

## Contents

<b>1</b>	<b>Format specifications</b>	<b>2</b>
<b>2</b>	<b>Formatting various data-types</b>	<b>3</b>
<b>3</b>	<b>Possibilities, and things to do</b>	<b>3</b>
<b>4</b>	<b>l3str-format implementation</b>	<b>3</b>
4.1	Helpers . . . . .	4
4.2	Parsing a format specification . . . . .	5
4.3	Alignment . . . . .	6
4.4	Formatting token lists . . . . .	7
4.5	Formatting sequences . . . . .	9
4.6	Formatting integers . . . . .	10
4.7	Formatting floating points . . . . .	12
4.8	Messages . . . . .	16
4.9	Todos . . . . .	16
	<b>Index</b>	<b>16</b>

# The `l3str-format` package: formatting strings of characters\*

The L<sup>A</sup>T<sub>E</sub>X3 Project<sup>†</sup>

Released 2012/07/09

## 1 Format specifications

In this module, we introduce the notion of a string  $\langle format \rangle$ . The syntax follows that of Python's `format` built-in function. A  $\langle format specification \rangle$  is a string of the form

$$\langle format specification \rangle = [[\langle fill \rangle] \langle alignment \rangle][\langle sign \rangle][\langle width \rangle][.\langle precision \rangle][\langle style \rangle]$$

where each [...] denotes an independent optional part.

- $\langle fill \rangle$  can be any character: it is assumed to be present whenever the second character of the  $\langle format specification \rangle$  is a valid  $\langle alignment \rangle$  character.
- $\langle alignment \rangle$  can be `<` (left alignment), `>` (right alignment), `^` (centering), or `=` (for numeric types only).
- $\langle sign \rangle$  is allowed for numeric types; it can be `+` (show a sign for positive and negative numbers), `-` (only put a sign for negative numbers), or a space (show a space or a `-`).
- $\langle width \rangle$  is the minimum number of characters of the result: if the result is naturally shorter than this  $\langle width \rangle$ , then it is padded with copies of the character  $\langle fill \rangle$ , with a position depending on the choice of  $\langle alignment \rangle$ . If the result is naturally longer, it is not truncated.
- $\langle precision \rangle$ , whose presence is indicated by a period, can have different meanings depending on the type.
- $\langle style \rangle$  is one character, which controls how the given data should be formatted. The list of allowed  $\langle styles \rangle$  depends on the type.

The choice of  $\langle alignment \rangle =$  is only valid for numeric types: in this case the padding is inserted between the sign and the rest of the number.

---

\*This file describes v3940, last revised 2012/07/09.

<sup>†</sup>E-mail: [latex-team@latex-project.org](mailto:latex-team@latex-project.org)

## 2 Formatting various data-types

<hr/> <code>\tl_format:Nn</code> ★	<code>\tl_format:nn {&lt;token list&gt;} {&lt;format specification&gt;}</code>
<code>\tl_format:(cn nn)</code> ★	Converts the <i>&lt;token list&gt;</i> to a string according to the <i>&lt;format specification&gt;</i> . The <i>&lt;style&gt;</i> , if present, must be <b>s</b> . If <i>&lt;precision&gt;</i> is given, all characters of the string representation of the <i>&lt;token list&gt;</i> beyond the first <i>&lt;precision&gt;</i> characters are discarded.
<hr/> <code>\seq_format:Nn</code> ★	<code>\seq_format:Nn {&lt;sequence&gt;} {&lt;format specification&gt;}</code>
<code>\seq_format:cn</code> ★	Converts each item in the <i>&lt;sequence&gt;</i> to a string according to the <i>&lt;format specification&gt;</i> , and concatenates the results.
<hr/> <code>\int_format:nn</code> ★	<code>\int_format:nn {&lt;intexpr&gt;} {&lt;format specification&gt;}</code>
	Evaluates the <i>&lt;integer expression&gt;</i> and converts the result to a string according to the <i>&lt;format specification&gt;</i> . The <i>&lt;precision&gt;</i> argument is not allowed. The <i>&lt;style&gt;</i> can be <b>b</b> for binary output, <b>d</b> for decimal output (this is the default), <b>o</b> for octal output, <b>X</b> for hexadecimal output (using capital letters).
<hr/> <code>\fp_format:nn</code> ★	<code>\fp_format:nn {&lt;fpexpr&gt;} {&lt;format specification&gt;}</code>
	Evaluates the <i>&lt;floating point expression&gt;</i> and converts the result to a string according to the <i>&lt;format specification&gt;</i> . The <i>&lt;precision&gt;</i> defaults to 6. The <i>&lt;style&gt;</i> can be <ul style="list-style-type: none"> <li>• <b>e</b> for scientific notation, with one digit before and <i>&lt;precision&gt;</i> digits after the decimal separator, and an integer exponent, following <b>e</b>;</li> <li>• <b>f</b> for a fixed point notation, with <i>&lt;precision&gt;</i> digits after the decimal separator and no exponent;</li> <li>• <b>g</b> for a general format, which uses style <b>f</b> for numbers in the range <math>[10^{-4}, 10^{&lt;precision&gt;})</math> and style <b>e</b> otherwise.</li> </ul>

## 3 Possibilities, and things to do

- Provide a token list formatting *<style>* which keeps the last *<precision>* characters rather than the first *<precision>*.

