

## 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 . . . . .	4
4.3	Alignment . . . . .	5
4.4	Formatting token lists . . . . .	7
4.5	Formatting sequences . . . . .	9
4.6	Formatting integers . . . . .	10
4.7	Formatting floating points . . . . .	11
4.8	Messages . . . . .	15
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 not implemented yet.

---

\*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 <@@=str>
3 <*package>
4 \ProvidesExplPackage
5   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}

```

```

6 \RequirePackage{l3str}
7 </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 16 \cs_new:Npn \__str_format_put:nw #1 #2 \s__stop { #2 \s__stop #1 }
\__str_format_put:fw 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)

```

## 4.2 Parsing a format specification

```

\__str_format_parse:n The goal is to parse
\__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
          18 \cs_new:Npn \__str_format_parse:n #1
          19 {
          20   \exp_last_unbraced:Nf \__str_format_parse_auxi:NN
          21   \__str_to_other:n {#1} \s__stop \s__stop {#1}
          22 }
          23 \cs_new:Npx \__str_format_parse_auxi:NN #1#2
          24 {
          25   \exp_not:N \__str_if_contains_char:nNTF { < > = ^ } #2
          26   { \exp_not:N \__str_format_parse_auxiii:nN { #1 #2 } }
          27   {
          28     \exp_not:N \__str_format_parse_auxii:nN
          29     { \c_catcode_other_space_tl } #1 #2
          30   }
          31 }
          32 \cs_new:Npn \__str_format_parse_auxii:nN #1#2
          33 {

```

```

34     \_str_if_contains_char:nNTF { < > = ^ } #2
35     { \_str_format_parse_auxiii:nN { #1 #2 } }
36     { \_str_format_parse_auxiii:nN { #1 ? } #2 }
37 }
38 \cs_new:Npx \_str_format_parse_auxiii:nN #1#2
39 {
40     \exp_not:N \_str_if_contains_char:nNTF
41     { + - \c_catcode_other_space_tl }
42     #2
43     { \exp_not:N \_str_format_parse_auxiv:nwN { #1 #2 } ; }
44     { \exp_not:N \_str_format_parse_auxiv:nwN { #1 ? } ; #2 }
45 }
46 \cs_new:Npn \_str_format_parse_auxiv:nwN #1#2; #3
47 {
48     \_str_format_if_digit:NTF #3
49     { \_str_format_parse_auxiv:nwN {#1} #2 #3 ; }
50     { \_str_format_parse_auxv:nN { #1 {#2} } #3 }
51 }
52 \cs_new:Npn \_str_format_parse_auxv:nN #1#2
53 {
54     \token_if_eq_charcode:NNTF . #2
55     { \_str_format_parse_auxvi:nwN {#1} 0 ; }
56     { \_str_format_parse_auxvii:nN { #1 { } } #2 }
57 }
58 \cs_new:Npn \_str_format_parse_auxvi:nwN #1#2; #3
59 {
60     \_str_format_if_digit:NTF #3
61     { \_str_format_parse_auxvi:nwN {#1} #2 #3 ; }
62     { \_str_format_parse_auxvii:nN { #1 {#2} } #3 }
63 }
64 \cs_new:Npn \_str_format_parse_auxvii:nN #1#2
65 {
66     \token_if_eq_meaning:NNTF \s__stop #2
67     { \_str_format_parse_end:nwn { #1 ? } #2 }
68     { \_str_format_parse_end:nwn { #1 #2 } }
69 }
70 \cs_new:Npn \_str_format_parse_end:nwn #1 #2 \s__stop \s__stop #3
71 {
72     \tl_if_empty:nF {#2}
73     { \_msg_kernel_expandable_error:nnn { str } { invalid-format } {#3} }
74     #1
75 }

```

(End definition for `\_str_format_parse:n` This function is documented on page ??.)

### 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 ^ 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.

```

76 \cs_new:cpn { __str_format_align_<:nnnN } #1#2#3#4
77 {
78   \use:nf { #2 #1 }
79   {
80     \prg_replicate:nn
81       { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
82       {#4}
83   }
84 }

```

(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.

