

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     {
195         \_str_format_seq:ff { \exp_after:wN \_str_format_seq_extract:w #1 }
196         { \_str_format_parse:n {#2} }
197     }
198 }
199 \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_extract:w Extract the contents of a sequence as items of the form _seq_item:n {<item>}.
 200 \cs_new:Npn _str_format_seq_extract:w \s__seq #1 \s_obj_end { \exp_stop_f: #1 }
 (End definition for _str_format_seq_extract:w.)

_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.

```

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 `<precision>` was absent. Finally, dispatch depending on the `<style>`.

```

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 <function> \s__stop {<width>} {<sign>} <fill>
<alignment> {<integer>}`

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 \__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`.

```

271 \cs_new:Npn \fp_format:nn #1
272 { \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: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 \__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 $\langle style \rangle$.

```

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 $\backslash_str_format_fp:NNNnnNw$.)

```

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

```

```

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 figures (one before the decimal separator, #4 after).

```

\_str_format_fp_e_aux:wn
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`. This function is documented on page 3.)

`__str_format_fp_f:wn`
`__str_format_fp_f_aux:wwn`

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

```

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_abs_o:w #1; \prg_do_nothing: }
412 }

```

(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.

```

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 ∞ . 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 | | __msg_kernel_expandable_error:nnnn |
|--|---------------|-------------------------------------|
| __fp_abs_o:w | 411 | 154, 165, 176, 245, 257, 310 |
| __fp_chk:w | 283, | __msg_kernel_new:nnn |
| 315, 333, 343, 348, 365, 376, 387, 397 | | 413, 415, 417, 419, 421 |
| __fp_exponent:w | 327, 404, 407 | __prg_break: |
| __fp_parse:n | 272, 327 | __prg_break_point: |
| __msg_kernel_expandable_error:nnn | 87 | __str_count_unsafe:n |

| | |
|---|---|
| 95, 102, 113, 123, 134, 385 | _str_format_seq_extract:w 195, 200, 200 |
| _str_format_align<:nnnN 90 | _str_format_seq_loop:nnNn 203, 208, 208, 211 |
| _str_format_align>:nnnN 99 | _str_format_tl:NNNnnNn 145, 150, 150, 212 |
| _str_format_align^:nnnN 106 | _str_format_tl_s:NNnnNNn 180, 183, 183 |
| _str_format_fp:NNNnnNw .. 277, 282, 282 | _str_range_unsafe:nnn 170, 171 |
| _str_format_fp:nn 272, 273, 273 | _str_to_other:n 35, 187 |
| _str_format_fp:wnnnNNw 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:www 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 326, 326, 343, 397 | |
| _str_format_if_digit:N 11 | |
| _str_format_if_digit:NTF ... 11, 62, 74 | |
| _str_format_if_in:nN 18 | |
| _str_format_if_in:nNTF .. 18, 39, 48, 54 | |
| _str_format_if_in_aux:NN 18, 20, 24, 30 | |
| _str_format_int:NNNnnNn . 222, 227, 227 | |
| _str_format_int:NwnnNNn 250, 251, 252, 253, 254, 259, 263, 263 | |
| _str_format_int:nn 217, 218, 218 | |
| _str_format_parse:n 32, 32, 146, 196, 223, 278 | |