## 4 l3str-format implementation

```

1 <*initex | package>
2 <@@=strformat>
3 <*package>
4 \ProvidesExplPackage
5   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}

```

```

6 \RequirePackage{l3str}
7 \end{package}

```

## 4.1 Helpers

```

\use:nf A simple variant.
\use:fnf
8 \cs_generate_variant:Nn \use:nn { nf }
9 \cs_generate_variant:Nn \use:nnn { fnf }
(End definition for \use:nf and \use:fnf.)

```

```

\tl_to_str:f A simple variant.
10 \cs_generate_variant:Nn \tl_to_str:n { f }
(End definition for \tl_to_str:f.)

```

```

\__str_format_if_digit:NTF Here we expect #1 to be a character with category other, or \s__stop.
11 \prg_new_conditional:Npnn \__str_format_if_digit:N #1 { TF }
12 {
13   \if_int_compare:w \c_nine < 1 #1 \exp_stop_f:
14   \prg_return_true: \else: \prg_return_false: \fi:
15 }
(End definition for \__str_format_if_digit:NTF.)

```

```

\__str_format_put:nw Put #1 after an \s__stop delimiter.
\__str_format_put:ow
\__str_format_put:fw
16 \cs_new:Npn \__str_format_put:nw #1 #2 \s__stop { #2 \s__stop #1 }
17 \cs_generate_variant:Nn \__str_format_put:nw { o , f }
(End definition for \__str_format_put:nw, \__str_format_put:ow, and \__str_format_put:fw.)

```

```

\__str_format_if_in:nNTF A copy of \__str_if_contains_char:nNTF to avoid relying on this weird internal string
\__str_format_if_in_aux:NN function.
18 \prg_new_conditional:Npnn \__str_format_if_in:nN #1#2 { TF }
19 {
20   \__str_format_if_in_aux:NN #2 #1
21   { #2 \prg_return_false: \exp_after:wN \__prg_break: \else: }
22   \__prg_break_point:
23 }
24 \cs_new:Npn \__str_format_if_in_aux:NN #1#2
25 {
26   \if_charcode:w #1 #2
27   \prg_return_true:
28   \exp_after:wN \__prg_break:
29   \fi:
30   \__str_format_if_in_aux:NN #1
31 }
(End definition for \__str_format_if_in:nNTF.)

```

## 4.2 Parsing a format specification

The goal is to parse

$$\langle \text{format specification} \rangle = [[\langle \text{fill} \rangle][\langle \text{alignment} \rangle]][\langle \text{sign} \rangle][\langle \text{width} \rangle][\langle \text{precision} \rangle][\langle \text{style} \rangle]$$

```

\__str_format_parse:n
\__str_format_parse_auxi:NN
\__str_format_parse_auxii:nN
  \_str_format_parse_auxiii:nN
  \_str_format_parse_auxiv:nwN
\__str_format_parse_auxv:nN
  \_str_format_parse_auxvi:nwN
  \_str_format_parse_auxvii:nN
\__str_format_parse_end:nwn
32 \cs_new:Npn \__str_format_parse:n #1
33 {
34   \exp_last_unbraced:Nf \__str_format_parse_auxi:NN
35   \__str_to_other:n {#1} \s_stop \s_stop {#1}
36 }
37 \cs_new:Npx \__str_format_parse_auxi:NN #1#2
38 {
39   \exp_not:N \__str_format_if_in:nNTF { < > = ^ } #2
40   { \exp_not:N \__str_format_parse_auxiii:nN { #1 #2 } }
41   {
42     \exp_not:N \__str_format_parse_auxii:nN
43     { \c_catcode_other_space_tl } #1 #2
44   }
45 }
46 \cs_new:Npn \__str_format_parse_auxii:nN #1#2
47 {
48   \__str_format_if_in:nNTF { < > = ^ } #2
49   { \__str_format_parse_auxiii:nN { #1 #2 } }
50   { \__str_format_parse_auxiii:nN { #1 ? } #2 }
51 }
52 \cs_new:Npx \__str_format_parse_auxiii:nN #1#2
53 {
54   \exp_not:N \__str_format_if_in:nNTF
55   { + - \c_catcode_other_space_tl }
56   #2
57   { \exp_not:N \__str_format_parse_auxiv:nwN { #1 #2 } ; }
58   { \exp_not:N \__str_format_parse_auxiv:nwN { #1 ? } ; #2 }
59 }
60 \cs_new:Npn \__str_format_parse_auxiv:nwN #1#2; #3
61 {
62   \__str_format_if_digit:NTF #3
63   { \__str_format_parse_auxv:nN { #1 } #2 #3 ; }
64   { \__str_format_parse_auxv:nN { #1 {#2} } #3 }
65 }
66 \cs_new:Npn \__str_format_parse_auxv:nN #1#2
67 {
68   \token_if_eq_charcode:NNTF . #2
69   { \__str_format_parse_auxvi:nwN {#1} 0 ; }
70   { \__str_format_parse_auxvii:nN { #1 { } } #2 }
71 }
72 \cs_new:Npn \__str_format_parse_auxvi:nwN #1#2; #3
73 {
74   \__str_format_if_digit:NTF #3
75   { \__str_format_parse_auxvii:nN {#1} #2 #3 ; }
76   { \__str_format_parse_auxvii:nN { #1 {#2} } #3 }

```

```

77   }
78   \cs_new:Npn \__str_format_parse_auxvii:nN #1#2
79   {
80     \token_if_eq_meaning:NNTF \s__stop #2
81     { \__str_format_parse_end:nwn { #1 ? } #2 }
82     { \__str_format_parse_end:nwn { #1 #2 } }
83   }
84   \cs_new:Npn \__str_format_parse_end:nwn #1 #2 \s__stop \s__stop #3
85   {
86     \tl_if_empty:nF {#2}
87     { \__msg_kernel_expandable_error:nnn { str } { invalid-format } {#3} }
88     #1
89   }

```

(End definition for \\_\_str\_format\_parse:n.)

