Rocky Enterprise Linux 9.2 Manual Pages on command 'sscanf.3'

$ man sscanf.3

SCANF(3)   Linux Programmer's Manual   SCANF(3)

NAME

   scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf - input format conversion

SYNOPSIS

   #include <stdio.h>

   int scanf(const char *format, ...);

   int fscanf(FILE *stream, const char *format, ...);

   int sscanf(const char *str, const char *format, ...);

   #include <stdarg.h>

   int vscanf(const char *format, va_list ap);

   int vsscanf(const char *str, const char *format, va_list ap);

   int vfscanf(FILE *stream, const char *format, va_list ap);

 Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

   vscanf(), vsscanf(), vfscanf():

      _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION

   The scanf() family of functions scans input according to format as described below. This

   format may contain conversion specifications; the results from such conversions, if any,

   are stored in the locations pointed to by the pointer arguments that follow format. Each

   pointer argument must be of a type that is appropriate for the value returned by the cor?

   responding conversion specification.

   If the number of conversion specifications in format exceeds the number of pointer argu?

   ments, the results are undefined. If the number of pointer arguments exceeds the number

of conversion specifications, then the excess pointer arguments are evaluated, but are otherwise ignored.

The scanf() function reads input from the standard input stream stdin, fscanf() reads in‐put from the stream pointer stream, and sscanf() reads its input from the character string pointed to by str.

The vfscanf() function is analogous to vfprintf(3) and reads input from the stream pointer stream using a variable argument list of pointers (see stdarg(3). The vscanf() function scans a variable argument list from the standard input and the vsscanf() function scans it from a string; these are analogous to the vprintf(3) and vsprintf(3) functions respec‐tively.

The format string consists of a sequence of directives which describe how to process the sequence of input characters. If processing of a directive fails, no further input is read, and scanf() returns. A "failure" can be either of the following: input failure, meaning that input characters were unavailable, or matching failure, meaning that the in‐put was inappropriate (see below).

A directive is one of the following:

? A sequence of white-space characters (space, tab, newline, etc.; see isspace(3)). This directive matches any amount of white space, including none, in the input.

? An ordinary character (i.e., one other than white space or '%'). This character must exactly match the next character of input.

? A conversion specification, which commences with a '%' (percent) character. A se‐quence of characters from the input is converted according to this specification, and the result is placed in the corresponding pointer argument. If the next item of input does not match the conversion specification, the conversion fails?this is a matching failure.

Each conversion specification in format begins with either the character '%' or the char‐acter sequence "%n$" (see below for the distinction) followed by:

? An optional '*' assignment-suppression character: scanf() reads input as directed by the conversion specification, but discards the input. No corresponding pointer argument is required, and this specification is not included in the count of suc‐cessful assignments returned by scanf().

? For decimal conversions, an optional quote character ('). This specifies that the input number may include thousands' separators as defined by the LC_NUMERIC cate‐

gory of the current locale. (See setlocale(3).) The quote character may precede or follow the '*' assignment-suppression character.

?   An optional 'm' character. This is used with string conversions (%s, %c, %[), and relieves the caller of the need to allocate a corresponding buffer to hold the in‐put: instead, scanf() allocates a buffer of sufficient size, and assigns the ad‐dress of this buffer to the corresponding pointer argument, which should be a pointer to a char * variable (this variable does not need to be initialized before the call). The caller should subsequently free(3) this buffer when it is no longer required.

?   An optional decimal integer which specifies the maximum field width. Reading of characters stops either when this maximum is reached or when a nonmatching charac‐ter is found, whichever happens first. Most conversions discard initial white space characters (the exceptions are noted below), and these discarded characters don't count toward the maximum field width. String input conversions store a ter‐minating null byte ('\0') to mark the end of the input; the maximum field width does not include this terminator.

?   An optional type modifier character. For example, the l type modifier is used with integer conversions such as %d to specify that the corresponding pointer argument refers to a long rather than a pointer to an int.

?   A conversion specifier that specifies the type of input conversion to be performed.

The conversion specifications in format are of two forms, either beginning with '%' or be‐ginning with "%n$". The two forms should not be mixed in the same format string, except that a string containing "%n$" specifications can include %% and %*. If format contains '%' specifications, then these correspond in order with successive pointer arguments.   In the "%n$" form (which is specified in POSIX.1-2001, but not C99), n is a decimal integer that specifies that the converted input should be placed in the location referred to by the n-th pointer argument following format.

