

The **−no-ieee** option [default] *disables* IEEE 754 arithmetic behavior, in particular, forcing negative zero to positive zero, and treating zero divides as a fatal error, just like the original *awk* did to diagnose such errors with a sensible message and exit with an error status, rather than allowing abrupt process termination with a possibly less helpful diagnostic.

The **−no-profile** [default] option turns off profiling.

The **−no-safe** [default] option turns off any previous **−safe** option.

The **−−** option terminates option processing, and following arguments are treated as filenames, even if they begin with a hyphen.

Illegal options produce a warning message on *stderr* and are ignored.

If *awk* is invoked without arguments, it prints a help message on *stderr* and exits with a nonzero status code.

An input line is normally made up of fields separated by white space, or by the regular expression **FS**. The fields are denoted **$1**, **$2**, . . . , while **$0** refers to the entire line. If **FS** is null, the input line is split into one field per character.

A pattern−action statement has the form:

pattern { *action* }

A missing { *action* } means print the line; a missing pattern always matches. pattern−action statements are separated by newlines or semicolons.

An action is a sequence of statements. A statement can be one of the following:

`if(` *expression* `)` *statement* [`else` *statement*]
`while(` *expression* `)` *statement*
`for(` *expression* `;` *expression* `;` *expression* `)` *statement*
`for(` *var* `in` *array* `)` *statement*
`do` *statement* `while(` *expression* `)`

```
break
continue
{ [ statement ... ] }
expression                      # commonly var = expression
print [ expression-list ] [ > expression ]
printf format [ , expression-list ] [ > expression ]
return [ expression ]
next                            # skip remaining patterns on this input line
nextfile                        # skip rest of this file, open next, start at top
delete array[ expression ]      # delete an array element
delete array                    # delete all elements of array
exit [ expression ]             # exit immediately; status is expression
```

Statements are terminated by semicolons, newlines or right braces. An empty *expression-list* stands for **$0**. String constants are quoted " ", with the usual C escapes recognized within. Expressions take on string or numeric values as appropriate, and are built using the operators **+ − * / % ^** (exponentiation), and concatenation (indicated by white space). The operators **! ++ −− += −= *= /= %= ^= > >= < <= == != ?:** are also available in expressions. Variables may be scalars, array elements (denoted *x***[***i***]**) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; that allows for a form of associative memory. Multiple subscripts such as *[i,j,k]* are permitted; the constituents are concatenated, separated by the value of **SUBSEP**.

The **print** statement prints its arguments on the standard output (or on a file if **>** *file* or **>>** *file* is present or on a pipe if **|** *cmd* is present), separated by the current output field separator, and terminated by the output record separator. *file* and *cmd* may be literal names or parenthesized expressions; identical string values in different statements denote the same open file. The **printf** statement formats its expression list according to the *format* (see *printf*(3)). The built-in function **close(***expr***)** closes the file or pipe *expr*. The built-in function **fflush(***expr***)** flushes any buffered output for the file or pipe *expr*.

The mathematical functions **atan2**, **cos**, **exp**, **log**, **sin**, and **sqrt** are built in. Other built-in functions are:

gsub(*r***,** *t* [, *s*]**)**
: Same as **sub** except that all occurrences of the regular expression are replaced; **sub** and **gsub** return the number of replacements.

index(*s***,** *t***)**
: Position in *s* where the string *t* occurs, or 0 if it does not.

int(*x***)**
: Truncate *x* to an integer value. Infinity and NaN convert to 0.

length([*v*]**)**
: Length of its argument taken as a string, number of elements in an array for an array argument, or length of **$0** if no argument.

match(*s***,** *r***)**
: Position in *s* where the regular expression *r* occurs, or 0 if it does not. The variables **RSTART** and **RLENGTH** are set to the position and length of the matched string.

rand()
: Random number uniformly distributed on [0,1), that is, $0 \le rand() < 1$. In IEEE 754 binary64 arithmetic, the smallest likely nonzero return value is 0x1p−31 (the C99 hexadecimal floating-point notation for 2^(−31) ~= 4.66e−10), and the largest likely is 1 − 0x1p−31 ~= 0.999_999_999_534. Subnormal returns are not possible. See the later section, RANDOM NUMBER NOTES, for details on the generator period, range, and reproducibility.

split(*s***,** *a* **[,** *fs***])**
Split the string *s* into array elements *a***[1]**, *a***[2]**, . . . , *a***[n]**, and return *n*. The separation is done with the regular expression *fs* or with the field separator **FS** if *fs* is not given. An empty string as field separator splits the string into one array element per character.

sprintf(*fmt***,** *expr***,** *. . .***)**
String resulting from formatting *expr*. . . according to the *printf*(3) format *fmt*.

srand([*seed***])** Set seed for *rand()* and return the previous seed. If the argument is omitted, the seed used is the value from **time**(2), the number of seconds since the Epoch (1970-01-01 00:00:00 +0000 UTC). Unlike *awk*, with *pawk*, that time is augmented with a nanosecond value, so that even frequent calls to *srand()* produce different sequences.

Each call to *srand*(same_seed) initializes the generator to an identical internal state, producing the same sequence from subsequent calls to *rand*(). Calling *srand*() in user code is optional, because *awk* itself calls *srand*(1) at program start.

sub(*r***,** *t* **[,** *s***])** Substitute *t* for the first occurrence of the regular expression *r* in the string *s*. If *s* is not given, **$0** is used.

substr(*s***,** *m* **[,** *n***])**
n-character substring of *s* that begins at position *m* counted from 1. If no *n*, use the rest of the string.

system(*cmd***)** Execute *cmd* and return its exit status. That is −1 upon error, *cmd*'s exit status upon a normal exit, 256 + *sig* upon death-by-signal, where *sig* is the number of the murdering signal, or 512 + *sig* if there was a core dump.

tolower(*str***)** Copy of *str* with all uppercase characters translated to their corresponding lowercase equivalents.

toupper(*str***)** Copy of *str* with all lowercase characters translated to their corresponding uppercase equivalents.

Standard *pawk* provides several additional functions that are useful for conversion of numbers to hexadecimal forms, library function error code access, profiling, random numbers over wide intervals, and timing:

clock() Current processor time in seconds

cpuclock() Current high-resolution timer tick count

geterrno() Current value of the C global *errno*, a sticky integer that records the most recent error from a library function. The default value is 0, meaning no error.

group(*s, n***)** Given a decimal or hexadecimal numeric string *s*, remove any spaces and underscores, then if $n > 0$, insert underscores to produce groups of *n* digits away from any (possibly implicit) base point. Return the adjusted string.

The separators improve readability of long digit strings, and using $n \le 0$ can remove the separators when *s* is to be converted to a number.

hexflt(*x***), hexfp(***x***)**
C99-style signed hexadecimal floating-point representation of *x*. This is *exact* in binary arithmetic, but may need to be rounded in decimal

arithmetic.

hexint(*x***)** C-style signed hexadecimal integer representation of *x*. This is *exact* in binary arithmetic, but may need to be rounded in decimal arithmetic.

randlog(*a, b***)** Random number logarithmically distributed in *[a, b]* (both > 0). If *a* > *b*, they are first swapped internally.

seterrno(*n***)** Save the value of the C global *errno*, set *errno* to *n*, and return the saved value. The only symbolic names provided for this purpose in *pawk* are **EDOM**, **EILSEQ**, **EINVAL**, and **ERANGE**, (three from C99, and EINVAL from POSIX), and they are *immutable*. The value 0 means no error. It has no standard symbolic name, and *errno* is set to 0 at program startup. C library functions never set *errno* to 0, but they may set it to a nonzero value from *system-dependent* symbolic names *Exxxxx* defined by inclusion in C code of the *<errno.h>* header file. There is no provenance of that new value: the name of the function that last altered *errno* is not recorded.

sleep(*s***)** Sleep for at least *s* wall-clock seconds. The normal return value is 0 if the delay was successful, but −1 if the sleep was interrupted by a signal. The delay *s* may be fractional.

strerror(*n***)** String description of *errno* value *n*.

tic() Save current high-resolution timer tick count and return it. Timer ticks increase monotonically during job execution, but system-dependent clock resolution differences, and operating system activities, make it difficult to convert timer ticks to CPU seconds or wall-clock seconds.

time() Wall-clock time in seconds since the Epoch (1970-01-01 00:00:00 +0000 UTC). The value is always a whole number.

toc() High-resolution timer ticks since last call to **tic()**.

