# siunitx – A comprehensive (SI) units package*

Joseph Wright†

Released 2017/11/26

## Contents



*This file describes v3.0alpha, last revised 2017/11/26.

†E-mail: joseph.wright@morningstar2.co.uk

# Part I
# siunitx – Overall set up

## 1 siunitx implementation

Start the DocStrip guards.

```
1 ⟨*package⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention).

```
2 ⟨@@=siunitx⟩
```

### 1.1 Initial set up

Load only the essential support (expl3) "up-front".

```
3 \RequirePackage{expl3}
```

Make sure that the version of l3kernel in use is sufficiently new. This will also trap any problems with l3packages (as the two are now tied together, version-wise).

```
4 \@ifpackagelater {expl3}{2015/11/15}
5   {}
6   {%
7     \PackageError{siunitx} {Support package expl3 too old}
8       {%
9         You need to update your installation of the bundles 'l3kernel' and
10        'l3packages'.\MessageBreak
11        Loading~siunitx~will~abort!%
12      }%
13    \endinput
14  }%
```

Identify the package and give the over all version information.

```
15 \ProvidesExplPackage {siunitx} {2017/11/26} {3.0alpha}
16   {A comprehensive (SI) units package}
```

### 1.2 Safety checks

__siunitx_load_check:

There are a number of packages that are incompatible with siunitx as they cover the same concepts and in some cases define the same command names. These are all tested at the point of loading to try to trap issues, and a couple are also tested later as it's possible for them to load without an obvious error if siunitx was loaded first.

The testing here is done in a group so that the tests do not add anything to the hash table.

```
17 \msg_new:nnnn { siunitx } { incompatible-package }
18   { Package~'#1'~incompatible. }
19   { The~#1~package~and~siunitx~are~incompatible. }
20 \cs_new_protected:Npn \__siunitx_load_check:n #1
21   {
22     \group_begin:
23       \@ifpackageloaded {#1}
24         {
25           \group_end:
```

```
          \msg_error:nnx { siunitx } { incompatible-package } {#1}
        }
        { \group_end: }
  }
\clist_map_function:nN
  { SIunits , sistyle , unitsdef , fancyunits }
  \__siunitx_load_check:n
\AtBeginDocument
  {
    \clist_map_function:nN { SIunits , sistyle }
      \__siunitx_load_check:n
  }
```

(*End definition for* `\__siunitx_load_check:`.)

## 1.3 Top-level scratch space

`\l__siunitx_tmp_tl` Scratch space for the interfaces.

```
\tl_new:N \l__siunitx_tmp_tl
```

(*End definition for* `\l__siunitx_tmp_tl`.)

## 1.4 User interfaces

The user interfaces are defined in terms of documented code-level ones. This is all done here, and will appear in the `.sty` file before the relevant code. Things could be rearranged by DocStrip but there is no advantage.

User level interfaces are all created by xparse

```
\RequirePackage { xparse }
```

### 1.4.1 Preamble commands

`\DeclareSIPower` `\DeclareSIPrefix` `\DeclareSIQualifier` `\DeclareSIUnit` Pass data to the code layer.

```
\NewDocumentCommand \DeclareSIPower { +m +m m }
  {
    \siunitx_declare_power:NNn #1 #2 {#3}
  }
\NewDocumentCommand \DeclareSIPrefix { +m m m }
  {
    \siunitx_declare_prefix:Nnn #1 {#2} {#3}
  }
\NewDocumentCommand \DeclareSIQualifier { +m m }
  {
    \siunitx_declare_qualifier:Nn #1 {#2}
  }
\NewDocumentCommand \DeclareSIUnit { O { } +m m }
  {
    \siunitx_declare_unit:Nn #2 {#3}
  }
```

(*End definition for* `\DeclareSIPower` *and others. These functions are documented on page* **??**.)

### 1.4.2 Document commands

\qty

```
\NewDocumentCommand \qty { O { } m m }
  {
    \leavevmode
    \group_begin:
      \keys_set:nn { siunitx } {#1}
      \siunitx_number_format:nN {#2} \l__siunitx_tmp_tl
      \siunitx_print:nV { number } \l__siunitx_tmp_tl
      \, \nobreak % TEMP
      \siunitx_unit_format:nN {#3} \l__siunitx_tmp_tl
      \siunitx_print:nV { unit } \l__siunitx_tmp_tl
    \group_end:
  }
```

(*End definition for* `\qty`*. This function is documented on page* **??**.)

\num
\unit

All of a standard form: start a paragraph (if required), set local key values, do the formatting, print the result.