Conversions

The following type modifier characters can appear in a conversion specification:

h   Indicates that the conversion will be one of d, i, o, u, x, X, or n and the next pointer is a pointer to a short or unsigned short (rather than int).

hh   As for h, but the next pointer is a pointer to a signed char or unsigned char.

j   As for h, but the next pointer is a pointer to an intmax_t or a uintmax_t. This

modifier was introduced in C99.

l   Indicates either that the conversion will be one of d, i, o, u, x, X, or n and the next pointer is a pointer to a long or unsigned long (rather than int), or that the conversion will be one of e, f, or g and the next pointer is a pointer to double (rather than float). Specifying two l characters is equivalent to L. If used with %c or %s, the corresponding parameter is considered as a pointer to a wide charac? ter or wide-character string respectively.

L   Indicates that the conversion will be either e, f, or g and the next pointer is a pointer to long double or the conversion will be d, i, o, u, or x and the next pointer is a pointer to long long.

q   equivalent to L. This specifier does not exist in ANSI C.

t   As for h, but the next pointer is a pointer to a ptrdiff_t. This modifier was in? troduced in C99.

z   As for h, but the next pointer is a pointer to a size_t. This modifier was intro? duced in C99.

The following conversion specifiers are available:

%   Matches a literal '%'. That is, %% in the format string matches a single input '%' character. No conversion is done (but initial white space characters are dis? carded), and assignment does not occur.

d   Matches an optionally signed decimal integer; the next pointer must be a pointer to int.

i   Matches an optionally signed integer; the next pointer must be a pointer to int. The integer is read in base 16 if it begins with 0x or 0X, in base 8 if it begins with 0, and in base 10 otherwise. Only characters that correspond to the base are used.

o   Matches an unsigned octal integer; the next pointer must be a pointer to unsigned int.

u   Matches an unsigned decimal integer; the next pointer must be a pointer to unsigned int.

x   Matches an unsigned hexadecimal integer (that may optionally begin with a prefix of 0x or 0X, which is discarded); the next pointer must be a pointer to unsigned int.

X   Equivalent to x.

f   Matches an optionally signed floating-point number; the next pointer must be a

pointer to float.

e   Equivalent to f.

g   Equivalent to f.

E   Equivalent to f.

a   (C99) Equivalent to f.

s   Matches a sequence of non-white-space characters; the next pointer must be a

pointer to the initial element of a character array that is long enough to hold the

input sequence and the terminating null byte ('\0'), which is added automatically.

The input string stops at white space or at the maximum field width, whichever oc?

curs first.

c   Matches a sequence of characters whose length is specified by the maximum field

width (default 1); the next pointer must be a pointer to char, and there must be

enough room for all the characters (no terminating null byte is added). The usual

skip of leading white space is suppressed. To skip white space first, use an ex?

plicit space in the format.

[Matches a nonempty sequence of characters from the specified set of accepted char?

acters; the next pointer must be a pointer to char, and there must be enough room

for all the characters in the string, plus a terminating null byte. The usual skip

of leading white space is suppressed. The string is to be made up of characters in

(or not in) a particular set; the set is defined by the characters between the open

bracket [character and a close bracket] character. The set excludes those char?

acters if the first character after the open bracket is a circumflex (^).   To in?

clude a close bracket in the set, make it the first character after the open

bracket or the circumflex; any other position will end the set. The hyphen charac?

ter - is also special; when placed between two other characters, it adds all inter?

vening characters to the set. To include a hyphen, make it the last character be?

fore the final close bracket. For instance, [^]0-9-] means the set "everything ex?

cept close bracket, zero through nine, and hyphen". The string ends with the ap?

pearance of a character not in the (or, with a circumflex, in) set or when the

field width runs out.

p   Matches a pointer value (as printed by %p in printf(3)); the next pointer must be a

pointer to a pointer to void.

n   Nothing is expected; instead, the number of characters consumed thus far from the

input is stored through the next pointer, which must be a pointer to int. This is
not a conversion and does not increase the count returned by the function. The as?
signment can be suppressed with the * assignment-suppression character, but the ef?
fect on the return value is undefined. Therefore %*n conversions should not be
used.