The "function" **getline** sets **$0** to the next input record from the current input file; **getline <** *file* sets **$0** to the next record from *file*. **getline** *x* sets variable *x* instead. Finally, *cmd* **| getline** pipes the output of *cmd* into **getline**; each call of **getline** returns the next line of output from *cmd*. In all cases, **getline** returns 1 for a successful input, 0 for end of file, and −1 for an error.

Patterns are arbitrary Boolean combinations (with **! || &&**) of regular expressions and relational expressions. Regular expressions are as in *egrep*; see *grep*(1). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions, using the operators **~** and **!~**. **/***re***/** is a constant regular expression; any string (constant or variable) may be used as a regular expression, except in the position of an isolated regular expression in a pattern.

A pattern may consist of two patterns separated by a comma; in that case, the action is performed for all lines from an occurrence of the first pattern through an occurrence of the second, inclusive.

A relational expression is one of the following:

expression matchop regular-expression
expression relop expression
expression **in** *array-name*
(*expr***,** *expr***, . . .) in** *array-name*

where a *relop* is any of the six relational operators in C, and a *matchop* is either **~** (matches) or **!~** (does not match). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns **BEGIN** and **END** may be used to capture control before the first input line is read and after the last. **BEGIN** and **END** do not combine with other patterns. They may appear multiple times in a program and execute in the order they are read by *awk*.

Variable names with special meanings:

ARGC
Argument count, assignable.

ARGV
Argument array, assignable; nonnull members are taken as filenames.

CONVFMT
Conversion format used when converting numbers (default **%.6g**).

ENVIRON
Array of environment variables; subscripts are names.

FILENAME
Name of the current input file.

FNR Ordinal number of the current record in the current file.

FS Regular expression used to separate fields; also settable by option **−F** *fs*.

NF Number of fields in the current record.

NR Ordinal number of the current record.

OFMT
Output format for numbers (default **%.6g**).

OFS Output field separator (default space).

ORS Output record separator (default newline).

RLENGTH
Length of a string matched by **match**.

RS Input record separator (default newline). If empty, blank lines separate records. If more than one character long, **RS** is treated as a regular expression, and records are separated by text matching the expression.

RSTART
Start position of a string matched by **match**.

SUBSEP
Separator between multiple subscripts (default 034, ASCII FS (file separator) control character).

Functions may be defined (at the position of a pattern−action statement) thus:

function foo(a, b, c) { . . . ; return x }

Parameters are passed by value if scalar and by reference if array name; functions may be called recursively.

Parameters are local to the function; all other variables are global. Local variables may be created by providing excess parameters in the function definition. Traditionally, the first local variable in the function definition is preceded by a few extra spaces to indicate to a human that it is not a parameter.

ENVIRONMENT VARIABLES

If **POSIXLY_CORRECT** is set in the environment, then *awk* follows the POSIX rules for **sub** and **gsub** with respect to consecutive backslashes and ampersands.

Environment variables set at program startup are recorded in the *ENVIRON[]* array. Thus, *ENVIRON["PATH"]* is the shell search path.

Awk implementations differ on whether assignments made to members of the *ENVIRON[]* array are found in the environment of child processes started after the assignments. In *nawk* and *pawk*, they are *not*.

PROFILING USER AWK PROGRAMS

Pawk always accumulates profiling data, at some modest cost compared to *awk*, but does not write a profile report unless the environment variable **PROFILE** is defined, or the **-profile** option is specified. Their value determines the destination of the profile report: 1 for *stdout*, 2 for *stderr*, or a pathname, for output to that file. If the open of that file fails, then the report is written to *stderr* without complaint.

On most systems, accurate timing is provided by high-resolution hardware timer instructions; otherwise, the profiler falls back to **getrusage**(3), or if that is not available, to the low-resolution **clock**(3) function.

EXAMPLES

```
length($0) > 72
```
Print lines longer than 72 characters.

```
{ print $2, $1 }
```
Print first two fields in opposite order.

```
BEGIN { FS = ",[ \t]*|[ \t]+" }
      { print $2, $1 }
```
Same, with input fields separated by comma and/or spaces and tabs.

```
      { s += $1 }
END   { print "sum is", s, " average is", s/NR }
```
Add up first column, print sum and average.

```
/start/, /stop/
```
Print all lines between start/stop pairs.

```
BEGIN {     # Simulate echo(1)
      for (i = 1; i < ARGC; i++) printf "%s ", ARGV[i]
      printf "\n"
      exit }
```

```
BEGIN { pi1000str = sprintf("%.36g", 2000 * atan2(1,0));
        for (g = -1; g < 6; ++g)
            printf("2d: %s", g, group(pi1000str, g)) }
-1: 3141.592653589793238462643383279502892
 0: 3141.592653589793238462643383279502892
 1: 3_1_4_1.5_9_2_6_5_3_5_8_9_7_9_3_2_3_8_4_6_2_6_4_3_3_8_3_2_7_9_5_0_2_9_2
 2: 31_41.59_26_53_58_97_93_23_84_62_64_33_83_27_95_02_92
 3: 3_141.592_653_589_793_238_462_643_383_279_502_92
 4: 3141.5926_5358_9793_2384_6264_3383_2795_0292
 5: 3141.59265_35897_93238_46264_33832_79502_92
```
Demonstrate digit grouping using *pawk128*.

```
function report(name, x) { printf("%-24s  %-23s  %s0,
                           name, ftoh(x), hexflt(x)) }

BEGIN { report("qnan("0xcafefacefeed")", qnan("0xcafefacefeed"));
        report("infty()", infty());
        report("MAXNORMAL", MAXNORMAL);
```

```
        report("1", 1);
        report("1/2", 1/2);
        report("MINNORMAL", MINNORMAL);
        report("MAXSUBNORMAL", MAXSUBNORMAL);
        report("MINSUBNORMAL", MINSUBNORMAL);
        report("0", 0);
        report("-0", -0);
        report("-infty()", -infty());
        report("-qnan("0xfacadedecade")", -qnan("0xfacadedecade")) }
```

```
qnan("0xcafefacefeed")    0xfff8_cafe_face_feed    +NaN
infty()                   0x7ff0_0000_0000_0000    +Inf
MAXNORMAL                 0x7fef_ffff_ffff_ffff    +0x1.fffffffffffffp+1023
1                         0x3ff0_0000_0000_0000    +0x1p+0
1/2                       0x3fe0_0000_0000_0000    +0x1p-1
MINNORMAL                 0x0010_0000_0000_0000    +0x1p-1022
MAXSUBNORMAL              0x000f_ffff_ffff_ffff    +0x1.ffffffffffffep-1023
MINSUBNORMAL              0x0000_0000_0000_0001    +0x1p-1074
0                         0x0000_0000_0000_0000    +0x0p+0
-0                        0x8000_0000_0000_0000    -0x0p+0
-infty()                  0xfff0_0000_0000_0000    -Inf
-qnan("0xfacadedecade")   0x7ff8_faca_dede_cade    +NaN
```

Sample storage encodings of particular IEEE 754 binary64 numbers on an x86_64 system with the extended function support version of *pawk*.