```
\NewDocumentCommand \num { O { } m }
  {
    \leavevmode
    \group_begin:
      \keys_set:nn { siunitx } {#1}
      \siunitx_number_format:nN {#2} \l__siunitx_tmp_tl
      \siunitx_print:nV { number } \l__siunitx_tmp_tl
    \group_end:
  }
\NewDocumentCommand \unit { O { } m }
  {
    \leavevmode
    \group_begin:
      \keys_set:nn { siunitx } {#1}
      \siunitx_unit_format:nN {#2} \l__siunitx_tmp_tl
      \siunitx_print:nV { unit } \l__siunitx_tmp_tl
    \group_end:
  }
```

(*End definition for* `\num` *and* `\unit`*. These functions are documented on page* **??**.)

\sisetup

A very thin wrapper.

```
\NewDocumentCommand \sisetup { m }
  { \keys_set:nn { siunitx } {#1} }
```

(*End definition for* `\sisetup`*. This function is documented on page* **??**.)

### 1.4.3 Table column

User interfaces in tabular constructs are provided using the mechanisms from the array package.

```
\RequirePackage { array }
```

There is a slight problem with the cellspace package: it also uses `S` for a column type Here, `S` seems to make more sense for siunitx, with `C` used for cellspace. To enable this to work well, the column rewriting code happens `\AtBeginDocument`. The `S` column is deleted from `\NC@list` so that no warning appears.

```
\AtBeginDocument
  {
    \@ifpackageloaded { cellspace }
      {
        \newcolumntype { C } [ 1 ]
          { > { \bcolumn #1 \@nil } #1 < { \ecolumn } }
        \cs_set:Npn \__siunitx_tmp:w #1 \NC@do S #2 \q_stop
          { \NC@list { #1 #2 } }
        \exp_after:wN \__siunitx_tmp:w \tex_the:D \NC@list \q_stop
        \cs_undefine:N \NC@find@S
        \msg_warning:nnn { siunitx } { moved-cellspace-column } { C }
        \ifcellspace@m
          \def \env@matrix
            {
              \hskip -\arraycolsep
              \let \@ifnextchar \new@ifnextchar
              \array
                {
                  * { \c@MaxMatrixCols }
                    { > { \bcolumn c \@nil $ } c < { $ \ecolumn } } @ { }
                }
            }
        \fi
      }
      { }
  }
\msg_new:nnn { siunitx } { moved-cellspace-column }
  { Column~type~for~cellspace~package~moved~to~'#1'. }
```

`\__siunitx_declare_column:Nnn` Creating numerical columns requires that these are declared before anything else in `\NC@list`: this is necessary to work with optional arguments. This means a bit of manual effort after the simple declaration of a new column type. The token assigned to the column type is not fixed as this allows the same code to be used in compatibility with version 2.

```
\cs_new_protected:Npn \__siunitx_declare_column:Nnn #1#2#3
  {
    \newcolumntype {#1} { }
    \cs_set_protected:Npn \__siunitx_tmp:w \NC@do ##1##2 \NC@do #1
      { \NC@list { \NC@do ##1 \NC@do #1 ##2 } }
    \exp_after:wN \__siunitx_tmp:w \tex_the:D \NC@list
    \exp_args:NNc \renewcommand * { NC@rewrite@ #1 } [ 1 ] [ ]
      {
        \@temptokena \expandafter
          {
            \the \@temptokena
            > {#2} c < {#3}
          }
        \NC@find
      }
```

```
  }
```

When mdwtab is loaded the syntax required is slightly different.