RETURN VALUE

On success, these functions return the number of input items successfully matched and as?
signed; this can be fewer than provided for, or even zero, in the event of an early match?
ing failure.

The value EOF is returned if the end of input is reached before either the first success?
ful conversion or a matching failure occurs. EOF is also returned if a read error occurs,
in which case the error indicator for the stream (see ferror(3)) is set, and errno is set
to indicate the error.

ERRORS

EAGAIN The file descriptor underlying stream is marked nonblocking, and the read operation
would block.

EBADF The file descriptor underlying stream is invalid, or not open for reading.

EILSEQ Input byte sequence does not form a valid character.

EINTR The read operation was interrupted by a signal; see signal(7).

EINVAL Not enough arguments; or format is NULL.

ENOMEM Out of memory.

ERANGE The result of an integer conversion would exceed the size that can be stored in the
corresponding integer type.

ATTRIBUTES

For an explanation of the terms used in this section, see attributes(7).

??
?Interface   ? Attribute   ? Value   ?
??
?scanf(), fscanf(),   ? Thread safety ? MT-Safe locale ?
?sscanf(), vscanf(), ?   ?   ?
?vsscanf(), vfscanf() ?   ?   ?
??

CONFORMING TO

The functions fscanf(), scanf(), and sscanf() conform to C89 and C99 and POSIX.1-2001.
These standards do not specify the ERANGE error.

The q specifier is the 4.4BSD notation for long long, while ll or the usage of L in inte?
ger conversions is the GNU notation.

The Linux version of these functions is based on the GNU libio library. Take a look at
the info documentation of GNU libc (glibc-1.08) for a more concise description.

NOTES

The 'a' assignment-allocation modifier

Originally, the GNU C library supported dynamic allocation for string inputs (as a non?
standard extension) via the a character. (This feature is present at least as far back as
glibc 2.0.) Thus, one could write the following to have scanf() allocate a buffer for an
input string, with a pointer to that buffer being returned in *buf:

```
    char *buf;
    scanf("%as", &buf);
```

The use of the letter a for this purpose was problematic, since a is also specified by the
ISO C standard as a synonym for f (floating-point input). POSIX.1-2008 instead specifies
the m modifier for assignment allocation (as documented in DESCRIPTION, above).

Note that the a modifier is not available if the program is compiled with gcc -std=c99 or
gcc -D_ISOC99_SOURCE (unless _GNU_SOURCE is also specified), in which case the a is inter?
preted as a specifier for floating-point numbers (see above).

Support for the m modifier was added to glibc starting with version 2.7, and new programs
should use that modifier instead of a.

As well as being standardized by POSIX, the m modifier has the following further advan?
tages over the use of a:

* It may also be applied to %c conversion specifiers (e.g., %3mc).

* It avoids ambiguity with respect to the %a floating-point conversion specifier (and is
   unaffected by gcc -std=c99 etc.).

BUGS

All functions are fully C89 conformant, but provide the additional specifiers q and a as
well as an additional behavior of the L and l specifiers. The latter may be considered to
be a bug, as it changes the behavior of specifiers defined in C89.

Some combinations of the type modifiers and conversion specifiers defined by ANSI C do not
make sense (e.g., %Ld). While they may have a well-defined behavior on Linux, this need

not to be so on other architectures. Therefore it usually is better to use modifiers that are not defined by ANSI C at all, that is, use q instead of L in combination with d, i, o, u, x, and X conversions or ll.

The usage of q is not the same as on 4.4BSD, as it may be used in float conversions equiv‐ alently to L.

EXAMPLES

To use the dynamic allocation conversion specifier, specify m as a length modifier (thus %ms or %m[range]). The caller must free(3) the returned string, as in the following exam‐ ple:

```
char *p;
int n;
errno = 0;
n = scanf("%m[a-z]", &p);
if (n == 1) {
    printf("read: %s\n", p);
    free(p);
} else if (errno != 0) {
    perror("scanf");
} else {
    fprintf(stderr, "No matching characters\n");
}
```

As shown in the above example, it is necessary to call free(3) only if the scanf() call successfully read a string.

SEE ALSO

getc(3), printf(3), setlocale(3), strtod(3), strtol(3), strtoul(3)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

GNU   2020-08-13   SCANF(3)