NUMBERS IN AWK

Computer hardware, and most programming languages, normally support *two* kinds of numbers, integers (. . . , −3, −2, −1, 0, 1, 2, 3, . . .) used for counting, and floating-point values (0.5, −1.5e−3, +2.75e+200, . . .) that have an integer part and a fractional part, either of which may be absent, with an exponent for the multiplying power of the floating-point base, which is 2 on modern systems.

In IEEE 754 binary64 arithmetic, floating-point numbers consist of a 1-bit sign (0 for positive, and 1 for negative), an 11-bit biased exponent of 2, and a 53-bit significand, the highest-order bit of which is implicit, and not stored.

The binary64 representable nonzero number range is [0x1p−1074, 0x1.ffff_ffff_ffff_fp+1023], or about [4.95e−324, 1.79e+308]. Magnitudes smaller than 0x1p−1022 (about 2.22e−308) are *subnormal*, and they are recognized by their zero biased exponent. For them, the implicit high-order bit is 0, instead of 1, and their nonzero significands have from 1 to 52 leading 0-bits. Thus, subnormal numbers have *reduced precision* compared to normal numbers. Subnormals support *gradual underflow*, which is needed to preserve certain important computational properties.

With normal numbers, multiplying or dividing by powers of the base is an *exact* operation if the result neither underflows nor overflows, because only the exponent is adjusted: the significand is unchanged. With subnormals, however, the exponent is fixed, so upward scaling by powers of the base is exact, but downward scaling may require rounding.

Integer arithmetic is *exact*, as long as the operands and result are in the representable range. Floating-point arithmetic may also be exact, unless it requires more bits than are available in the significand, in which case, the result must be rounded to an *inexact* value.

It is important to understand that humans normally work with decimal arithmetic, but computers usually work with binary arithmetic. Base-2 numbers can always be exactly

represented in base 10, but the reverse is **not** true. For example, 1/8 = 0.125 is exactly representable in both bases. However, 0.1 is exact in base 10, but has an infinite expansion in base 2, +0x1.999...p−4, and thus must be rounded, which in binary64 arithmetic becomes +0x1.9999_9999_9999_ap−4.

Awk differs from most programming languages, because it has only one kind of number, represented internally as a floating-point number. In particular, that means that there is no difference in *awk* between 1 and 1.0, whereas most programming languages interpret those values as an integer and a floating-point value.

Because the *awk* default significand precision of 53 bits is large enough to represent at least 15 decimal digits, for most applications, you can count things exactly in *awk*, but you can also handle very large and very small magnitudes as floating-point values that may be exact, or inexact.

When rounding of numeric results is required, arithmetic in *awk* normally uses the IEEE 754 default of *round-to-nearest, with ties to even*. If *pawk* is built with optional extended function support, all four IEEE 754 rounding modes are accessible. Math library functions are unlikely to obey alternate rounding modes, so, for example, setting rounding directions to +/−Infinity in two calls to **f(***x***)** will likely *not* produce correct interval bounds on the function value.

The later section, UNUSUAL FLOATING-POINT VALUES, discusses some other features of IEEE 754 arithmetic that can affect computations, and input and output, in *awk.*

RANDOM NUMBER NOTES

With the **−dieharder** option, and if desired, the **−seed** *seed* option to select a nondefault random-number sequence, *pawk* outputs a continuous stream of random bits on *stdout* for input to the *dieharder*(1) tool, or any similarly intended program, for testing the quality of its internal random-number generator, instead of doing its usual job of pattern matching and actions on input data. The infinite loop in *pawk* is only terminated by the exit of a down-pipeline process, or resource exhaustion.

On a 5 GHz Intel Ultra 7 CPU, the maximal Dieharder test run

```
pawk --dieharder | dieharder -a -g 20
```

takes about 15 minutes wall-clock time, and produces a report of about 125 lines.

Historical generators used in earlier *awk* implementations have been found to be deficient in at least five areas: (i) insufficient randomness; (ii) correlations between successive random numbers; (iii) too short, and undocumented, periods; (iv) limited sequence reproducibility across platforms; and (v) undocumented algorithms, making it impractical, or impossible, to reproduce *awk* computations in other programming languages.

Published articles on random number generation have recommended using no more stream values than the cube root of the period. Thus, with the period of 2^31 − 1 of many older generators, no more than 1290 random values are advisable.

This version of *pawk* solves all five problems with a new Bays−Durham shuffle-buffer generator that passes multiple validation tests on platforms with several operating systems and CPU types, has a period of about 3.3e+583, and uses the POSIX-standard *rand48* generator family that is required to be reproducible on all platforms.

In addition, for n choices of seed and a sequence of L values from a generator with period P, it has been shown that the probability of sequence overlap does not exceed *n^2 * L / P*. Thus, even with the high values $n = L =$ *1e+100*, that probability is only about 3.0e−284, so sequence overlap for different seeds will *not* be observed. Different starting seeds can then be used to run strictly independent simulations of computer models written in *awk*.

While uniform distributions of random numbers are commonly used for sampling, it is important to note that floating-point numbers are only uniform when they share a

common exponent. Otherwise, a logarithmic distribution is needed to sample a numeric range, such as with this simplified function, which omits critical sanity checks on its arguments:

```
function simple_randlog(a, b)
{   # return random value logarithmically distributed in [a,b]
    return (a * exp(rand() * (log(b) - log(a))))
}
```

The arguments must be positive, with $0 < a < b$. Although the uniform random number stream from *rand*() is platform independent, results from the exponential and logarithm functions may differ slightly across operating systems and CPUs, producing system-dependent differences in the results from `simple_randlog(a, b)` and the built-in `randlog(a, b)`.

When *pawk* is built with different working precisions, the random number sequence from *rand*() is identical for all of them, although for *pawk32* there are only 24 significand bits. *Pawk* does so by always sampling four successive 31-bit random integers that are concatenated to a long integer, and then rounded to form the working-precision significand.

EXTENDED ARITHMETIC AND FUNCTION SUPPORT

The *awk* language was initially targeted at simple text filtering applications. Its arithmetic uses the default C language datatype *double*, with a subset of the C numerical operators, a power operator, and just nine numerical functions:

```
atan2, cos, exp, int, log, rand, sin, sqrt, srand.
```

There is/are *no* access to library error codes, *no* support for bitwise operations on unsigned integers, *no* hexadecimal constants, *no* rounding mode control, *no* numerical exception handling, *no* other floating-point types (some systems have up to four binary, and three decimal), *no* complex arithmetic, *no* conversions between floating-point types, *no* access to hardware representations of numbers, *nor* to most of the roughly 100 other numerical functions defined in C89 and C99, each of them available for all of the floating-point types, and distinguished by suffix letters on their names.

If numerical output from other programs is processed by *awk* software, those omissions are a serious impediment, especially for data in higher precisions, or in hexadecimal, which is needed to represent binary floating-point numbers exactly.

When *pawk* is compiled with optional extended function support, and targeted at any or all of the floating-point precisions in C (binary32, binary64, and on most systems, either binary80 and/or binary128), it supplies interfaces to all of the C89 and C99 numerical functions that do not require pointer arguments, and provides alternatives for those that do. The functions are:

```
abs, acos, acosh, asin, asinh, atan, atan2, atanh, binom,
cbrt, ceil, copysign, cos, cosh, erf, erfc, exp, expm1, exp2,
exp2m1, exp10, exp10m1, fabs, fac1, fac2, fac3, fac4, facn,
fdim, feclearexcept, fegetround, feraiseexcept, fesetround,
fetestexcept, floor, fma, fmax, fmin, fmod, fpclassify, hypot,
ilogb, isfinite, isgreater, isgreaterequal, isinf, isless,
islessequal, islessgreater, isnan, isnormal, isunordered, j0,
j1, jn, ldexp, lgamma, llrint, llround, log, log1p, log2,
log21p, log10, log101p, logb, lrint, lround, nearbyint,
nextafter, poch, pow, remainder, rint, round, rsqrt, scalbln,
scalbn, signbit, sin, sinh, sqrt, tan, tanh, tgamma, trunc,
y0, y1, yn.
```

*abs(*x*)* is a synonym for *fabs(*x*)*.