### 4.3 Alignment

The 4 functions in this section receive an  $\langle body \rangle$ , a  $\langle sign \rangle$ , a  $\langle width \rangle$  and a  $\langle fill \rangle$  character (exactly one character). For non-numeric types, the  $\langle sign \rangle$  is empty and the  $\langle body \rangle$  is the (other) string we want to format. For numeric types, we wish to format  $\langle sign \rangle \langle body \rangle$  (both are other strings). The alignment types  $<$ ,  $>$  and  $\wedge$  keep  $\langle sign \rangle$  and  $\langle body \rangle$  together. The  $=$  alignment type, however, inserts the padding between the  $\langle sign \rangle$  and the  $\langle body \rangle$ , hence the need to keep those separate.

```

\__str_format_align_<:nnnN \__str_format_align_<:nnnN { \langle body \rangle } { \langle sign \rangle } { \langle width \rangle } { \langle fill \rangle }

```

Aligning “ $\langle sign \rangle \langle body \rangle$ ” to the left entails appending #4 the correct number of times. Then convert the result to a string.

```

90 \cs_new:cpn { __str_format_align_<:nnnN } #1#2#3#4
91   {
92     \use:nf { #2 #1 }
93     {
94       \prg_replicate:nn
95       { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
96       {#4}
97     }
98   }

```

(End definition for \\_\_str\_format\_align\_<:nnnN.)

```

\__str_format_align_>:nnnN \__str_format_align_>:nnnN { \langle body \rangle } { \langle sign \rangle } { \langle width \rangle } { \langle fill \rangle }

```

Aligning an “ $\langle sign \rangle \langle body \rangle$ ” to the right entails prepending #4 the correct number of times. Then convert the result to a string.

```

99 \cs_new:cpn { __str_format_align_>:nnnN } #1#2#3#4
100   {
101     \prg_replicate:nn
102     { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
103     {#4}
104     #2 #1
105   }

```

(End definition for `\_str\_format\_align_>:nnnN`.)

`\_str\_format\_align_~:nnnN`

`\_str\_format\_align_~:nnnN {<body>} {<sign>} {<width>} {<fill>}`

Centering “*<sign> <body>*” entails prepending and appending #4 the correct number of times. If the number of #4 to be added is odd, we add one more after than before.

```

106 \cs_new:cpn { \_str\_format\_align_~:nnnN } #1#2#3#4
107   {
108     \use:fnn
109     {
110       \prg_replicate:nn
111       {
112         \int_max:nn \c_zero
113         { #3 - \_str\_count\_unsafe:n { #2 #1 } - \c_one }
114         / \c_two
115       }
116       {#4}
117     }
118     { #2 #1 }
119     {
120       \prg_replicate:nn
121       {
122         \int_max:nn \c_zero
123         { #3 - \_str\_count\_unsafe:n { #2 #1 } }
124         / \c_two
125       }
126       {#4}
127     }
128   }

```

`\_str\_format\_align_=:nnnN`

`\_str\_format\_align_=:nnnN {<body>} {<sign>} {<width>} {<fill>}`

The special numeric alignment = means that we insert the appropriate number of copies of #4 between the *<sign>* and the *<body>*. Then convert the result to a string.

```

129 \cs_new:cpn { \_str\_format\_align_=:nnnN } #1#2#3#4
130   {
131     \use:nf {#2}
132     {
133       \prg_replicate:nn
134       { \int_max:nn { #3 - \_str\_count\_unsafe:n { #2 #1 } } { 0 } }
135       {#4}
136     }
137     #1
138   }

```

(End definition for `\_str\_format\_align_=:nnnN`.)

## 4.4 Formatting token lists

`\tl\_format:Nn`

`\tl\_format:cn`

`\tl\_format:nn`

Call `\_str\_format\_tl:NNnnNn` to read the parsed *<format specification>*. Then convert the result to a string.

```

139 \cs_new_nopar:Npn \tl_format:Nn { \exp_args:No \tl_format:nn }
140 \cs_generate_variant:Nn \tl_format:Nn { c }
141 \cs_new:Npn \tl_format:nn #1#2
142 {
143   \tl_to_str:f
144   {
145     \exp_last_unbraced:Nf \__str_format_tl:NNNnnNn
146     { \__str_format_parse:n {#2} }
147     {#1}
148   }
149 }

```

(End definition for `\tl_format:Nn`, `\tl_format:cn`, and `\tl_format:nn`. These functions are documented on page ??.)

```

\__str_format_tl:NNNnnNn \__str_format_tl:NNNnnNn <fill> <alignment> <sign> {<width>} {<precision>}
<style> {<token list>}