```
\AtBeginDocument
  {
    \@ifpackageloaded { mdwtab }
      {
        \cs_set_protected:Npn \__siunitx_declare_column:Nnn #1#2#3
          {
            \newcolumntype {#1} [ 1 ] [ ]
              { > {#2} c < {#3} }
          }
      }
      { }
  }
\AtBeginDocument
  {
    \__siunitx_declare_column:Nnn n
      {
        \keys_set:nn { siunitx } {#1}
        \siunitx_cell_begin:
      }
      { \siunitx_cell_end: }
  }
```

(*End definition for* `\__siunitx_declare_column:Nnn`.)

```
⟨/package⟩
```

# Part II
# siunitx-angle – Formatting angles

## 1 siunitx-angle implementation

Start the DocStrip guards.

```
1 ⟨*package⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention): only internal material in this *submodule* should be used directly.

```
2 ⟨@@=siunitx_table⟩
```

```
3 ⟨/package⟩
```

# Part III
# siunitx-number – Parsing and formatting numbers

\siunitx_number_format:nN
\siunitx_number_format:VN
\siunitx_number_format:nN {⟨number⟩} ⟨tl var⟩

\siunitx_number_format:nNN
\siunitx_number_format:nNN {⟨number⟩} ⟨tl var⟩ ⟨marker⟩

\siunitx_if_number:nTF
\siunitx_if_number_token:NTF {⟨tokens⟩}
{⟨true code⟩} {⟨false code⟩}

Determines if the ⟨*tokens*⟩ form a valid number which can be fully parsed by siunitx.

\siunitx_if_number_token:NTF
\siunitx_if_number_token:NTF {⟨token⟩}
{⟨true code⟩} {⟨false code⟩}

Determines if the ⟨*token*⟩ is valid in a number based on those tokens currently set up for detection in a number.

## 1 siunitx-number implementation

Start the DocStrip guards.

```
1 ⟨*package⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention): only internal material in this *submodule* should be used directly.

```
2 ⟨@@=siunitx_number⟩
```

### 1.1 Initial set-up

Variants not provided by expl3.

```
3 \cs_generate_variant:Nn \tl_if_blank:nTF { f }
4 \cs_generate_variant:Nn \tl_if_blank_p:n { f }
5 \cs_generate_variant:Nn \tl_if_in:NnTF { NV }
```

\l__siunitx_number_tmp_tl

Scratch space.

```
6 \tl_new:N \l__siunitx_number_tmp_tl
```

(*End definition for* \l__siunitx_number_tmp_tl.)

## 1.2 Main formatting routine

\l__siunitx_number_formatted_tl

A token list for the final formatted result: may or may not be generated by the parser, depending on settings which are active.

```
\tl_new:N \l__siunitx_number_formatted_tl
```

(*End definition for* `\l__siunitx_number_formatted_tl`.)

\l__siunitx_number_tab_tl

A token list for marking the position of tabular alignments in formatted output.

```
\tl_new:N \l__siunitx_number_tab_tl
```

(*End definition for* `\l__siunitx_number_tab_tl`.)

\siunitx_number_format:nN
\siunitx_number_format:VN
\siunitx_number_format:nNN
__siunitx_number_format:nN

