

Contents

1	Format specifications	2
2	Formatting various data-types	3
3	Possibilities, and things to do	3
4	l3str-format implementation	3
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
	Index	16

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

The L^AT_EX3 Project[†]

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.

[†]E-mail: latex-team@latex-project.org

2 Formatting various data-types

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

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:nN. This function is documented on page ??.)

```

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`. 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 \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 {<body>} {<sign>} {<width>} {<fill>}
    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 {<body>} {<sign>} {<width>} {<fill>}
    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:fnt
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} {tl}
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} {tl}
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     { \__str_format_seq:of {#1} { \__str_format_parse:n {#2} } }
195 }
196 \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:of specification>. Set up the loop.

```

197 \cs_new:Npn \__str_format_seq:nn #1#2
198 {
199     \__str_format_seq_loop:nnNn { } {#2}
200     #1
201     { ? \__str_format_seq_end:w } { }
202 }
203 \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.

```

204 \cs_new:Npn \__str_format_seq_loop:nnNn #1#2#3#4
205 {
206   \use_none:n #3
207   \exp_args:Nf \__str_format_seq_loop:nnNn
208   { \use:nf {#1} { \__str_format_tl:NNNnnNn #2 {#4} } }
209   {#2}
210 }

```

(End definition for `__str_format_seq_loop:nnNn`.)

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

```

211 \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`.

```

212 \cs_new:Npn \int_format:nn #1
213 { \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

```

214 \cs_new:Npn \__str_format_int:nn #1#2
215 {
216   \tl_to_str:f
217   {
218     \exp_last_unbraced:Nf \__str_format_int:NNNnnNn
219     { \__str_format_parse:n {#2} }
220     {#1}
221   }
222 }

```

(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*.

```

223 \cs_new:Npn \__str_format_int:NNNnnNn #1#2#3#4#5#6#7

```

```

224 {
225   \token_if_eq_charcode:NNTF #2 ?
226   { \__str_format_put:nw { #1 > } }
227   { \__str_format_put:nw { #1 #2 } }
228   \int_compare:nNnTF {#7} < \c_zero
229   { \__str_format_put:nw { - } }
230   {
231     \str_case:nnn {#3}
232     {
233       { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
234       { + } { \__str_format_put:nw { + } }
235     }
236     { \__str_format_put:nw { { } } }
237   }
238   \__str_format_put:nw { {#4} }
239   \tl_if_empty:nF {#5}
240   {
241     \__msg_kernel_expandable_error:nnnn
242     { str } { invalid-precision-format } {#5} {int}
243   }
244   \str_case:nnn {#6}
245   {
246     { ? } { \__str_format_int:NwnnNNn \use:n }
247     { d } { \__str_format_int:NwnnNNn \use:n }
248     { b } { \__str_format_int:NwnnNNn \int_to_binary:n }
249     { o } { \__str_format_int:NwnnNNn \int_to_octal:n }
250     { X } { \__str_format_int:NwnnNNn \int_to_hexadecimal:n }
251   }
252   {
253     \__msg_kernel_expandable_error:nnnn
254     { str } { invalid-style-format } {#6} { int }
255     \__str_format_int:NwnnNNn \use:n
256   }
257   \s__stop {#7}
258 }

```

(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.

```

259 \cs_new:Npn \__str_format_int:NwnnNNn #1#2 \s__stop #3#4#5#6#7

```

```

260 {
261     \exp_args:Nc \exp_args:Nf
262     { __str_format_align_#6:nnnN }
263     { #1 { \int_abs:n {#7} } }
264     {#4}
265     {#3} #5
266 }

```

(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`.

```

267 \cs_new:Npn \fp_format:nn #1
268 { \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 *format specification* and feed it to `__str_format_fp:NNNnnNw`. Then convert the result to a string

```

269 \cs_new:Npn \__str_format_fp:nn #1#2
270 {
271     \tl_to_str:f
272     {
273         \exp_last_unbraced:Nf \__str_format_fp:NNNnnNw
274         { \__str_format_parse:n {#2} }
275         #1
276     }
277 }

```

(End definition for `__str_format_fp:nn`.)

`__str_format_fp:NNNnnNw` `__str_format_fp:NNNnnNw` *fill* *alignment* *format sign* {*width*} {*precision*}
style `\s__fp` `__fp_chk:w` *fp type* *fp sign* *fp body* ;

First set the default alignment ? to >. Place the modified information after a trailing `\s__stop` for later retrieval. Then check the *format sign* and the *fp sign*: if the floating point is negative, always put -. Otherwise (including `nan`), 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 the *precision*: if it was not given, replace it by 6 (default precision). Finally, dispatch depending on the *style*.