The binomial coefficient function

```
binom(n, k) = n! / ((n - k)! k!),
```

is missing from the C libraries, as are the factorial functions *fac1(*n*)* through *fac4(*n*)* that compute *n!*, *n!!*, *n!!!*, and *n!!!!*, and *facn(*N, n*)* that computes the product *n * (n − N) * (n − 2*N) * ... * smallest nonzero term*. Although they can be extended mathematically to handle negative and fractional arguments, their *pawk* implementations *discard* any fractional part of their arguments. The factorial functions are computed such that their results are always correctly rounded, and are cached to avoid recomputation. Over its representable range, the binary80 version of the binomial coefficient has a measured average error of 0.70 ulps (units in the last place), and a worst-case error of 1.98 ulps. The largest error for the binary64 version is 0.68 ulps.

The *ordinary* Bessel functions, *j0*, *j1*, *jn*, *y0*, *y1*, and *yn*, are capitalized and subscripted in mathematics, where the lowercase names refer to the *spherical* Bessel functions. The ordinary Bessel functions are absent from C89 and C99, but are required by POSIX.1:2001, and have been in most Unix-family math libraries for decades.

The function names are the standard ones for the C datatype *double*, and remain *unchanged* in all floating-point precisions in *pawk*, which supplies internally any needed precision suffix on those names.

The C function *frexp(*x, &npow*)* splits *x* into a fraction in [1/2, 1) as the return value, and sets *npow* to the corresponding power of 2. The equivalent *pawk* code in binary arithmetic is:

```
f = significand(x) / 2
npow = exponent(x) + 1
```

For decimal arithmetic, change the divisor 2 to 10.

The C function *modf(*x, &intpart*)* splits *x* into a returned fractional part, and a companion integer-valued part returned via the second argument, each with the same sign as *x*. They can be easily computed in *awk* with this code:

```
xwhole = int(x)
xfract = x - xwhole
```

For bitwise operations, unsigned integers must be stored in working-precision floating-point numbers. Integers of up to 23 bits can be handled with all *pawk* versions; integers of 32 bits need at least *pawk64*, and those of 63 bits need *pawk80*. For 64 bits, *pawk128* is required. The needed operations are supplied by these functions:

```
bitand,    bitclr, bitget, bitgetfld, bitnlo, bitnlz, bitnot,
bitnto,    bitntz, bitor,  bitpop,    bitrol, bitror, bitset,
bitsetfld, bitshl, bitshr, bitxor.
```

To support them in *pawk*, numbers are converted internally between floating-point types and *64-bit* unsigned integers, *independent* of the working precision. For the bit functions, Infinity and NaN arguments are treated as zero, and negative arguments (except shift counts) are converted to positive ones. Bit positions are numbered from 0 to 63, starting at the low (rightmost) end; that is *little-endian* bit order on *all* systems. Thus, bit *n* has value *2^n* in the number.

With the compilation extension, *pawk* predefines names for several *immutable* numerical constants:

BASE

Floating-point base (normally 2).

EPSILON

Positive machine epsilon: smallest positive *e* such that *1 + e != 1*.

FE_ALL_EXCEPT FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW FE_UNDERFLOW
System-dependent values for *feclearexcept()*, *feraiseexcept()*, and *fetestexcept()*,

FE_TONEAREST FE_UPWARD FE_DOWNWARD FE_TOWARDZERO
System-dependent values for *fegetround()* and *fesetround()*.

FP_INFINITE FP_NAN FP_NORMAL FP_SUBNORMAL FP_ZERO
System-dependent return values from fpclassify(*x*).

MAXNORMAL
Largest magnitude normal number.

MINNORMAL
Smallest magnitude normal number.

MAXSUBNORMAL
Largest magnitude subnormal number.

MINSUBNORMAL
Smallest magnitude subnormal number.

PI Mathematical constant *pi*.

PI_HI
Mathematical constant *pi* (*correctly rounded* high part in working precision).

PI_LO
Mathematical constant *pi* (accurate low part in working precision). For correct rounding, compute multiples of *pi* like this: *PI_times_n = fma(n, PI_HI, n * PI_LO)*, instead of with *n * PI*.

SIGDIG
Number of significand bits in binary formats, or coefficient digits in decimal formats.

These *pawk*-specific functions supply further numerical support:

bitand(*x, y***)**
Bitwise AND of the unsigned integer values; the result of each bit pair is 1 if both are 1, and otherwise, is 0.

bitclr(*x, n***)**
Clear (set to zero) bit *n* in the value of *x*.

bitget(*x, n***)**
Get bit *n* of *x*.

bitgetfld(*x, n, width***)**
Get a field of *width* bits whose low-order bit *starts* at bit *n* in *x*, and extends to the left. If *n* is outside [0, 63], or *width < 1*, the field is 0. Bits outside field positions 0 to 63 are implicitly 0.

bitnlo(*x***)**
Number of leading 1-bits.

bitnlz(*x***)**
Number of leading 0-bits.

bitnot(*x***)**
Bitwise NOT (complement, or bit flip) of the unsigned integer value.

bitnto(*x***)**
Number of trailing 1-bits.

bitntz(*x***)**
Number of trailing 0-bits.

bitor(*x, y***)**
Bitwise OR of the unsigned integer values; the result of each bit pair is 0 if both are 0, and otherwise, is 1.

bitpop(*x***)**
Population count (number of 1-bits).

bitrol(*x, n***)**
n-bit left rotation of the unsigned integer value of *x*; a negative left bit rotation is a right rotation by |*n*| bits.

bitror(*x, n***)**
n-bit right rotation of the unsigned integer value of *x*; a negative right bit rotation is a left rotation by |*n*| bits.

bitset(*x, n***)**
Set bit *n* to 1 in the value of *x*.

bitsetfld(*x, n, width, value***)**
Insert the rightmost *width* bits of *value* into a field of *width* bits whose low-order bit *starts* at bit *n* of the value of *x*, and extends to the left.

bitshl(*x, n***)**
n-bit left shift of the unsigned integer value of *x*, supplying zero bits at the right; a negative left bit shift is a right shift by |*n*| bits. Bits shifted out of positions [0, 63] are lost.

bitshr(*x, n***)**
n-bit right shift of the unsigned integer value of *x*, supplying zero bits at the left; a negative right bit shift is a left shift by |*n*| bits. Bits shifted out of positions [0, 63] are lost.

bitxor(*x, y***)**
Bitwise XOR (exclusive OR) of the unsigned integer values; the result for each bit pair is 1 if they differ, and 0 if they are the same.

double(*x***)**
Cast *x* to C type *double*, then back to working precision. In decimal versions of *pawk*, the cast is to the decimal64 format.

exponent(*x***)**
Floating-point integer exponent (power of number base) of *x*. If *x* is finite, it can be reconstructed from *BASE* ^ *exponent(*x*)* * *significand(*x*)*. If *x* is zero, its significand is *x*, and its exponent is a negative integer of large magnitude.

ftoh(*x***)**
Convert *x* to a native hardware hexadecimal string in *platform-independent* big-endian order (sign, biased exponent, significand). That order is *identical* for a given floating-point size across all IEEE 754 platforms, *except* that the encodings of quiet and signaling NaNs depend on the CPU architecture.

hexflt(*x***), hexfp(***x***)**
Convert *x* to a signed hexadecimal floating-point string, always with a 1-digit before the significand base point, unless *x* is zero, Infinity, or NaN, in which case the sign is followed by `0x0p+0`, `Inf`, or `NaN`.