```
\cs_new_protected:Npn \siunitx_number_format:nN #1#2
  {
    \tl_clear:N \l__siunitx_number_tab_tl
    \__siunitx_number_format:nN {#1} #2
  }
\cs_generate_variant:Nn \siunitx_number_format:nN { V }
\cs_new_protected:Npn \siunitx_number_format:nNN #1#2#3
  {
    \tl_set:Nn \l__siunitx_number_tab_tl {#3}
    \__siunitx_number_format:nN {#1} #2
  }
\cs_new_protected:Npn \__siunitx_number_format:nN #1#2
  {
    \group_begin:
      \__siunitx_number_parse:n {#1}
      \__siunitx_number_format:
    \exp_args:NNNV \group_end:
    \tl_set:Nn #2 \l__siunitx_number_formatted_tl
  }
```

(*End definition for* `\siunitx_number_format:nN`, `\siunitx_number_format:nNN`, *and* `\__siunitx_number_format:nN`. *These functions are documented on page* 7.)

## 1.3 Parsing numbers

Before numbers can be manipulated or formatted they need to be parsed into an internal form. In particular, if multiple code paths are to be avoided, it is necessary to do such parsing even for relatively simple cases such as converting `1e10` to `1 \times 10^{10}`.

Storing the result of such parsing can be done in a number of ways. In the first version of siunitx a series of separate data stores were used. This is potentially quite fast (as recovery of items relies only on TEX's hash table) but makes managing the various data entries somewhat tedious and error-prone. For version two of the package, a single data structure (property list) was used for each part of the parsed number. Whilst this is easy to manage and extend, it is somewhat slower as at a TEX level there are repeated pack–unpack steps. In particular, the fact that there are a limited number of items to track for a "number" means that a more efficient approach is desirable (contrast parsing units, which is open-ended and therefore fits well with using a property list).

To allow for complex numbers, two parallel data structures are used, one for the real part and one for the imaginary part. If the part is entirely absent then the data structures are left empty. Within each part, the structure is

{⟨*comparator*⟩}⟨*sign*⟩{⟨*integer*⟩}{⟨*decimal*⟩} {⟨*uncertainty*⟩}
⟨*exponent sign*⟩{⟨*exponent*⟩}

where the two sign parts must be single tokens and all other components must be given in braces. *All* of the components must be present in a stored number (*i.e.* at the end of parsing). The number must have at least one digit for both the ⟨*integer*⟩ and ⟨*exponent*⟩ parts.

\l__siunitx_number_expression_bool
\l__siunitx_number_input_uncert_close_tl
\l__siunitx_number_input_complex_tl
\l__siunitx_number_input_comparator_tl
\l__siunitx_number_input_decimal_tl
\l__siunitx_number_input_digit_tl
\l__siunitx_number_input_exponent_tl
\l__siunitx_number_input_ignore_tl
\l__siunitx_number_input_uncert_open_tl
\l__siunitx_number_input_sign_tl
\l__siunitx_number_input_uncert_sign_tl
__siunitx_number_expression:n

Options which determine the various valid parts of a parsed number.

```
\keys_define:nn { siunitx }
  {
    evaluate-expression .bool_set:N =
      \l__siunitx_number_expression_bool ,
    expression .code:n     =
      \cs_set:Npn \__siunitx_number_expression:n ##1 {#1} ,
    input-close-uncertainty .tl_set:N =
      \l__siunitx_number_input_uncert_close_tl ,
    input-complex-roots .tl_set:N =
      \l__siunitx_number_input_complex_tl ,
    input-comparators .tl_set:N =
      \l__siunitx_number_input_comparator_tl ,
    input-decimal-markers .tl_set:N =
      \l__siunitx_number_input_decimal_tl ,
    input-digits .tl_set:N =
      \l__siunitx_number_input_digit_tl ,
    input-exponent-markers .tl_set:N =
      \l__siunitx_number_input_exponent_tl ,
    input-ignore .tl_set:N =
      \l__siunitx_number_input_ignore_tl ,
    input-open-uncertainty .tl_set:N =
      \l__siunitx_number_input_uncert_open_tl  ,
    input-signs .tl_set:N   =
      \l__siunitx_number_input_sign_tl ,
    input-uncertainty-signs .code:n =
      {
        \tl_set:Nn \l__siunitx_number_input_uncert_sign_tl {#1}
        \tl_map_inline:nn {#1}
          {
            \tl_if_in:NnF \l__siunitx_number_input_sign_tl {##1}
              { \tl_put_right:Nn \l__siunitx_number_input_sign_tl {##1} }
          }
      }
  }
\cs_new:Npn \__siunitx_number_expression:n #1 { }
```

(*End definition for* `\l__siunitx_number_expression_bool` *and others.*)

\l__siunitx_number_arg_tl

The input argument or a part thereof, depending on the position in the parsing routine.