```

85 \cs_new:cpn { __str_format_align_>:nnnN } #1#2#3#4
86 {
87   \prg_replicate:nn
88     { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
89     {#4}
90   #2 #1
91 }

```

(End definition for `\__str_format_align_>:nnnN`)

`\__str_format_align_^:nnnN`

`\__str_format_align_^:nnnN { $\langle body \rangle$ } { $\langle sign \rangle$ } { $\langle width \rangle$ }  $\langle fill \rangle$`

Centering “ $\langle sign \rangle$   $\langle body \rangle$ ” 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.

```

92 \cs_new:cpn { __str_format_align_^:nnnN } #1#2#3#4
93 {
94   \use:fnf
95   {
96     \prg_replicate:nn
97       {
98         \int_max:nn \c_zero
99         { #3 - \__str_count_unsafe:n { #2 #1 } - \c_one }
100       / \c_two
101     }
102     {#4}
103   }
104   { #2 #1 }
105   {
106     \prg_replicate:nn

```

```

107     {
108         \int_max:nn \c_zero
109         { #3 - \__str_count_unsafe:n { #2 #1 } }
110         / \c_two
111     }
112     {#4}
113 }
114 }

```

`\__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.

```

115 \cs_new:cpn { __str_format_align=:nnnN } #1#2#3#4
116 {
117     \use:nf {#2}
118     {
119         \prg_replicate:nn
120         { \int_max:nn { #3 - \__str_count_unsafe:n { #2 #1 } } { 0 } }
121         {#4}
122     }
123     #1
124 }

```

(End definition for `\__str_format_align=:nnnN`)

## 4.4 Formatting token lists

`\tl_format:Nn` Call `\__str_format_tl:NNNnnNn` to read the parsed <format specification>. Then convert the result to a string.

```

\tl_format:cn
\tl_format:nn
125 \cs_new_nopar:Npn \tl_format:Nn { \exp_args:No \tl_format:nn }
126 \cs_generate_variant:Nn \tl_format:Nn { c }
127 \cs_new:Npn \tl_format:nn #1#2
128 {
129     \tl_to_str:f
130     {
131         \exp_last_unbraced:Nf \__str_format_tl:NNNnnNn
132         { \__str_format_parse:n {#2} }
133         {#1}
134     }
135 }

```

(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, check the <precision>: if it is empty, we will eventually use the whole string, otherwise we

will only use a substring, starting at the index 1, and ending at #5. There is a need to use the “unsafe” function, as otherwise leading spaces would get stripped by f-expansion. Finally, check that the  $\langle style \rangle$  is ? or s.

```

136 \cs_new:Npn \__str_format_tl:NNNnnNn #1#2#3#4#5#6
137 {
138   \token_if_eq_charcode:NNTF #2 =
139   {
140     \__msg_kernel_expandable_error:nnnn
141     { str } { invalid-align-format } {#2} {tl}
142     \__str_format_put:nw { #1 < }
143   }
144   {
145     \token_if_eq_charcode:NNTF #2 ?
146     { \__str_format_put:nw { #1 < } }
147     { \__str_format_put:nw { #1 #2 } }
148   }
149   \token_if_eq_charcode:NNTF #3 ?
150   {
151     \__msg_kernel_expandable_error:nnnn
152     { str } { invalid-sign-format } {#3} {tl}
153   }
154   \__str_format_put:nw { {#4} }
155   \tl_if_empty:nTF {#5}
156   { \__str_format_put:nw { \use:n { } } }
157   { \__str_format_put:nw { \__str_substr_unsafe:nnn { {1} {#5} } } }
158   \token_if_eq_charcode:NNTF #6 s
159   {
160     \token_if_eq_charcode:NNTF #6 ?
161     {
162       \__msg_kernel_expandable_error:nnnn
163       { str } { invalid-style-format } {#6} {tl}
164     }
165   }
166   \__str_format_tl_s:NNnnNNn
167   \s__stop
168 }

```

(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  $\langle function \rangle$  and  $\langle arguments \rangle$  are built in such a way that f-expanding  $\langle function \rangle$  { $\langle other string \rangle$ }  $\langle arguments \rangle$  yields the piece of the  $\langle other string \rangle$  that we want to output. The  $\langle other string \rangle$  is built from the  $\langle token list \rangle$  by f-expanding \\_\_str\_to\_other:n.