```

First check that the `<alignment>` is not `=`, and set the default alignment `?` to `<`. Place the modified information after a trailing `\s__stop` for later retrieval. Then check that there was no `<sign>`. The width will be useful later, store it after `\s__stop`. Afterwards, store the precision, and the function `\__str_range_unsafe:nnn` that will be used to extract the first `#5` characters of the string. There is a need to use the “unsafe” function, as otherwise leading spaces would get stripped by `f`-expansion. Finally, check that the `<style>` is `?` or `s`.

```

150 \cs_new:Npn \__str_format_tl:NNNnnNn #1#2#3#4#5#6
151 {
152   \token_if_eq_charcode:NNTF #2 =
153   {
154     \__msg_kernel_expandable_error:nnnn
155     { str } { invalid-align-format } {#2} {t1}
156     \__str_format_put:nw { #1 < }
157   }
158   {
159     \token_if_eq_charcode:NNTF #2 ?
160     { \__str_format_put:nw { #1 < } }
161     { \__str_format_put:nw { #1 #2 } }
162   }
163   \token_if_eq_charcode:NNTF #3 ?
164   {
165     \__msg_kernel_expandable_error:nnnn
166     { str } { invalid-sign-format } {#3} {t1}
167   }
168   \__str_format_put:nw { {#4} }
169   \tl_if_empty:nTF {#5}
170   { \__str_format_put:nw { \__str_range_unsafe:nnn { {1} {-1} } } }
171   { \__str_format_put:nw { \__str_range_unsafe:nnn { {1} {#5} } } }
172   \token_if_eq_charcode:NNTF #6 s
173   {
174     \token_if_eq_charcode:NNTF #6 ?
175     {

```



```

176         \_msg_kernel_expandable_error:nnnn
177         { str } { invalid-style-format } {#6} {t1}
178     }
179 }
180 \__str_format_tl_s:NNnnNNn
181 \s__stop
182 }
(End definition for \__str_format_tl:NNnnNNn.)

```

```

\__str_format_tl_s:NNnnNNn \__str_format_tl_s:NNnnNNn \s__stop <function> {<arguments>} {<width>}
<fill> <alignment> {<token list>}
The <function> and <arguments> are built in such a way that f-expanding <function>
{<other string>} <arguments> yields the piece of the <other string> that we want to output.
The <other string> is built from the <token list> by f-expanding \__str_to_other:n.
183 \cs_new:Npn \__str_format_tl_s:NNnnNNn #1#2#3#4#5#6#7
184 {
185     \exp_args:Nc \exp_args:Nf
186     { \__str_format_align_#6:nnnN }
187     { \exp_args:Nf #2 { \__str_to_other:n {#7} } #3 }
188     { }
189     {#4} #5
190 }
(End definition for \__str_format_tl_s:NNnnNNn.)

```

## 4.5 Formatting sequences

**\seq\_format:Nn** Each item is formatted as a token list according to the specification. First parse the  
**\seq\_format:cn** format and expand the sequence, then loop through the items. Eventually, convert to a string.

```

191 \cs_new:Npn \seq_format:Nn #1#2
192 {
193     \tl_to_str:f
194     {
195         \__str_format_seq:ff
196         { \exp_after:wN \use_i:nn \exp_after:wN \exp_stop_f: #1 }
197         { \__str_format_parse:n {#2} }
198     }
199 }
200 \cs_generate_variant:Nn \seq_format:Nn { c }
(End definition for \seq_format:Nn and \seq_format:cn. These functions are documented on page ??.)

```

**\\_\_str\_format\_seq:nn** The first argument is the contents of a seq variable. The second is a parsed <format  
**\\_\_str\_format\_seq:ff** specification>. Set up the loop.

```

201 \cs_new:Npn \__str_format_seq:nn #1#2
202 {
203     \__str_format_seq_loop:nnNn { } {#2}
204     #1
205     { ? \__str_format_seq_end:w } { }

```

```

206 }
207 \cs_generate_variant:Nn \__str_format_seq:nn { ff }
(End definition for \__str_format_seq:nn and \__str_format_seq:ff.)

```

```

\__str_format_seq_loop:nnNn \__str_format_seq_loop:nnNn {⟨done⟩} {⟨parsed format⟩} \__seq_item:n
{⟨item⟩}

```

The first argument is the result of formatting the items read so far. The third argument is a single token (`\__seq_item:n`), until we reach the end of the sequence, where `\use_none:n #3` ends the loop.

```

208 \cs_new:Npn \__str_format_seq_loop:nnNn #1#2#3#4
209 {
210   \use_none:n #3
211   \exp_args:Nf \__str_format_seq_loop:nnNn
212     { \use_nf {#1} { \__str_format_tl:NNNnnNn #2 {#4} } }
213   {#2}
214 }

```

(End definition for `\__str_format_seq_loop:nnNn`.)

`\__str_format_seq_end:w` Pick the right piece in the loop above.

```

215 \cs_new:Npn \__str_format_seq_end:w #1#2#3#4 { \use_ii:nnn #3 }
(End definition for \__str_format_seq_end:w.)

```

## 4.6 Formatting integers

`\int_format:nn` Evaluate the first argument and feed it to `\__str_format_int:nn`.

```

216 \cs_new:Npn \int_format:nn #1
217 { \exp_args:Nf \__str_format_int:nn { \int_eval:n {#1} } }
(End definition for \int_format:nn. This function is documented on page 3.)