```
\tl_new:N \l__siunitx_number_arg_tl
```

(*End definition for* `\l__siunitx_number_arg_tl`.)

\l__siunitx_number_comparator_tl

A comparator, if found, is held here.

```
\tl_new:N \l__siunitx_number_comparator_tl
```

(*End definition for* `\l__siunitx_number_comparator_tl`.)

`\l__siunitx_number_exponent_tl` The exponent part of a parsed number. It is easiest to find this relatively early in the parsing process, but as it needs to go at the end of the internal format is held separately until required.

```
\tl_new:N \l__siunitx_number_exponent_tl
```

(*End definition for* `\l__siunitx_number_exponent_tl`.)

`\l__siunitx_number_flex_tl` When parsing for a separate uncertainty or complex number, the nature of the grabbed part cannot be determined until the end of the number. To avoid abusing the storage areas, this dedicated one is used for "flexible" cases.

```
\tl_new:N \l__siunitx_number_flex_tl
```

(*End definition for* `\l__siunitx_number_flex_tl`.)

`\l__siunitx_number_imaginary_tl` Used to hold the real and imaginary parts of a number in the standardised format.
`\l__siunitx_number_real_tl`

```
\tl_new:N \l__siunitx_number_imaginary_tl
\tl_new:N \l__siunitx_number_real_tl
```

(*End definition for* `\l__siunitx_number_imaginary_tl` *and* `\l__siunitx_number_real_tl`.)

`\l__siunitx_number_input_tl` The numerical input exactly as given by the user.

```
\tl_new:N \l__siunitx_number_input_tl
```

(*End definition for* `\l__siunitx_number_input_tl`.)

`\l__siunitx_number_partial_tl` To avoid needing to worry about the fact that the final data stores are somewhat tricky to add to token-by-token, a simple store is used to build up the parsed part of a number before transferring in one go.

```
\tl_new:N \l__siunitx_number_partial_tl
```

(*End definition for* `\l__siunitx_number_partial_tl`.)

`\l__siunitx_number_validate_bool` Used to set up for validation with no error production.

```
\bool_new:N \l__siunitx_number_validate_bool
```

(*End definition for* `\l__siunitx_number_validate_bool`.)

`\__siunitx_number_parse:n` After some initial set up, the parser expands the input and then replaces as far as possible tricky tokens with ones that can be handled using delimited arguments. The parser begins with the assumption that the input is a real number. To avoid multiple conditionals here, the parser is set up as a chain of commands initially, with a loop only later. This avoids more conditionals than are necessary.

```
\q
\cs_new_protected:Npn \__siunitx_number_parse:n #1
  {
    \tl_clear:N \l__siunitx_number_imaginary_tl
    \tl_clear:N \l__siunitx_number_real_tl
    \protected@edef \l__siunitx_number_arg_tl
      {
        \bool_if:NTF \l__siunitx_number_expression_bool
          { \fp_eval:n { \__siunitx_number_expression:n {#1} } }
          {#1}
```

```
      }
    \tl_set_eq:NN \l__siunitx_number_input_tl \l__siunitx_number_arg_tl
    \__siunitx_number_parse_replace:
    \tl_if_empty:NF \l__siunitx_number_arg_tl
      { \__siunitx_number_parse_comparator: }
    \__siunitx_number_parse_check:
  }
```

(*End definition for* `\__siunitx_number_parse:n`.)

`\__siunitx_number_parse_check:` After the loop there is one case that might need tidying up. If a separated uncertainty was found it will be currently in `\l__siunitx_number_flex_tl` and needs moving. A series of tests pick up that case, then the check is made that some content was found for at least one of the real or imaginary parts of the number.