```

169 \cs_new:Npn \__str_format_tl_s:NNnnNNn #1#2#3#4#5#6#7
170 {
171   \exp_args:Nc \exp_args:Nf
172   { \__str_format_align_#6:nnnN }
173   { \exp_args:Nf #2 { \__str_to_other:n {#7} } #3 }
174   { }

```



```

175         {#4} #5
176     }
(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.

```

177 \cs_new:Npn \seq_format:Nn #1#2
178 {
179     \tl_to_str:f
180     { \_str_format_seq:of {#1} { \_str_format_parse:n {#2} } }
181 }
182 \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 specification*. Set up the loop.

```

183 \cs_new:Npn \_str_format_seq:nn #1#2
184 {
185     \_str_format_seq_loop:nnNn { } {#2}
186     #1
187     { ? \_str_format_seq_end:w } { }
188 }
189 \cs_generate_variant:Nn \_str_format_seq:nn { of }
(End definition for \_str_format_seq:nn and \_str_format_seq:of)

```

```

\_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.

```

190 \cs_new:Npn \_str_format_seq_loop:nnNn #1#2#3#4
191 {
192     \use_none:n #3
193     \exp_args:Nf \_str_format_seq_loop:nnNn
194     { \use:nf {#1} { \_str_format_tl:NNnnNNn #2 {#4} } }
195     {#2}
196 }
(End definition for \_str_format_seq_loop:nnNn)

```

**\\_str\_format\_seq\_end:w** Pick the right piece in the loop above.

```

197 \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`.

```
198 \cs_new:Npn \int_format:nn #1
199   { \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

```
200 \cs_new:Npn \__str_format_int:nn #1#2
201   {
202     \tl_to_str:f
203     {
204       \exp_last_unbraced:Nf \__str_format_int:NNNnnNn
205       { \__str_format_parse:n {#2} }
206       {#1}
207     }
208   }
(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 *precision* was absent. Finally, dispatch depending on the *style*.

```
209 \cs_new:Npn \__str_format_int:NNNnnNn #1#2#3#4#5#6#7
210   {
211     \token_if_eq_charcode:NNTF #2 ?
212     { \__str_format_put:nw { #1 > } }
213     { \__str_format_put:nw { #1 #2 } }
214     \int_compare:nNnTF {#7} < \c_zero
215     { \__str_format_put:nw { - } }
216     {
217       \str_case:nnn {#3}
218       {
219         { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
220         { + } { \__str_format_put:nw { + } }
221       }
222       { \__str_format_put:nw { { } } }
223     }
224     \__str_format_put:nw { {#4} }
225     \tl_if_empty:nF {#5}
226     {
227       \__msg_kernel_expandable_error:nnnn
228       { str } { invalid-precision-format } {#5} {int}
229     }
```

```

230 \str_case:nnn {#6}
231 {
232   { ? } { \__str_format_int:NwnnNNn \use:n }
233   { d } { \__str_format_int:NwnnNNn \use:n }
234   { b } { \__str_format_int:NwnnNNn \int_to_binary:n }
235   { o } { \__str_format_int:NwnnNNn \int_to_octal:n }
236   { X } { \__str_format_int:NwnnNNn \int_to_hexadecimal:n }
237 }
238 {
239   \__msg_kernel_expandable_error:nnnn
240   { str } { invalid-style-format } {#6} { int }
241   \__str_format_int:NwnnNNn \use:n
242 }
243 \s__stop {#7}
244 }

```

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

```

\__str_format_int:NwnnNNn \__str_format_int:NwnnNNn <function> \s__stop {<width>} {<sign>} <fill>
<alignment> {<integer>}

```

Use the `format_align` function corresponding to the *<alignment>*, with the following arguments:

- the string formed by combining the sign *#4* with the result of converting the absolute value of the *<integer>* *#7* according to the conversion function *#1*;
- the *<width>*;
- the *<fill>* character.