```

278 \cs_new:Npn \__str_format_fp:NNNnnNw
279     #1#2#3#4#5#6 \s__fp \__fp_chk:w #7 #8
280 {
281     \token_if_eq_charcode:NNTF #2 ?
282     { \__str_format_put:nw { #1 > } }
283     { \__str_format_put:nw { #1 #2 } }
284     \token_if_eq_meaning:NNTF 2 #8
285     { \__str_format_put:nw { - } }

```

```

286     {
287         \str_case:nnn {#3}
288         {
289             { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
290             { + } { \__str_format_put:nw { + } }
291         }
292         { \__str_format_put:nw { { } } }
293     }
294     \__str_format_put:nw { {#4} }
295     \tl_if_empty:nTF {#5}
296     { \__str_format_put:nw { { 6} } }
297     { \__str_format_put:nw { {#5} } }
298     \str_case:nnn {#6}
299     {
300         { e } { \__str_format_fp:wnnnNNw \__str_format_fp_e:wn }
301         { f } { \__str_format_fp:wnnnNNw \__str_format_fp_f:wn }
302         { g } { \__str_format_fp:wnnnNNw \__str_format_fp_g:wn }
303         { ? } { \__str_format_fp:wnnnNNw \__str_format_fp_g:wn }
304     }
305     {
306         \__msg_kernel_expandable_error:nnnn
307         { str } { invalid-style-format } {#6} { fp }
308         \__str_format_fp:wnnnNNw \__str_format_fp_g:wn
309     }
310     \s__stop
311     \s__fp \__fp_chk:w #7 #8
312 }

```

(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>} ;

```

```

313 \cs_new:Npn \__str_format_fp:wnnnNNw
314     #1 \s__stop #2 #3 #4 #5#6 #7 ;
315 {
316     \exp_args:Nc \exp_args:Nf
317     { \__str_format_align_#6:nnnN }
318     { #1 #7 ; {#2} }
319     {#4}
320     {#3} #5
321 }

```

(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).

```

322 \cs_new:Npn \__str_format_fp_round:wn #1 ; #2
323 { \__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).

```

324 \group_begin:
325 \char_set_catcode_other:N E
326 \tl_to_lowercase:n
327 {
328   \group_end:
329   \cs_new:Npn \__str_format_fp_e:wn \s__fp \__fp_chk:w #1#2#3 ; #4
330   {
331     \int_case:nnn {#1}
332     {
333       {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } e 0 }
334       {2} { inf }
335       {3} { nan }
336     }
337     {
338       \exp_last_unbraced:Nf \__str_format_fp_e_aux:wn
339       \__str_format_fp_round:wn \s__fp \__fp_chk:w #1#2#3 ; { #4 + 1 }
340       {#4}
341     }
342   }
343   \cs_new:Npn \__str_format_fp_e_aux:wn
344   \s__fp \__fp_chk:w #1#2 #3 #4#5#6#7 ; #8
345   {
346     \__str_format_put:fw { \int_eval:n { #3 - 1 } }
347     \__str_format_put:nw { e }
348     \int_compare:nNnTF {#8} > \c_sixteen
349     {
350       \__str_format_put:fw { \prg_replicate:nn { #8 - \c_fifteen } {0} }
351       \__str_format_put:fw { \use_none:n #4#5#6#7 }
352     }
353     {
354       \__str_format_put:fw
355       { \str_range:nnn { #4#5#6#7 0 } { 2 } { #8 + 1 } }
356     }
357     \__str_format_put:fw { \use_i:nnnn #4 . }
358     \use_none:n \s__stop
359   }
360 }

```

(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:www`

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

```

361 \cs_new:Npn \__str_format_fp_f:wn \s__fp \__fp_chk:w #1#2#3 ; #4
362 {
363   \int_case:nnn {#1}
364   {
365     {0} { \use:nf { 0 . } { \prg_replicate:nn {#4} { 0 } } }
366     {2} { inf }

```

```

367     {3} { nan }
368   }
369   {
370     \exp_last_unbraced:Nf \_str_format_fp_f_aux:wwwn
371     \fp_to_decimal:n
372     { abs ( round ( \s__fp \_fp_chk:w #1#2#3 ; , #4 ) ) }
373     . . ;
374     {#4}
375   }
376 }
377 \cs_new:Npn \_str_format_fp_f_aux:wwwn #1 . #2 . #3 ; #4
378 {
379   \use:nf
380   { #1 . #2 }
381   { \prg_replicate:nn { #4 - \_str_count_unsafe:n {#2} } {0} }
382 }

```

(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.