```

`\__str_format_int:nn` Parse the *⟨format specification⟩* and feed it to `\__str_format_int:NNNnnNn`. Then convert the result to a string

```

218 \cs_new:Npn \__str_format_int:nn #1#2
219 {
220   \tl_to_str:f
221   {
222     \exp_last_unbraced:Nf \__str_format_int:NNNnnNn
223       { \__str_format_parse:n {#2} }
224     {#1}
225   }
226 }

```

(End definition for `\__str_format_int:nn`.)

```

\__str_format_int:NNNnnNn \__str_format_int:NNNnnNn ⟨fill⟩ ⟨alignment⟩ ⟨sign⟩ {⟨width⟩} {⟨precision⟩}
{⟨style⟩} {⟨integer⟩}

```

First set the default alignment ? to >. Place the modified information after a trailing `\s_stop` for later retrieval. Then check the *⟨sign⟩*: if the integer is negative, always put -. Otherwise, if the format's *⟨sign⟩* is ~, put a space (with category "other"); if it

is + put +; if it is - (default), put nothing, represented as a brace group. The width #4 will be useful later, store it after `\s__stop`. Afterwards, check that the  $\langle precision \rangle$  was absent. Finally, dispatch depending on the  $\langle style \rangle$ .

```

227 \cs_new:Npn \__str_format_int:NNNnnNn #1#2#3#4#5#6#7
228 {
229   \token_if_eq_charcode:NNTF #2 ?
230   { \__str_format_put:nw { #1 > } }
231   { \__str_format_put:nw { #1 #2 } }
232   \int_compare:nNnTF {#7} < \c_zero
233   { \__str_format_put:nw { - } }
234   {
235     \str_case:nnF {#3}
236     {
237       { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
238       { + } { \__str_format_put:nw { + } }
239     }
240     { \__str_format_put:nw { { } } }
241   }
242   \__str_format_put:nw { {#4} }
243   \tl_if_empty:nF {#5}
244   {
245     \__msg_kernel_expandable_error:nnnn
246     { str } { invalid-precision-format } {#5} {int}
247   }
248   \str_case:nnF {#6}
249   {
250     { ? } { \__str_format_int:NwnnNNn \use:n }
251     { d } { \__str_format_int:NwnnNNn \use:n }
252     { b } { \__str_format_int:NwnnNNn \int_to_binary:n }
253     { o } { \__str_format_int:NwnnNNn \int_to_octal:n }
254     { X } { \__str_format_int:NwnnNNn \int_to_hexadecimal:n }
255   }
256   {
257     \__msg_kernel_expandable_error:nnnn
258     { str } { invalid-style-format } {#6} { int }
259     \__str_format_int:NwnnNNn \use:n
260   }
261   \s__stop {#7}
262 }

```

(End definition for `\__str_format_int:NNNnnNn`.)

`\__str_format_int:NwnnNNn`      `\__str_format_int:NwnnNNn`  $\langle function \rangle$  `\s__stop`  $\{\langle width \rangle\}$   $\{\langle sign \rangle\}$   $\langle fill \rangle$   
 $\langle alignment \rangle$   $\{\langle integer \rangle\}$

Use the `format_align` function corresponding to the  $\langle alignment \rangle$ , with the following arguments:

- the string formed by combining the sign #4 with the result of converting the absolute value of the  $\langle integer \rangle$  #7 according to the conversion function #1;
- the  $\langle width \rangle$ ;

- the  $\langle fill \rangle$  character.

```

263 \cs_new:Npn \__str_format_int:NwnnNNn #1#2 \s__stop #3#4#5#6#7
264 {
265   \exp_args:Nc \exp_args:Nf
266     { \__str_format_align_#6:nnnN }
267     { #1 { \int_abs:n {#7} } }
268     {#4}
269     {#3} #5
270 }

```

(End definition for  $\backslash\_\_str\_format\_int:NwnnNNn$ .)

## 4.7 Formatting floating points

$\backslash fp\_format:nn$  Evaluate the first argument to an internal floating point number, and feed it to  $\backslash\_\_str\_format\_fp:nn$ .

```

271 \cs_new:Npn \fp_format:nn #1
272 { \exp_args:Nf \__str_format_fp:nn { \__fp_parse:n {#1} } }

```

(End definition for  $\backslash fp\_format:nn$ . This function is documented on page 3.)

$\backslash\_\_str\_format\_fp:nn$  Parse the  $\langle format\ specification \rangle$  and feed it to  $\backslash\_\_str\_format\_fp:NNNnnNw$ . Then convert the result to a string

```

273 \cs_new:Npn \__str_format_fp:nn #1#2
274 {
275   \tl_to_str:f
276   {
277     \exp_last_unbraced:Nf \__str_format_fp:NNNnnNw
278     { \__str_format_parse:n {#2} }
279     #1
280   }
281 }

```

(End definition for  $\backslash\_\_str\_format\_fp:nn$ .)