```

245 \cs_new:Npn \__str_format_int:NwnnNNn #1#2 \s__stop #3#4#5#6#7
246 {
247   \exp_args:Nc \exp_args:Nf
248   { \__str_format_align_#6:nnnN }
249   { #1 { \int_abs:n {#7} } }
250   {#4}
251   {#3} #5
252 }

```

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

## 4.7 Formatting floating points

**\fp\_format:nn** Evaluate the first argument to an internal floating point number, and feed it to `\__str_format_fp:nn`.

```

253 \cs_new:Npn \fp_format:nn #1
254 { \exp_args:Nf \__str_format_fp:nn { \__fp_parse:n {#1} } }

```

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

`\_str_format_fp:nn` Parse the  $\langle format\ specification \rangle$  and feed it to `\_str_format_fp:NNNnnNn`. Then convert the result to a string

```

255 \cs_new:Npn \_str_format_fp:nn #1#2
256 {
257   \tl_to_str:f
258   {
259     \exp_last_unbraced:Nf \_str_format_fp:NNNnnNw
260     { \_str_format_parse:n {#2} }
261     #1
262   }
263 }

```

(End definition for `\_str_format_fp:nn`)

`\_str_format_fp:NNNnnNw` `\_str_format_fp:NNNnnNw`  $\langle fill \rangle$   $\langle alignment \rangle$   $\langle format\ sign \rangle$   $\{ \langle width \rangle \}$   $\{ \langle precision \rangle \}$   $\langle style \rangle$  `\s_fp` `\_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 `\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 `\s__stop`. Afterwards, check the  $\langle precision \rangle$ : if it was not given, replace it by 6 (default precision). Finally, dispatch depending on the  $\langle style \rangle$ .

```

264 \cs_new:Npn \_str_format_fp:NNNnnNw
265   #1#2#3#4#5#6 \s_fp \_fp_chk:w #7 #8
266 {
267   \token_if_eq_charcode:NNTF #2 ?
268   { \_str_format_put:nw { #1 > } }
269   { \_str_format_put:nw { #1 #2 } }
270   \token_if_eq_meaning:NNTF 2 #8
271   { \_str_format_put:nw { - } }
272   {
273     \str_case:nnn {#3}
274     {
275       { ~ } { \_str_format_put:ow { \c_catcode_other_space_tl } }
276       { + } { \_str_format_put:nw { + } }
277     }
278     { \_str_format_put:nw { { } } }
279   }
280   \_str_format_put:nw { {#4} }
281   \tl_if_empty:nTF {#5}
282   { \_str_format_put:nw { { 6} } }
283   { \_str_format_put:nw { {#5} } }
284   \str_case:nnn {#6}
285   {
286     { e } { \_str_format_fp:wnnnNNw \_str_format_fp_e:wn }
287     { f } { \_str_format_fp:wnnnNNw \_str_format_fp_f:wn }
288     { g } { \_str_format_fp:wnnnNNw \_str_format_fp_g:wn }
289     { ? } { \_str_format_fp:wnnnNNw \_str_format_fp_g:wn }

```

```

290     }
291     {
292       \__msg_kernel_expandable_error:nnnn
293       { str } { invalid-style-format } {#6} { fp }
294       \__str_format_fp:wnnnNNw \__str_format_fp_g:wn
295     }
296     \s__stop
297     \s__fp \__fp_chk:w #7 #8
298   }

```

(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> ;

```

```

299 \cs_new:Npn \__str_format_fp:wnnnNNw
300   #1 \s__stop #2 #3 #4 #5#6 #7 ;
301   {
302     \exp_args:Nc \exp_args:Nf
303     { \__str_format_align_#6:nnnN }
304     { #1 #7 ; {#2} }
305     {#4}
306     {#3} #5
307   }

```

(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).

```

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