```

383 \cs_new:Npn \_str_format_fp_g:wn \s__fp \_fp_chk:w #1#2 ; #3
384 {
385   \int_case:nnn {#1}
386   {
387     {0} { 0 }
388     {2} { inf }
389     {3} { nan }
390   }
391   {
392     \exp_last_unbraced:Nf \_str_format_fp_g_aux:wn
393     \_str_format_fp_round:wn \s__fp \_fp_chk:w #1#2 ;
394     { \int_max:nn {1} {#3} }
395     { \int_max:nn {1} {#3} }
396   }
397 }
398 \cs_new:Npn \_str_format_fp_g_aux:wn #1; #2
399 {
400   \int_compare:nNnTF { \_fp_exponent:w #1; } < { -3 }
401   { \fp_to_scientific:n }
402   {
403     \int_compare:nNnTF { \_fp_exponent:w #1; } > {#2}
404     { \fp_to_scientific:n }
405     { \fp_to_decimal:n }
406   }
407   { \_fp_abs_o:w #1; \prg_do_nothing: }

```

408 }

(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.

```
409 \_msg_kernel_new:nnn { str } { invalid-format }
410 { Invalid-format~'#1'. }
411 \_msg_kernel_new:nnn { str } { invalid-align-format }
412 { Invalid-alignment~'#1'~for-type~'#2'. }
413 \_msg_kernel_new:nnn { str } { invalid-sign-format }
414 { Invalid-sign~'#1'~for-type~'#2'. }
415 \_msg_kernel_new:nnn { str } { invalid-precision-format }
416 { Invalid-precision~'#1'~for-type~'#2'. }
417 \_msg_kernel_new:nnn { str } { invalid-style-format }
418 { Invalid-style~'#1'~for-type~'#2'. }
```

4.9 Todos

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

419 `</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>	407
<code>_fp_chk:w</code>	279, 311, 329, 339, 344, 361, 372, 383, 393
<code>_fp_exponent:w</code>	323, 400, 403
<code>_fp_parse:n</code>	268, 323
<code>_msg_kernel_expandable_error:nnn</code> .	87
<code>_msg_kernel_expandable_error:nnnn</code>	154, 165, 176, 241, 253, 306
<code>_msg_kernel_new:nnn</code>	409, 411, 413, 415, 417
<code>_prg_break:</code>	21, 28
<code>_prg_break_point:</code>	22
<code>_str_count_unsafe:n</code>	95, 102, 113, 123, 134, 381
<code>_str_format_align<:nnnN</code>	90
<code>_str_format_align>:nnnN</code>	99
<code>_str_format_align~:nnnN</code>	106
<code>_str_format_fp:NNNnnNw</code>	273, 278, 278
<code>_str_format_fp:nn</code>	268, 269, 269
<code>_str_format_fp:wnnnNNw</code>	300, 301, 302, 303, 308, 313, 313
<code>_str_format_fp_e:wn</code>	300, 324, 329
<code>_str_format_fp_e_aux:wn</code>	324, 338, 343
<code>_str_format_fp_f:wn</code>	301, 361, 361
<code>_str_format_fp_f_aux:wwn</code>	361, 370, 377
<code>_str_format_fp_g:wn</code>	302, 303, 308, 383, 383
<code>_str_format_fp_g_aux:wn</code>	383, 392, 398
<code>_str_format_fp_round:wn</code>	322, 322, 339, 393
<code>_str_format_if_digit:N</code>	11
<code>_str_format_if_digit:NTF</code>	11, 62, 74
<code>_str_format_if_in:nN</code>	18

\prg_return_false:	14, 21	\tl_format:nn	139, 139, 141
\prg_return_true:	14, 27	\tl_if_empty:nF	86, 239
\ProvidesExplPackage	4	\tl_if_empty:nTF	169, 295
R		\tl_to_lowercase:n	326
\RequirePackage	6	\tl_to_str:f	10, 143, 193, 216, 271
S		\tl_to_str:n	10
\s_fp	279,	\token_if_eq_charcode:NNF .	163, 172, 174
	311, 329, 339, 344, 361, 372, 383, 393	\token_if_eq_charcode:NNTF	68, 152, 159, 225, 281
\s_stop	16,	\token_if_eq_meaning:NNTF	80, 284
	35, 80, 84, 181, 257, 259, 310, 314, 358	U	
\seq_format:cn	191	\use:fnf	8, 108
\seq_format:Nn	3, 191, 191, 196	\use:n	246, 247, 255
\str_case:nnn	231, 244, 287, 298	\use:nf	8, 92, 131, 208, 333, 365, 379
\str_range:nnn	355	\use:nn	8
T		\use:nnn	9
\tl_format:cn	139	\use_i:nnnn	357
\tl_format:Nn	3, 139, 139, 140	\use_ii:nnn	211
		\use_none:n	206, 351, 358