$\backslash\_\_str\_format\_fp:NNNnnNw$   $\backslash\_\_str\_format\_fp:NNNnnNw \langle fill \rangle \langle alignment \rangle \langle format\ sign \rangle \{ \langle width \rangle \} \{ \langle precision \rangle \} \langle style \rangle \backslash s\_fp \backslash\_\_fp\_chk:w \langle fp\ type \rangle \langle fp\ sign \rangle \langle fp\ body \rangle ;$

First set the default alignment ? to >. Place the modified information after a trailing  $\backslash s\_stop$  for later retrieval. Then check the  $\langle format\ sign \rangle$  and the  $\langle fp\ sign \rangle$ : if the floating point is negative, always put -. Otherwise (including nan), if the format's  $\langle sign \rangle$  is ~, put a space (with category "other"); if it is + put +; if it is - (default), put nothing, represented as a brace group. The width #4 will be useful later, store it after  $\backslash s\_stop$ . Afterwards, check the  $\langle precision \rangle$ : if it was not given, replace it by 6 (default precision). Finally, dispatch depending on the  $\langle style \rangle$ .

```

282 \cs_new:Npn \__str_format_fp:NNNnnNw
283   #1#2#3#4#5#6 \s__fp \__fp_chk:w #7 #8
284 {
285   \token_if_eq_charcode:NNTF #2 ?
286   { \__str_format_put:nw { #1 > } }
287   { \__str_format_put:nw { #1 #2 } }

```

```

288 \token_if_eq_meaning:NNTF 2 #8
289 { \__str_format_put:nw { - } }
290 {
291   \str_case:nnF {#3}
292   {
293     { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
294     { + } { \__str_format_put:nw { + } }
295   }
296   { \__str_format_put:nw { { } } }
297 }
298 \__str_format_put:nw { {#4} }
299 \tl_if_empty:nTF {#5}
300 { \__str_format_put:nw { { 6} } }
301 { \__str_format_put:nw { {#5} } }
302 \str_case:nnF {#6}
303 {
304   { e } { \__str_format_fp:wnnnNNw \__str_format_fp_e:wn }
305   { f } { \__str_format_fp:wnnnNNw \__str_format_fp_f:wn }
306   { g } { \__str_format_fp:wnnnNNw \__str_format_fp_g:wn }
307   { ? } { \__str_format_fp:wnnnNNw \__str_format_fp_g:wn }
308 }
309 {
310   \__msg_kernel_expandable_error:nnnn
311   { str } { invalid-style-format } {#6} { fp }
312   \__str_format_fp:wnnnNNw \__str_format_fp_g:wn
313 }
314 \s__stop
315 \s__fp \__fp_chk:w #7 #8
316 }

```

(End definition for \\_\_str\_format\_fp:NNnnNNw.)

```

\__str_format_fp:wnnnNNw \__str_format_fp:wnnnNNw <formatting function> \s__stop {<precision>}
{<width>} {<sign>} {<fill>} <alignment> \s__fp \__fp_chk:w <fp type> <fp sign>
<fp body> ;

```

```

317 \cs_new:Npn \__str_format_fp:wnnnNNw
318 #1 \s__stop #2 #3 #4 #5#6 #7 ;
319 {
320   \exp_args:Nc \exp_args:Nf
321   { \__str_format_align_#6:nnnN }
322   { #1 #7 ; {#2} }
323   {#4}
324   {#3} #5
325 }

```

(End definition for \\_\_str\_format\_fp:wnnnNNw.)

\\_\_str\_format\_fp\_round:wn Round the given floating point (not its absolute value, to play nicely with unusual rounding modes).

```

326 \cs_new:Npn \__str_format_fp_round:wn #1 ; #2
327 { \__fp_parse:n { round ( #1; , #2 - \__fp_exponent:w #1; ) } }

```

(End definition for \\_str\_format\_fp\_round:wn.)

\\_str\_format\_fp\_e:wn     With the **e** type, first filter out special cases. In the normal case, round to #4+1 significant  
\\_str\_format\_fp\_e\_aux:wn   figures (one before the decimal separator, #4 after).

```

328 \group_begin:
329 \char_set_catcode_other:N E
330 \tl_to_lowercase:n
331 {
332   \group_end:
333   \cs_new:Npn \_str_format_fp_e:wn \s__fp \_fp_chk:w #1#2#3 ; #4
334   {
335     \int_case:nnF {#1}
336     {
337       {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } e 0 }
338       {2} { inf }
339       {3} { nan }
340     }
341     {
342       \exp_last_unbraced:Nf \_str_format_fp_e_aux:wn
343       \_str_format_fp_round:wn \s__fp \_fp_chk:w #1#2#3 ; { #4 + 1 }
344       {#4}
345     }
346   }
347   \cs_new:Npn \_str_format_fp_e_aux:wn
348   \s__fp \_fp_chk:w #1#2 #3 #4#5#6#7 ; #8
349   {
350     \_str_format_put:fw { \int_eval:n { #3 - 1 } }
351     \_str_format_put:nw { e }
352     \int_compare:nNnTF {#8} > \c_sixteen
353     {
354       \_str_format_put:fw { \prg_replicate:nn { #8 - \c_fifteen } {0} }
355       \_str_format_put:fw { \use_none:n #4#5#6#7 }
356     }
357     {
358       \_str_format_put:fw
359       { \str_range:nnn { #4#5#6#7 0 } { 2 } { #8 + 1 } }
360     }
361     \_str_format_put:fw { \use_i:nnnn #4 . }
362     \use_none:n \s__stop
363   }
364 }

```

(End definition for \\_str\_format\_fp\_e:wn.)