```

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

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

```

310 \group_begin:
311 \char_set_catcode_other:N E
312 \tl_to_lowercase:n
313   {
314     \group_end:
315     \cs_new:Npn \__str_format_fp_e:wn \s__fp \__fp_chk:w #1#2#3 ; #4
316     {
317       \int_case:nnn {#1}
318       {
319         {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } e 0 }
320         {2} { inf }
321         {3} { nan }
322       }
323       {
324         \exp_last_unbraced:Nf \__str_format_fp_e_aux:wn

```

```

325         \__str_format_fp_round:wn \s__fp \__fp_chk:w #1#2#3 ; { #4 + 1 }
326         {#4}
327     }
328 }
329 \cs_new:Npn \__str_format_fp_e_aux:wn
330   \s__fp \__fp_chk:w #1#2 #3 #4#5#6#7 ; #8
331   {
332     \__str_format_put:fw { \int_eval:n { #3 - 1 } }
333     \__str_format_put:nw { e }
334     \int_compare:nNnTF {#8} > \c_sixteen
335     {
336       \__str_format_put:fw { \prg_replicate:nn { #8 - \c_fifteen } {0} }
337       \__str_format_put:fw { \use_none:n #4#5#6#7 }
338     }
339     {
340       \__str_format_put:fw
341       { \str_substr:nnn { #4#5#6#7 0 } { 2 } { #8 + 1 } }
342     }
343     \__str_format_put:fw { \use_i:nnnn #4 . }
344     \use_none:n \s__stop
345   }
346 }

```

(End definition for \\_\_str\_format\_fp\_e:wn This function is documented on page 3.)

\\_\_str\_format\_fp\_f:wn  
 \\_\_str\_format\_fp\_f\_aux:wwwn

With the **f** type, first filter out special cases. In the normal case, round to #4 (absolute) decimal places.

```

347 \cs_new:Npn \__str_format_fp_f:wn \s__fp \__fp_chk:w #1#2#3 ; #4
348   {
349     \int_case:nnn {#1}
350     {
351       {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } }
352       {2} { inf }
353       {3} { nan }
354     }
355     {
356       \exp_last_unbraced:Nf \__str_format_fp_f_aux:wwwn
357       \fp_to_decimal:n
358       { abs ( round ( \s__fp \__fp_chk:w #1#2#3 ; , #4 ) ) }
359       . . ;
360       {#4}
361     }
362   }
363 \cs_new:Npn \__str_format_fp_f_aux:wwwn #1 . #2 . #3 ; #4
364   {
365     \use:nf
366     { #1 . #2 }
367     { \prg_replicate:nn { #4 - \__str_count_unsafe:n {#2} } {0} }
368   }

```

(End definition for \\_\_str\_format\_fp\_f:wn This function is documented on page 3.)

```

    \_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.

```

369 \cs_new:Npn \_str_format_fp_g:wn \s__fp \_fp_chk:w #1#2 ; #3
370 {
371   \int_case:nnn {#1}
372   {
373     {0} { 0 }
374     {2} { inf }
375     {3} { nan }
376   }
377   {
378     \exp_last_unbraced:Nf \_str_format_fp_g_aux:wn
379     \_str_format_fp_round:wn \s__fp \_fp_chk:w #1#2 ;
380     { \int_max:nn {1} {#3} }
381     { \int_max:nn {1} {#3} }
382   }
383 }
384 \cs_new:Npn \_str_format_fp_g_aux:wn #1; #2
385 {
386   \int_compare:nNnTF { \_fp_exponent:w #1; } < { -3 }
387   { \fp_to_scientific:n }
388   {
389     \int_compare:nNnTF { \_fp_exponent:w #1; } > {#2}
390     { \fp_to_scientific:n }
391     { \fp_to_decimal:n }
392   }
393   { \_fp_abs_o:w #1; \prg_do_nothing: }
394 }

```

(End definition for `\_str_format_fp_g:wn` This function is documented on page 3.)

## 4.8 Messages

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

```

395 \_msg_kernel_new:nnn { str } { invalid-format }
396 { Invalid~format~'#1'. }
397 \_msg_kernel_new:nnn { str } { invalid-align-format }
398 { Invalid~alignment~'#1'~for~type~'#2'. }
399 \_msg_kernel_new:nnn { str } { invalid-sign-format }
400 { Invalid~sign~'#1'~for~type~'#2'. }
401 \_msg_kernel_new:nnn { str } { invalid-precision-format }
402 { Invalid~precision~'#1'~for~type~'#2'. }
403 \_msg_kernel_new:nnn { str } { invalid-style-format }
404 { 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.