NaN payloads are not displayed, because *awk* does not recognize them in input, but they can be extracted with the *payload(*x*)* function.

The functions provide *platform-independent* exact views of binary floating-point numbers. The *%a* format item of the C *printf()* family does *not* provide cross-platform consistency, because of differing leading-digit choice, and because some systems display leading zeros for subnormal numbers, while others do not.

hexint(*x***)**
Convert *x* to a signed hexadecimal integer string.

htof(*s***)**
Convert the native hardware hexadecimal string *s* to floating-point.

infty(*[s]***)**
IEEE 754 run-time Infinity with optional payload in argument string *s* that is used only in decimal arithmetic. The function may raise *FE_DIVBYZERO* and/or *FE_OVERFLOW* exception flags.

isinf(*x***)**
Nonzero if x is an Infinity of either sign.

isqnan(*x***)**
Nonzero if x is a quiet NaN.

issame(*x, y***)**
Nonzero if *x* and *y* have identical memory bit patterns (e.g., two NaNs have same payload, or two numerically equal decimal numbers have the same quantization).

If both arguments are strings, their string values are compared. If one argument is a number, and the other a string, the string is converted to a number before the comparison.

issnan(*x***)**
Nonzero if *x* is a signaling NaN.

lg(*x***)** Same as *log2(*x*)*.

lg1p(*x***)**
Same as *log21p(*x*)*.

longdouble(*x***)**
Cast *x* to C type *long double*, then back to working precision. On systems with both binary80 and binary128 arithmetic, the first of those is *long double*, and the cast may reduce precision. In decimal versions of *pawk*, the cast is to the decimal128 format, the longest currently defined, and the cast never changes the value.

macheps(*x***)**
Generalized machine epsilon: the smallest positive *e* such that *(x + e) != x*; use *macheps(*−x*)* to find the neighboring positive value such that *(x − e) != x*; *nextafter*(x, +infty()) and *nextafter*(x, −infty()) can also compute those values.

payload(*x***)**
Payload as hexadecimal unsigned integer string if *x* is a NaN, or a decimal Infinity, else empty.

payloaderr(*x***)**
Brief string describing a NaN or decimal Infinity payload error code recorded in *x*, or an empty string if *x* contains no payload.

poch(*x, n***)**

Return the Pochhammer symbol, *(x)_n*, also called the rising factorial, defined by

```
(x)_n = x * (x + 1) * (x + 2) * ... * (x + n - 1).
```

Any fractional part of *n* is discarded. If *n* is out of range, the result is a quiet NaN.

Special cases: *(+/−0)_n = +/−0* , *(x)_0 = 1*, and *(x)_1 = x*.

Identities: *(1)_n = n!* and *(2)_n = (n + 1)!*.

For general *x*:

```
poch(x, n) = tgamma(x + n) / tgamma(n).
```

For integer *k*:

```
poch(k, n) = binom(k + n - 1, k - 1) * n!
           = (k + n - 1)! / (k - 1)!.
```

The possibly large number of multiplications in computing factorials, binomial coefficients, and Pochhammer values can introduce substantial rounding errors. To combat that problem, *pawk* computes those functions with simulated higher precision so that in most cases, the results are correctly rounded. The Pochhammer results are always within one unit in the last place of the exact value.

qnan(*[s]***)**

IEEE 754 run-time quiet NaN with optional unsigned decimal or hexadecimal payload in argument string *s*. The function may raise *FE_DIVBYZERO* and/or *FE_INVALID* and/or *FE_OVERFLOW* exception flags.

rsqrt(*x***)**

Accurate value of 1 / *sqrt(*x*)*. That operation is required to be correctly rounded in recent IEEE 754 standards work, and is already implemented in some CPU instruction sets. However, it is not yet standardized in C, or widely available in C libraries, so for now, *pawk* computes it in the highest available precision, applies a correction from a Newton−Raphson iteration, and then rounds that to working precision.

significand(*x***)**

Signed floating-point significand of *x*, with magnitude in [1, BASE) if *x* is normal. For zero, Infinity and NaN arguments, *significand(*x*)* returns *x*, preserving the sign, payload, and type of NaN, the sign of Infinity and zero, and the payload of a decimal Infinity. If *x* is normal, it can be reconstructed *exactly* from *BASE ^ exponent(*x*) * significand(*x*)*.

single(*x***)**

Cast *x* to C type *float* (single precision), then back to working precision. In decimal versions of *pawk*, the cast is to the decimal32 format.

snan(*[s]***)**

IEEE 754 run-time signaling NaN with optional unsigned decimal or hexadecimal payload in argument string *s*. The function may raise *FE_DIVBYZERO* and/or *FE_INVALID* and/or *FE_OVERFLOW* exception flags.

ulp(*a, e***)**

Error in working precision units in last place of approximate *a* versus exact *e*.

ulp32(*a, e***)**

Error in binary32 (or decimal32) units in last place of approximate *a* versus exact *e*.

ulp64(*a, e***)**
: Error in binary64 (or decimal64) units in last place of approximate *a* versus exact *e*.

ulp80(*a, e***)**
: Error in binary80 units in last place of approximate *a* versus exact *e*.

ulp128(*a, e***)**
: Error in binary128 (or decimal128) units in last place of approximate *a* versus exact *e*.

That rich set of named constants and functions gives the *pawk* programmer access to all of the features of IEEE 754 arithmetic, including number ranges and precisions, binary and decimal floating-point arithmetic, both quiet and signaling NaNs (some older CPU designs have only the quiet form), payloads of Infinity (decimal only) and NaNs, floating-point number classification, hardware representation of floating-point numbers, hexadecimal constants, rounding modes, and floating-point exception flags. Those features are *unavailable* in traditional *awk*, but all of them are useful for numerical programming. They should suffice for processing most numeric data produced by software written in other computer languages. The only significant omission in *pawk* is complex arithmetic. Arbitrary precision functions *may* be available; see the later section MULTIPLE-PRECISION FUNCTION SUPPORT.

The original *nawk* implementation checks the return values of most numerical functions, and when it finds the C *errno* global value set to a code for a domain or range error, it prints a warning message, and returns 1 as the function value. In *pawk*, when IEEE 754 arithmetic behavior is enabled, that intervention is suppressed, and the value from the called function is preserved. The arbitrary value 1 for an error cannot be changed for the original *awk* functions, but for the new functions in *pawk*, error conditions are generally reported via a quiet NaN with one of these payload values:

0xbad00 *binom()* called with invalid arguments.

0xbad10 *facn()* called with invalid arguments.

0xbad20 *htof()* called with invalid host native string.

0xbad30 *infty()* called with bad payload.

0xbad60 *macheps()* called with invalid argument.

0xbad70 *mpfr_binom()* called with invalid arguments.

0xbad80 *mpfr_facn()* called with invalid arguments.

0xbad90 *qnan() or snan()* called with bad payload.

0xbada0 *poch()* called with invalid arguments.

0xbadb0 *randlog()* called with invalid arguments.

0xbadc0 *rsqrt()* called with invalid argument.

0xbadd0 invalid string-to-number conversion.

0xbade0 *ulp*()* functions called with infinite argument(s).

0xbadf1 builtin function called with too few arguments: needs 1.

0xbadf2 builtin function called with too few arguments: needs 2.

0xbadf3 builtin function called with too few arguments: needs 3.