\\_str\_format\_fp\_f:wn     With the **f** type, first filter out special cases. In the normal case, round to #4 (absolute)  
\\_str\_format\_fp\_f\_aux:wwwn   decimal places.

```

365 \cs_new:Npn \_str_format_fp_f:wn \s__fp \_fp_chk:w #1#2#3 ; #4
366 {
367   \int_case:nnF {#1}
368   {

```

```

369     {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } }
370     {2} { inf }
371     {3} { nan }
372   }
373   {
374     \exp_last_unbraced:Nf \__str_format_fp_f_aux:wwwn
375     \fp_to_decimal:n
376     { abs ( round ( \s__fp \__fp_chk:w #1#2#3 ; , #4 ) ) }
377     . . ;
378     {#4}
379   }
380 }
381 \cs_new:Npn \__str_format_fp_f_aux:wwwn #1 . #2 . #3 ; #4
382 {
383   \use:nf
384   { #1 . #2 }
385   { \prg_replicate:nn { #4 - \__str_count_unsafe:n {#2} } {0} }
386 }
(End definition for \__str_format_fp_f:wn.)

```

`\__str_format_fp_g:wn`  
`\__str_format_fp_g_aux:wn`

With the `g` type, first filter out special cases. In the normal case, round to `#4` significant figures, then test the exponent: if  $-4 \leq \langle exponent \rangle < \langle precision \rangle$ , use the presentation type `f`, otherwise use the presentation type `e`. Also, a  `$\langle precision \rangle$`  of 0 is treated like a precision of 1. Actually, we don't reuse the `e` and `f` auxiliaries, because we want to trim trailing zeros. Thankfully, this is done by `\fp_to_decimal:n` and `\fp_to_scientific:n`, acting on the (absolute value of the) rounded value.

```

387 \cs_new:Npn \__str_format_fp_g:wn \s__fp \__fp_chk:w #1#2 ; #3
388 {
389   \int_case:nnF {#1}
390   {
391     {0} { 0 }
392     {2} { inf }
393     {3} { nan }
394   }
395   {
396     \exp_last_unbraced:Nf \__str_format_fp_g_aux:wn
397     \__str_format_fp_round:wn \s__fp \__fp_chk:w #1#2 ;
398     { \int_max:nn {1} {#3} }
399     { \int_max:nn {1} {#3} }
400   }
401 }
402 \cs_new:Npn \__str_format_fp_g_aux:wn #1; #2
403 {
404   \int_compare:nNnTF { \__fp_exponent:w #1; } < { -3 }
405   { \fp_to_scientific:n }
406   {
407     \int_compare:nNnTF { \__fp_exponent:w #1; } > {#2}
408     { \fp_to_scientific:n }
409     { \fp_to_decimal:n }

```

```

410     }
411     { \_fp_set_sign_o:w 0 #1; @ \prg_do_nothing: }
412 }
(End definition for \_str_format_fp_g:wn.)

```

## 4.8 Messages

All of the messages are produced expandably, so there is no need for an extra-text.

```

413 \_msg_kernel_new:nnn { str } { invalid-format }
414   { Invalid-format~'#1'. }
415 \_msg_kernel_new:nnn { str } { invalid-align-format }
416   { Invalid-alignment~'#1'~for~type~'#2'. }
417 \_msg_kernel_new:nnn { str } { invalid-sign-format }
418   { Invalid-sign~'#1'~for~type~'#2'. }
419 \_msg_kernel_new:nnn { str } { invalid-precision-format }
420   { Invalid-precision~'#1'~for~type~'#2'. }
421 \_msg_kernel_new:nnn { str } { invalid-style-format }
422   { Invalid-style~'#1'~for~type~'#2'. }

```

## 4.9 Todos

- Check what happens during floating point formatting when a number is rounded to 0 or  $\infty$ . I think the e and f types break horribly.

```

423 </initex | package>

```

# Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
\_fp_chk:w . . . . .	283, 315, 333, 343, 348, 365, 376, 387, 397
\_fp_exponent:w . . . . .	327, 404, 407
\_fp_parse:n . . . . .	272, 327
\_fp_set_sign_o:w . . . . .	411
\_msg_kernel_expandable_error:nnn .	87
\_msg_kernel_expandable_error:nnnn . . . . .	154, 165, 176, 245, 257, 310
\_msg_kernel_new:nnn . . . . .	413, 415, 417, 419, 421
\_prg_break: . . . . .	21, 28
\_prg_break_point: . . . . .	22
\_str_count_unsafe:n . . . . .	95, 102, 113, 123, 134, 385
\_str_format_align<:nnnN . . . . .	90
\_str_format_align>:nnnN . . . . .	99
\_str_format_align^:nnnN . . . . .	106
\_str_format_fp:NNnnNw . . . . .	277, 282, 282
\_str_format_fp:nn . . . . .	272, 273, 273
\_str_format_fp:wnnnNw . . . . .	304, 305, 306, 307, 312, 317, 317
\_str_format_fp_e:wn . . . . .	304, 328, 333
\_str_format_fp_e_aux:wn . . . . .	328, 342, 347
\_str_format_fp_f:wn . . . . .	305, 365, 365
\_str_format_fp_f_aux:wwn . . . . .	365, 374, 381
\_str_format_fp_g:wn . . . . .	306, 307, 312, 387, 387
\_str_format_fp_g_aux:wn . . . . .	387, 396, 402