405 `\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	
<code>\_fp_abs_o:w</code> . . . . .	393
<code>\_fp_chk:w</code> . . . . .	265, 297, 315, 325, 330, 347, 358, 369, 379
<code>\_fp_exponent:w</code> . . . . .	309, 386, 389
<code>\_fp_parse:n</code> . . . . .	254, 309
<code>\_msg_kernel_expandable_error:nnn</code> .	73
<code>\_msg_kernel_expandable_error:nnnn</code> . . . . .	140, 151, 162, 227, 239, 292
<code>\_msg_kernel_new:nnn</code> . . . . .	395, 397, 399, 401, 403
<code>\_str_count_unsafe:n</code> . . . . .	81, 88, 99, 109, 120, 367
<code>\_str_format_align&lt;:nnnN</code> . . . . .	76
<code>\_str_format_align&gt;:nnnN</code> . . . . .	85
<code>\_str_format_align^:nnnN</code> . . . . .	92
<code>\_str_format_fp:NNnnNw</code> . . . . .	259, 264, 264
<code>\_str_format_fp:nn</code> . . . . .	254, 255, 255
<code>\_str_format_fp:wnnnNw</code> . . . . .	286, 287, 288, 289, 294, 299, 299
<code>\_str_format_fp_e:wn</code> . . . . .	286, 310, 315
<code>\_str_format_fp_e_aux:wn</code> . . . . .	310, 324, 329
<code>\_str_format_fp_f:wn</code> . . . . .	287, 347, 347
<code>\_str_format_fp_f_aux:wwn</code> . . . . .	347, 356, 363
<code>\_str_format_fp_g:wn</code> . . . . .	288, 289, 294, 369, 369
<code>\_str_format_fp_g_aux:wn</code> . . . . .	369, 378, 384
<code>\_str_format_fp_round:wn</code> . . . . .	308, 308, 325, 379
<code>\_str_format_if_digit:N</code> . . . . .	11
<code>\_str_format_if_digit:NTF</code> . . . . .	11, 48, 60
<code>\_str_format_int:NNnnNn</code> . . . . .	204, 209, 209
<code>\_str_format_int:NwnnNn</code> . . . . .	232, 233, 234, 235, 236, 241, 245, 245
<code>\_str_format_int:nn</code> . . . . .	199, 200, 200
<code>\_str_format_parse:n</code> . . . . .	18, 18, 132, 180, 205, 260
<code>\_str_format_parse_auxi:NN</code> . . . . .	18, 20, 23
<code>\_str_format_parse_auxii:nN</code> . . . . .	18, 28, 32
<code>\_str_format_parse_auxiii:nN</code> . . . . .	18, 26, 35, 36, 38
<code>\_str_format_parse_auxiv:nwN</code> . . . . .	18, 43, 44, 46, 49
<code>\_str_format_parse_auxv:nN</code> . . . . .	18, 50, 52
<code>\_str_format_parse_auxvi:nwN</code> . . . . .	18, 55, 58, 61
<code>\_str_format_parse_auxvii:nN</code> . . . . .	18, 56, 62, 64
<code>\_str_format_parse_end:nwn</code> . . . . .	18, 67, 68, 70
<code>\_str_format_put:fw</code> . . . . .	16, 332, 336, 337, 340, 343
<code>\_str_format_put:nw</code> . . . . .	16, 16, 17, 142, 146, 147, 154, 156, 157, 212, 213, 215, 220, 222, 224, 268, 269, 271, 276, 278, 280, 282, 283, 333
<code>\_str_format_put:ow</code> . . . . .	16, 219, 275
<code>\_str_format_seq:nn</code> . . . . .	183, 183, 189
<code>\_str_format_seq:of</code> . . . . .	180, 183
<code>\_str_format_seq_end:w</code> . . . . .	187, 197, 197
<code>\_str_format_seq_loop:nnNn</code> . . . . .	185, 190, 190, 193
<code>\_str_format_tl:NNnnNn</code> . . . . .	131, 136, 136, 194
<code>\_str_format_tl_s:NNnnNn</code> . . . . .	166, 169, 169
<code>\_str_if_contains_char:nNTF</code> . . . . .	25, 34, 40
<code>\_str_substr_unsafe:nnn</code> . . . . .	157
<code>\_str_to_other:n</code> . . . . .	21, 173
C	
<code>\c_catcode_other_space_tl</code> . . . . .	29, 41, 219, 275