0xbadf4 builtin function called with too few arguments: needs 4.

The payload error numbers are small enough to fit in the 32-bit formats, and when viewed

as hexadecimal values, clearly indicate a problem. Their decimal values lie in [765_184, 765_428], and are unlikely to be helpful in that view.

The *payloaderr(*x*)* function returns a string that might clarify the meaning and source of an error code in the payload of *x*.

Unless otherwise documented, extended numerical functions in *pawk* detect unexpected NaN arguments, and return the first such as the function value, with the NaN payload intact.

The behavior of C89 and C99 functions that return NaN results is implementation, and/or CPU, dependent: payloads with all 0-bits, or all 1-bits, are common.

DECIMAL FLOATING-POINT ARITHMETIC

Because decimal floating-point arithmetic is not available on many systems, this section provides descriptions and functions that may be helpful for programmers new to decimal, or who need to move code back and forth between C and *pawk*.

Decimal arithmetic avoids user confusion from input/output conversions, provides a wider number range, and has features that make it *essential* for financial arithmetic.

There are three defined IEEE 754 decimal formats of storage sizes of 32, 64, and 128 bits (4, 8, and 16 bytes) that offer precisions of 7, 16, and 34 digits. For a given precision of p decimal digits, the largest finite numbers are a string of p 9-digits, with exponents 96, 384, and 6144. The corresponding smallest normal numbers are 1e−95, 1e−383, and 1e−6143, and the smallest subnormal numbers are 1e−101, 1e−398, and 1e−6176, with exponents decreased by $p - 1$ from those of their smallest normal counterparts.

The next longer format always has precision $2\ p + 2$ and more than double the exponent range. That means that, when a longer format is available, products of two values can be computed exactly, and without possibility of underflow or overflow, and reciprocals of tiny normal numbers cannot overflow. Reciprocals of MAXNORMAL are subnormal, with a loss of two decimal digits. Reciprocals of small normal numbers, and large subnormals, are finite, but reciprocals of small subnormals overflow to Infinity.

Finite binary floating-point numbers are stored with a sign, biased power-of-2 exponent, and a significand in [1, 2). By contrast, finite decimal floating-point numbers have a sign, flags, biased power-of-10 exponent, and a decimal *integer coefficient*. The flags allow fast detection of Infinity and two kinds of NaNs from just the high-order byte, and digits are stored such that trailing zeros can be preserved, space permitting. Thus, 7, 7.0, 7.00, 7.000, ... are distinguishable in decimal: they would not be so in binary arithmetic. The coefficients of those values are 7, 70, 700, and 7000, with unbiased exponents of 0, −1, −2, and −3 respectively. Those numbers compare *equal*, but their storage formats differ, and that can be detected with the *issame()*, *ftoh()*, and *ntos()* functions.

The exponent, or for fractional values, the number of fractional digits, is called its *quantum*, so 7 has a quantum of 0, 7.0 has a quantum of 1, and 7.00 has a quantum of 2.

Five functions available in the decimal versions of *pawk* support those distinctions:

normalize(*x***)**
: Remove trailing zero digits from the coefficient of *x*, increasing the stored exponent, so *normalize(*1.5000*)* returns the number 1.5, and *normalize(*15000000*)* returns the number +1.5e+7.

ntos(*x***)**
: Convert the decimal number *x* to a *signed* decimal string, *preserving* trailing zeros. Thus, *ntos(1.5000)* returns the string `"+1.5000"`, whereas the usual *awk* idiom of conversion via concatenation with an empty string, `"" 1.5000` would produce the string `"1.5"`.

quantize(*x, y***)**
: Return a value equal to *x*, but with the quantum of *y*. The digits of the numeric value of the second argument do not matter, only the exponent, so second arguments of 999.99, 9.00, and 0.01 have the same effect. Thus, *ntos(quantize(*1.5, 0.01*))* returns the number +1.50, and *ntos(quantize(*150, 0.001*))* returns +150.000.

quantum(*x***)**
: Return the quantum of *x*, equivalent to one unit in its last place. Thus, *quantum(*1.234*)* is 0.001, and *quantum(*1234.*)* is 1.

samequantum(*x, y***)**
: Return 1 if the two arguments have the same quantum, or equivalently, the same exponent, and 0 otherwise.

The five functions are also supplied in binary versions of *pawk*, but are less useful.

Quantization is preserved in addition and subtraction if the operands have the same quantization (the usual case in financial arithmetic) and rounding is not required, but other operations generally change quantization. For example, *ntos(*3.4 * 5.6*)*, returns +19.04, and the quantization changes from 1 to 2. Division does that too: changing the * to / in that example in decimal32 arithmetic produces +0.6071429, with a quantization of 7.

Financial arithmetic deals with currency units, and the four functions are critical tools in managing results. A common example is tax computation: find the price of an item costing 0.70 in some currency, with a tax rate of 5%. The computation is simple, 0.70 + 0.05 * 0.70 = 0.735. Tax rules normally require upward rounding of halfway cases like that one, in favor of the tax authorities, who would collect 0.04. However, in *any* precision of binary arithmetic, the computation gives 0.734999 . . . , which rounds downward to 0.73, depriving the tax department of 0.01, in violation of their rules.

In C programming with floating-point numbers, constants are suffixed with *f*, *<none>*, and *l* for the binary datatypes *float*, *double*, and *long double*. The suffixes for decimal numbers are *df*, *dd*, and *dl* (in either lettercase) for the *decimal_float*, *decimal_double*, and *decimal_long_double* types. In the *printf*(3) functions, C converts *float* arguments to *double*, which does not require a length specifier in format descriptors, and *long double* arguments need a length specifier of *L*. For the three decimal formats, the length specifiers are *H*, *DD*, and *DL*. The preferred choice for the first of those would have been *DF*, but *F* was already used as a fixed-point format descriptor.

Functions in the *printf*(3) family do not supply length suffixes on output numbers, unless the programmer put them in the format string. Length suffixes are not needed in *awk* because it works with only a single numeric data type, but *pawk* recognizes such suffixes, and discards them. It automatically strips length modifiers in user-supplied format items, and then inserts any required ones. Thus, the *pawk* programmer need not deal with type suffixes or format descriptor length modifiers.

Here is a summary of further differences between binary and decimal floating-point arithmetic:

- The *exponent()*, *ldexp()*, *scalbln()*, *scalbn()*, and *significand()* functions work as expected in both binary and decimal arithmetic, except that in decimal, their number base is 10.
- There is no hidden bit in the storage of decimal numbers.
- Memory words with all bits zero *are* floating-point zeros, but in decimal arithmetic, they correspond to *noncanonical* zeros: the coefficient is zero, but the exponent is a negative number of large magnitude. Canonical decimal zeros have zero coefficients and zero exponents. Their memory layouts are 0x2240_0000,

0x2234_0000_0000_0000, and 0x2207_c000_0000_0000_0000_0000_0000_0000 in the three formats.