\_str_format_fp_round:wn	112, 122, 232
..... 326, 326, 343, 397	\char_set_catcode_other:N
\_str_format_if_digit:N	329
\_str_format_if_digit:NTF	..... 8, 9, 10, 17, 140, 200, 207
\_str_format_if_in:nN	\cs_new:cpn
\_str_format_if_in:nNTF	90, 99, 106, 129
\_str_format_if_in_aux:NN	\cs_new:Npn
\_str_format_int:NNNnnNn	. 16, 24, 32, 46, 60, 66, 72, 78, 84,
\_str_format_int:NwnnNNn	141, 150, 183, 191, 201, 208, 215,
..... 250, 251, 252, 253, 254, 259, 263, 263	216, 218, 227, 263, 271, 273, 282,
\_str_format_int:nn	317, 326, 333, 347, 365, 381, 387, 402
\_str_format_parse:n	\cs_new:Npx
..... 32, 32, 146, 197, 223, 278	\cs_new_nopar:Npn
\_str_format_parse_auxi:NN	139
\_str_format_parse_auxii:nN	
\_str_format_parse_auxiii:nN	E
..... 32, 40, 49, 50, 52	\else:
\_str_format_parse_auxiv:nwN	14, 21
..... 32, 57, 58, 60, 63	\exp_after:wN
\_str_format_parse_auxv:nN	21, 28, 196
\_str_format_parse_auxvi:nwN	\exp_args:Nc
..... 32, 69, 72, 75	185, 265, 320
\_str_format_parse_auxvii:nN	\exp_args:Nf
..... 32, 70, 76, 78	185, 187, 211, 217, 265, 272, 320
\_str_format_parse_end:nwn	\exp_args:No
\_str_format_put:fw	139
..... 16, 350, 354, 355, 358, 361	\exp_last_unbraced:Nf
\_str_format_put:nw	..... 34, 145, 222, 277, 342, 374, 396
16, 17, 156, 160, 161, 168, 170, 171,	\exp_not:N
230, 231, 233, 238, 240, 242, 286,	39, 40, 42, 54, 57, 58
287, 289, 294, 296, 298, 300, 301, 351	\exp_stop_f:
\_str_format_put:ow	13, 196
16, 237, 293	\ExplFileDate
\_str_format_seq:ff	5
195, 201	\ExplFileDescription
\_str_format_seq:nn	5
201, 201, 207	\ExplFileName
\_str_format_seq_end:w	5
205, 215, 215	\ExplFileVersion
\_str_format_seq_loop:nnNn	5
..... 203, 208, 208, 211	
\_str_format_tl:NNNnnNn	F
..... 145, 150, 150, 212	\fi:
\_str_format_tl_s:NNnnNNn	14, 29
180, 183, 183	\fp_format:nn
\_str_range_unsafe:nnn	3, 271, 271
170, 171	\fp_to_decimal:n
\_str_to_other:n	375, 409
35, 187	\fp_to_scientific:n
	405, 408
C	
\c_catcode_other_space_tl	G
43, 55, 237, 293	\group_begin:
\c_fifteen	328
354	\group_end:
\c_nine	332
13	
\c_one	I
113	\if_charcode:w
\c_sixteen	26
352	\if_int_compare:w
114, 124	13
	\int_abs:n
	267
	\int_case:nnF
	335, 367, 389
	\int_compare:nNnTF
	..... 232, 352, 404, 407
	\int_eval:n
	217, 350
	\int_format:nn
	3, 216, 216
	\int_max:nn
	95, 102, 112, 122, 134, 398, 399
	\int_to_binary:n
	252
	\int_to_hexadecimal:n
	254
	\int_to_octal:n
	253

<b>P</b>			
\prg_do_nothing:	411	\tl_format:Nn	3, 139, 139, 140
\prg_new_conditional:Npnn	11, 18	\tl_format:nn	139, 139, 141
\prg_replicate:nn	94, 101, 110, 120, 133, 337, 354, 369, 385	\tl_if_empty:nF	86, 243
\prg_return_false:	14, 21	\tl_if_empty:nTF	169, 299
\prg_return_true:	14, 27	\tl_to_lowercase:n	330
\ProvidesExplPackage	4	\tl_to_str:f	10, 143, 193, 220, 275
<b>R</b>		\tl_to_str:n	10
\RequirePackage	6	\token_if_eq_charcode:NNF	163, 172, 174
<b>S</b>		\token_if_eq_charcode:NNTF	68, 152, 159, 229, 285
\s__fp	283, 315, 333, 343, 348, 365, 376, 387, 397	\token_if_eq_meaning:NNTF	80, 288
\s__stop	16, 35, 80, 84, 181, 261, 263, 314, 318, 362	<b>U</b>	
\seq_format:cn	191	\use:fnf	8, 108
\seq_format:Nn	3, 191, 191, 200	\use:n	250, 251, 259
\str_case:nnF	235, 248, 291, 302	\use:nf	8, 92, 131, 212, 337, 369, 383
\str_range:nnn	359	\use:nn	8
<b>T</b>		\use:nnn	9
\tl_format:cn	139	\use_i:nn	196
		\use_i:nnnn	361
		\use_ii:nnn	215
		\use_none:n	210, 355, 362