\c_fifteen	336	\int_to_binary:n	234
\c_nine	13	\int_to_hexadecimal:n	236
\c_one	99	\int_to_octal:n	235
\c_sixteen	334		
\c_two	100, 110	<b>P</b>	
\c_zero	98, 108, 214	\prg_do_nothing:	393
\char_set_catcode_other:N	311	\prg_new_conditional:Npnn	11
\cs_generate_variant:Nn	8, 9, 10, 17, 126, 182, 189	\prg_replicate:nn	80, 87, 96, 106, 119, 319, 336, 351, 367
\cs_new:cpn	76, 85, 92, 115	\prg_return_false:	14
\cs_new:Npn	16, 18, 32, 46, 52, 58, 64, 70, 127, 136, 169, 177, 183, 190, 197, 198, 200, 209, 245, 253, 255, 264, 299, 308, 315, 329, 347, 363, 369, 384	\prg_return_true:	14
\cs_new:Npx	23, 38	\ProvidesExplPackage	4
\cs_new_nopar:Npn	125		
		<b>R</b>	
<b>E</b>		\RequirePackage	6
\else:	14		
\exp_args:Nc	171, 247, 302	<b>S</b>	
\exp_args:Nf	171, 173, 193, 199, 247, 254, 302	\s_fp	265, 297, 315, 325, 330, 347, 358, 369, 379
\exp_args:No	125	\s_stop	16, 21, 66, 70, 167, 243, 245, 296, 300, 344
\exp_last_unbraced:Nf	20, 131, 204, 259, 324, 356, 378	\seq_format:cn	177
\exp_not:N	25, 26, 28, 40, 43, 44	\seq_format:Nn	3, 177, 177, 182
\exp_stop_f:	13	\str_case:nnn	217, 230, 273, 284
\ExplFileDate	5	\str_substr:nnn	341
\ExplFileDescription	5		
\ExplFileName	5	<b>T</b>	
\ExplFileVersion	5	\tl_format:cn	125
		\tl_format:Nn	3, 125, 125, 126
<b>F</b>		\tl_format:nn	125, 125, 127
\fi:	14	\tl_if_empty:nF	72, 225
\fp_format:nn	3, 253, 253	\tl_if_empty:nTF	155, 281
\fp_to_decimal:n	357, 391	\tl_to_lowercase:n	312
\fp_to_scientific:n	387, 390	\tl_to_str:f	10, 129, 179, 202, 257
		\tl_to_str:n	10
<b>G</b>		\token_if_eq_charcode:NnF	149, 158, 160
\group_begin:	310	\token_if_eq_charcode:NNTF	54, 138, 145, 211, 267
\group_end:	314	\token_if_eq_meaning:NNTF	66, 270
<b>I</b>		<b>U</b>	
\if_int_compare:w	13	\use:fnf	8, 94
\int_abs:n	249	\use:n	156, 232, 233, 241
\int_case:nnn	317, 349, 371	\use:nf	8, 78, 117, 194, 319, 351, 365
\int_compare:nNnTF	214, 334, 386, 389	\use:nn	8
\int_eval:n	199, 332	\use:nnn	9
\int_format:nn	3, 198, 198	\use_i:nnnn	343
\int_max:nn	81, 88, 98, 108, 120, 380, 381	\use_ii:nnn	197
		\use_none:n	192, 337, 344