| _str_format_parse_auxi:NN .. 32, 34, 37 | |
| _str_format_parse_auxii:nN . 32, 42, 46 | |
| _str_format_parse_auxiii:nN 32, 40, 49, 50, 52 | |
| _str_format_parse_auxiv:nwN 32, 57, 58, 60, 63 | |
| _str_format_parse_auxv:nN .. 32, 64, 66 | |
| _str_format_parse_auxvi:nwN 32, 69, 72, 75 | |
| _str_format_parse_auxvii:nN 32, 70, 76, 78 | |
| _str_format_parse_end:nwn 32, 81, 82, 84 | |
| _str_format_put:fw 16, 350, 354, 355, 358, 361 | |
| _str_format_put:nw 16, 16, 17, 156, 160, 161, 168, 170, 171, 230, 231, 233, 238, 240, 242, 286, 287, 289, 294, 296, 298, 300, 301, 351 | |
| _str_format_put:ow 16, 237, 293 | |
| _str_format_seq:ff 195, 201 | |
| _str_format_seq:nn 201, 201, 207 | |
| _str_format_seq_end:w .. 205, 215, 215 | |

C

| |
|---|
| \c_catcode_other_space_tl 43, 55, 237, 293 |
| \c_fifteen 354 |
| \c_nine 13 |
| \c_one 113 |
| \c_sixteen 352 |
| \c_two 114, 124 |
| \c_zero 112, 122, 232 |
| \char_set_catcode_other:N 329 |
| \cs_generate_variant:Nn 8, 9, 10, 17, 140, 199, 207 |
| \cs_new:cpn 90, 99, 106, 129 |
| \cs_new:Npn 16, 24, 32, 46, 60, 66, 72, 78, 84, 141, 150, 183, 191, 200, 201, 208, 215, 216, 218, 227, 263, 271, 273, 282, 317, 326, 333, 347, 365, 381, 387, 402 |
| \cs_new:Npx 37, 52 |
| \cs_new_nopar:Npn 139 |

E

| |
|--|
| \else: 14, 21 |
| \exp_after:wN 21, 28, 195 |
| \exp_args:Nc 185, 265, 320 |
| \exp_args:Nf 185, 187, 211, 217, 265, 272, 320 |
| \exp_args:No 139 |
| \exp_last_unbraced:Nf 34, 145, 222, 277, 342, 374, 396 |
| \exp_not:N 39, 40, 42, 54, 57, 58 |
| \exp_stop_f: 13, 200 |
| \ExplFileDate 5 |
| \ExplFileDescription 5 |
| \ExplFileName 5 |
| \ExplFileVersion 5 |

F

| |
|------------------------------------|
| \fi: 14, 29 |
| \fp_format:nn 3, 271, 271 |
| \fp_to_decimal:n 375, 409 |
| \fp_to_scientific:n 405, 408 |

| | | | |
|--|--------------------|--|--------------------------------|
| G | | \s__seq | 200 |
| \group_begin: | 328 | \s__stop | 16, |
| \group_end: | 332 | 35, 80, 84, 181, 261, 263, 314, 318, 362 | |
| I | | \s_obj_end | 200 |
| \if_charcode:w | 26 | \seq_format:cn | 191 |
| \if_int_compare:w | 13 | \seq_format:Nn | 3, 191, 191, 199 |
| \int_abs:n | 267 | \str_case:nnF | 235, 248, 291, 302 |
| \int_case:nnF | 335, 367, 389 | \str_range:nnn | 359 |
| \int_compare:nNnTF | 232, 352, 404, 407 | T | |
| \int_eval:n | 217, 350 | \tl_format:cn | 139 |
| \int_format:nn | 3, 216, 216 | \tl_format:Nn | 3, 139, 139, 140 |
| \int_max:nn 95, 102, 112, 122, 134, 398, 399 | | \tl_format:nn | 139, 139, 141 |
| \int_to_binary:n | 252 | \tl_if_empty:nF | 86, 243 |
| \int_to_hexadecimal:n | 254 | \tl_if_empty:nTF | 169, 299 |
| \int_to_octal:n | 253 | \tl_to_lowercase:n | 330 |
| P | | \tl_to_str:f | 10, 143, 193, 220, 275 |
| \prg_do_nothing: | 411 | \tl_to_str:n | 10 |
| \prg_new_conditional:Npnn | 11, 18 | \token_if_eq_charcode:NNF . | 163, 172, 174 |
| \prg_replicate:nn | 94, | \token_if_eq_charcode:NNTF | |
| 101, 110, 120, 133, 337, 354, 369, 385 | | 68, 152, 159, 229, 285 | |
| \prg_return_false: | 14, 21 | \token_if_eq_meaning:NNTF | 80, 288 |
| \prg_return_true: | 14, 27 | U | |
| \ProvidesExplPackage | 4 | \use:fnnf | 8, 108 |
| R | | \use:n | 250, 251, 259 |
| \RequirePackage | 6 | \use:nf | 8, 92, 131, 212, 337, 369, 383 |
| S | | \use:nn | 8 |
| \s__fp | 283, | \use:nnn | 9 |
| 315, 333, 343, 348, 365, 376, 387, 397 | | \use_i:nnnn | 361 |
| | | \use_ii:nnn | 215 |
| | | \use_none:n | 210, 355, 362 |