- Addition and subtraction of decimal numbers with the same quantum are always exact, unless there is overflow, or the result has a different quantum. That is of particular importance for financial arithmetic, where the quantization of results must be carefully maintained.
- Decimal infinities can carry a payload, which *pawk* supports with the *infty(*s*)* function.
- The function *quantize(*x, 1.*)* returns the nearest integer to *x*.
- For our tax example, *quantize(*0.70 + 0.5 * 0.70, 0.00*)* returns the round-to-even result of 0.74, but in a round-to-odd mode, the result would be 0.73. Tax rules matter for the choice of rounding modes.
- Trailing fractional zeros do not matter in binary floating-point arithmetic, but they do in decimal arithmetic, because they affect the quantum. Decimal constants should normally be written to minimize exponents: write 5. instead of 5.0, and write 5.e1 instead of 50. or 50.0.
- Unlike in IEEE 754 binary arithmetic, where the common operations of halving and doubling are exact for normal numbers, in decimal arithmetic, they may require rounding. For example, PI and 2*(PI/2) differ by one in their final digit in the three decimal precisions, but are identical in binary arithmetic. Computer code for decimal arithmetic likely needs careful checking for such practices.
- Current hardware implementations of decimal floating-point arithmetic offer up to eight rounding modes, whereas IEEE 754 binary arithmetic, and most, but not all, hardware implementations of binary arithmetic, offer only four modes. Alas, at present, the decimal versions of the *gcc* compilers, *dgcc*, do not implement rounding control in decimal arithmetic, and compiled decimal code always uses the IEEE 754 default of *round-to-nearest, with ties to even*.

ROUND-TRIP BASE CONVERSIONS

In computers, numbers commonly undergo conversion between different number bases, such as base 2 for computer arithmetic, and base 10 for humans. It is desirable that such conversions do not lose accuracy, but the naive estimate of how many bits, or decimal digits, are needed is usually too small. The correct requirements for decimal-to-binary were proved by Goldberg (1967), and for binary-to-decimal by Matula (1968). Their results can be expressed by this *pawk* code:

```
### Return number of bits needed to ensure correct round-trip
### conversion from decimal to binary
function goldberg(ndecdig) { return ceil(ndecdig/log10(2) + 1) }

### Return number of decimal digits needed to ensure correct
### round-trip from binary to decimal
function matula(nbits) { return ceil(nbits/log2(10) + 1) }
```

For the IEEE 754 binary significands of 24, 53, 64, and 113 bits, Matula's formula requires 9, 17, 21, and 36 decimal digits to guarantee correct round-trip conversion back to binary. For the IEEE 754 decimal significands of 7, 16, and 34 digits, Goldberg's formula requires 25, 55, and 114 bits.

The discrepancies in those numbers show that correct round-trip base conversion *requires* higher than working precision. Without extra precision, a 7-decimal-digit value in decimal32 format *cannot* be correctly converted in general to a value in binary32, and vice versa.

The default precision of numeric output in decimal for almost all programming languages

is *wrong*, and *too small*: it is insufficient to convert correctly back to binary, for example, to continue a computation.

Base conversion is difficult to program correctly, and requires much higher precision arithmetic in the worst cases. Thus, conversions in the I/O libraries of most programming languages are deficient. Consequently, when numbers computed in binary are to be analyzed by other software, it is advisable to transfer them in exact hexadecimal form, avoiding the base-conversion problem entirely.

The *pawk* functions **hexflt(***x***)**, **hexfp(***x***)**, **hexint(***x***)**, and **ntos(***x***)** all produce the required number of digits for correct round-trip base conversion, and the latter also preserves quantization in decimal arithmetic.

MULTIPLE-PRECISION FUNCTION SUPPORT

When binary arithmetic versions of *pawk* are built with support for the MPFR multiple-precision function library, which also implies the extended function repertoire, the following additional functions are available:

```
mpfr_3sum, mpfr_4sum, mpfr_abs, mpfr_acos, mpfr_acosh,
mpfr_acospi, mpfr_agm, mpfr_ai, mpfr_asin, mpfr_asinh,
mpfr_asinpi, mpfr_atan, mpfr_atanh, mpfr_atanpi, mpfr_beta,
mpfr_binom, mpfr_cbrt, mpfr_ceil, mpfr_compound, mpfr_cos,
mpfr_cosh, mpfr_cospi, mpfr_cot, mpfr_coth, mpfr_csc,
mpfr_csch, mpfr_digamma, mpfr_dim, mpfr_eint, mpfr_erf,
mpfr_erfc, mpfr_exp, mpfr_exp10, mpfr_exp10m1, mpfr_exp2,
mpfr_exp2m1, mpfr_expm1, mpfr_fabs, mpfr_fac1, mpfr_fac2,
mpfr_fac3, mpfr_fac4, mpfr_facn, mpfr_floor, mpfr_fma,
mpfr_fmma, mpfr_gamma, mpfr_gamma_inc, mpfr_get_prec,
mpfr_hypot, mpfr_igamma, mpfr_igammac, mpfr_j0, mpfr_j1,
mpfr_jn, mpfr_li2, mpfr_lngamma, mpfr_log, mpfr_log10,
mpfr_log10p1, mpfr_log1p, mpfr_log2, mpfr_log2p1, mpfr_neg,
mpfr_poch, mpfr_pow, mpfr_powr, mpfr_prod_0, mpfr_prod_1,
mpfr_prod_2, mpfr_psi, mpfr_reldiff, mpfr_rint, mpfr_rootn,
mpfr_round, mpfr_roundeven, mpfr_rsqrt, mpfr_sec, mpfr_sech,
mpfr_set_prec, mpfr_sin, mpfr_sinh, mpfr_sinpi, mpfr_sqr,
mpfr_sqrt, mpfr_str_bin, mpfr_str_dec, mpfr_str_hex,
mpfr_sum_0, mpfr_sum_1, mpfr_sum_2, mpfr_tan, mpfr_tanh,
mpfr_tanpi, mpfr_trunc, mpfr_y0, mpfr_y1, mpfr_yn, mpfr_zeta.
```

MPFR does not support separate quiet and signaling NaNs, nor does it support NaN payloads. Both kinds of NaNs in working precision become a NaN inside MPFR, and payloads are lost.

The default precision in the MPFR interface from *pawk* is 256 bits, roughly 77 decimal digits.

The MPFR functions allow separate rounding modes for their results, but their *pawk* interface uses only the IEEE 754 default mode of *round to nearest, with ties to even*. That allows them to be viewed as *"exact"* values that can be compared with results from other computations carried out in *pawk*, or supplied as input data from other software.

Most of the list members are standard MPFR functions returning a high-precision result that *pawk* converts to working precision. However, *pawk* preserves the internal high-precision result that, before any subsequent call to a numerical MPFR function, can be retrieved with any of the *mpfr_str_xxx()* functions.

The *pawk* extensions to MPFR include these:

mpfr_3sum(*x, y, z***)**
Return the 3SUM operation, *x + y + z*, computed by MPFR as if to infinite precision, and then rounded to working precision.

mpfr_4sum(*w, x, y, z***)**
Return the 4SUM operation, *w + x + y + z*, computed as if to infinite precision, and then rounded to working precision.

mpfr_agm(*x, y***)**
Return the *arithmetic–geometric mean*. With suitable starting values of the arguments, the AGM can be used in several algorithms to produce quadratic convergence to function results. However, unlike the self-correcting feature of Newton–Raphson iterations, errors in AGM methods grow with each iteration, which is why higher intermediate precision available in MPFR is desirable.

mpfr_compound(*x, n***)**
Return an accurate result for the function *(1 + x)^n*. One application is compound interest growth, with a rate *x* per period taken over *n* periods.

mpfr_fac1(*n***)**
Return the single factorial *n! = n * (n − 1) * (n − 2) * ... * 2 * 1*, with *0! = 1*.

mpfr_fac2(*n***)**
Return the double factorial *n!! = n * (n − 2) * (n − 4) * ... * (2 or 1)*, with *(−1)!! = 0!! = 1*.

mpfr_fac3(*n***)**
Return the triple factorial *n!!! = n * (n − 3) * (n − 6) * ... * (3 or 2 or 1)*, with *(−2)!!! = (−1)!!! = 0!!! = 1*.

mpfr_fac4(*n***)**
Return the quadruple factorial *n!!!! = n * (n − 4) * (n − 8) * ... * (4 or 3 or 2 or 1)*, with *(−3)!!!! = (−2)!!!! = (−1)!!!! = 0!!!! = 1*.

mpfr_facn(*N, n***)**
Return the *N*-th order factorial of *n*: *n * (n − N) * (n − 2*N) * ... * (N or (N − 1) or ... or 3 or 2 or 1)*.

mpfr_fmma(*w, x, y, z***)**
Return the double fused multiply-add operation, *w * x + y * z*, with exact products, and a single rounding from the addition.

mpfr_get_prec()
Return the current MPFR precision in bits.

mpfr_poch(*x, n***)**
Return the Pochhammer symbol, *(x)_n*, defined by
`(x)_n = x * (x + 1) * (x + 2) * ... * (x + n - 1).`

mpfr_prod_0()
Set an internal static product variable to 1, and return 1.

mpfr_prod_1(*x***)**
Multiply by *x* the internal product variable originally initialized to 1 by *mpfr_prod_0()* and return the current value of the product. If that initialization has not yet happened, *pawk* detects that and sets the internal product variable to *x*.

This function allows accurate computation of products of working-precision results.

This function does *not* change the internal variable that normally holds the last MPFR result. That allows other MPFR functions to be used to compute high-

precision values without affecting the product.

mpfr_prod_2()
Copy the internal product variable to the last MPFR result variable, and return the product.

If *mpfr_prod_1()* is called again without a call to *mpfr_prod_0()*, multiplication continues with the current value of the internal product variable.

*mpfr_prod_1(*1*)* can be used to monitor a developing product without affecting the internal last MPFR value. *mpfr_prod_2()* can do the same monitoring, but it changes the internal last MPFR value.

mpfr_psi(*x***)**
Return the logarithmic derivative of the Gamma function. This is a more common name for *mpfr_digamma(*x*)*; the two names invoke the same MPFR function.

mpfr_reldiff(*x, y***)**
Return the relative difference, *(x − y) / x*.

mpfr_set_prec(*n***)**
Set the new default MPFR precision to *n* bits, and return the previous default precision.

The precision is forced into the range [256, 65_536]. That corresponds to about 79 to 19_370 decimal digits, using the Matula formula for digits needed for exact round-trip base conversion.

Changing the precision resets the internal memory of the last MPFR result to +NaN.

mpfr_str_bin()
Return a binary floating-point string with the last numerical result computed by MPFR in a call to one of the *mpfr_xxx()* functions, or if none exists, "+NaN".

mpfr_str_dec()
Return a decimal floating-point string with the last numerical result computed by MPFR in a call to one of the *mpfr_xxx()* functions, or if none exists, "+NaN".

mpfr_str_hex()
Return a hexadecimal floating-point string with the last numerical result computed by MPFR in a call to one of the *mpfr_xxx()* functions, or if none exists, "+NaN".

mpfr_sum_0()
Set an internal static sum variable to 0, and return 0.

mpfr_sum_1(*x***)**
Add *x* to the internal sum variable originally initialized to 0 by *mpfr_sum_0()* and return the current value of the sum. If that initialization has not yet happened, *pawk* detects that and sets the internal sum variable to *x*.

This function allows accurate computation of sums of working-precision results.

This function does *not* change the internal variable that normally holds the last MPFR result. That allows other MPFR functions to be used to compute high-precision values without affecting the sum.

mpfr_sum_2()
Copy the internal sum variable to the last MPFR result variable, and return the sum.

If *mpfr_sum_1()* is called again without a call to *mpfr_sum_0()*, addition continues with the current value of the internal sum variable.

UNUSUAL FLOATING-POINT VALUES

When *awk* was designed, several different floating-point arithmetic designs were on the computer market. However, current platforms have almost universally adopted IEEE 754 arithmetic, or a subset thereof. That system defines Not-a-Number (NaN) and Infinity values, and signed zeros. Zeros of either sign compare equal, but can be distinguished because 1 / −0 is −Infinity, but 1 / +0 is +Infinity. The functions *copysign()* and *signbit()* can also do that job.

NaNs arise from expressions like *sqrt(−1)*, *0 / 0*, *Inf − Inf*, numerical operations with a NaN operand, and in most cases, from numerical functions that receive a NaN argument, or generate a NaN internally.

Infinities arise from operations like *finite / 0*, *huge * huge*, *huge / tiny*, and from functions for which the exact result is either truly infinite, or overflows because the result is finite, but too large to represent in working precision.

Because *awk* uses *strtof*(3), *strtod*(3), or *strtold*(3) to convert string values to floating-point values, modern C libraries also convert strings starting with **inf** and **nan** into infinity and NaN values respectively. That led to strange results in *awk*, with something like

```
echo nancy | awk '{ print $1 + 0 }'
```

printing **nan** instead of zero.

Awk now follows the GNU implementation, *gawk*(1), and prefilters string values before attempting to convert them to numbers, as follows:

Hexadecimal values
: Hexadecimal integer values, and hexadecimal floating-point values introduced in C99, convert to zero, as they did prior to C99, unless they are enabled by the **−ieee** option.

NaN values
: The two strings **+nan** and **−nan** (case independent) convert to a quiet NaN with a default payload. *No others do*. NaNs can have signs, but the sign is hardware dependent, and otherwise has no significance.

QNaN versus SNaN
: On all but the RISC-V architecture, copying NaNs of either type, or doing arithmetic with them, and usually, passing them as arguments to functions, preserves their payloads. Arithmetic operations on SNaNs turn them into QNaNs. On RISC-V, arithmetic operations zero NaN payloads.

Infinity values
: The two strings **+inf** and **−inf** (case independent) convert to positive and negative infinity, respectively. *No others do*.

SEE ALSO

awk(1), *clock*(3), *dieharder*(1), *errno*(1,3), *gawk*(1), *getrusage*(3), *grep*(1), *hoc*(1), *mawk*(1), *mathcw*(3cw), *nawk*(1), *printf*(3), *sed*(1), *srand48*(3), *strerror*(3), *strtod*(3), *strtof*(3), *strtold*(3), and *time*(2).

A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, First edition, Addison−Wesley, 1988. ISBN 0-201-07981-X. ISBN-13 978-0-201-07981-4. Second edition, 2023. ISBN 0-13-826972-6. ISBN-13 978-0-13-826972-2.

Nelson H. F. Beebe, *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library*, Springer, 2017. ISBN 3-319-64109-3, DOI:

10.1007/978-3-319-64110-2; see particularly *Appendix D: Decimal Arithmetic*.

MPFR: The Multiple Precision Floating-Point Reliable Library, https://www.mpfr.org/.

Carter Bays and S. D. Durham, *Improving a Poor Random Number Generator*, ACM Transactions on Mathematical Software (TOMS) **2**(1) 59–64, March 1976. DOI: 10.1145/355666.355670

Carter Bays, *C364. Improving a random number generator: a comparison between two shuffling methods*, Journal of Statistical Computation and Simulation **36**(1) 57–59, May 1990. DOI: 10.1080/00949659008811264

I. Bennett Goldberg, *27 Bits Are Not Enough For 8-Digit Accuracy*, Communications of the ACM **10**(2) 105–106, February 1967, DOI: 10.1145/363067.363112

David W. Matula, *In-and-Out Conversions*, Communications of the ACM **11**(1) 47–50, January 1968, DOI: 10.1145/362851.362887

Sebastiano Vigna, *On the probability of overlap of random subsequences of pseudorandom number generators*, Information Processing Letters **158** 1–3, June 2020, Article 105939. DOI: 10.1016/j.ipl.2020.105939

BUGS

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate "" to it.

The scope rules for variables in functions are a botch; the syntax is worse.

Input is expected to be UTF-8 encoded. Other multibyte character sets are not handled. However, in eight-bit locales, *awk* treats each input byte as a separate character.