```
\cs_new_protected:Npn \__siunitx_number_parse_check:
  {
    \tl_if_empty:NF \l__siunitx_number_flex_tl
      {
        \bool_lazy_and:nnTF
          {
            \tl_if_blank_p:f
              { \exp_after:wN \use_iv:nnnn \l__siunitx_number_real_tl }
          }
          {
             \tl_if_blank_p:f
               { \exp_after:wN \use_iv:nnnn \l__siunitx_number_flex_tl }
          }
          {
            \tl_set:Nx \l__siunitx_number_tmp_tl
              { \exp_after:wN \use_i:nnnn \l__siunitx_number_flex_tl }
            \tl_if_in:NVTF \l__siunitx_number_input_uncert_sign_tl
              \l__siunitx_number_tmp_tl
              { \__siunitx_number_parse_combine_uncert: }
              { \tl_clear:N \l__siunitx_number_real_tl }
          }
          { \tl_clear:N \l__siunitx_number_real_tl }
      }
    \bool_lazy_and:nnTF
      { \tl_if_empty_p:N \l__siunitx_number_real_tl  }
      { \tl_if_empty_p:N \l__siunitx_number_imaginary_tl }
      {
        \bool_if:NF \l__siunitx_number_validate_bool
          {
            \msg_error:nnx { siunitx } { number / invalid-input }
              { \exp_not:V \l__siunitx_number_input_tl }
          }
      }
      { \__siunitx_number_parse_finalise: }
  }
```

(*End definition for* `\__siunitx_number_parse_check:`.)

`\__siunitx_number_parse_combine_uncert:`
`tx_number_parse_combine_uncert_auxi:NnnnNnnn`
`nitx_number_parse_combine_uncert_auxii:nnnnn`
`nitx_number_parse_combine_uncert_auxii:fnnnn`
`tx_number_parse_combine_uncert_auxiii:nnnnnn`
`tx_number_parse_combine_uncert_auxiii:fnnnnn`
`unitx_number_parse_combine_uncert_auxiv:nnnn`
`__siunitx_number_parse_combine_uncert_auxv:w`
`_siunitx_number_parse_combine_uncert_auxvi:w`
Conversion of a second numerical part to an uncertainty needs a bit of work. The first step is to extract the useful information from the two stores: the sign, integer and decimal

parts from the real number and the integer and decimal parts from the second number. That is done using the input stack to avoid lots of assignments.

```
\cs_new_protected:Npn \__siunitx_number_parse_combine_uncert:
  {
    \exp_after:wN \exp_after:wN \exp_after:wN
    \__siunitx_number_parse_combine_uncert_auxi:NnnnNnnn
      \exp_after:wN \l__siunitx_number_real_tl \l__siunitx_number_flex_tl
  }
```

Here, `#4`, `#5` and `#8` are all junk arguments simply there to mop up tokens, while `#1` will be recovered later from `\l__siunitx_number_real_tl` so does not need to be passed about. The difference in places between the two decimal parts is now found: this is done just once to avoid having to parse token lists twice. The value is then used to generate a number of filler `0` tokens, and these are added to the appropriate part of the number. Finally, everything is recombined: the integer part only needs a test to avoid an empty main number.

```
\cs_new_protected:Npn
  \__siunitx_number_parse_combine_uncert_auxi:NnnnNnnn #1#2#3#4#5#6#7#8
  {
    \int_compare:nNnTF { \tl_count:n {#6} } > { \tl_count:n {#2} }
      {
        \tl_clear:N \l__siunitx_number_real_tl
        \tl_clear:N \l__siunitx_number_flex_tl
      }
      {
        \__siunitx_number_parse_combine_uncert_auxii:fnnnn
          { \int_eval:n { \tl_count:n {#3} - \tl_count:n {#7} } }
          {#2} {#3} {#6} {#7}
      }
  }
\cs_new_protected:Npn
  \__siunitx_number_parse_combine_uncert_auxii:nnnnn #1
  {
    \__siunitx_number_parse_combine_uncert_auxiii:fnnnnn
      { \prg_replicate:nn { \int_abs:n {#1} } { 0 } }
      {#1}
  }
\cs_generate_variant:Nn \__siunitx_number_parse_combine_uncert_auxii:nnnnn { f }
\cs_new_protected:Npn
  \__siunitx_number_parse_combine_uncert_auxiii:nnnnnn #1#2#3#4#5#6
  {
    \int_compare:nNnTF {#2} > 0
      {
        \__siunitx_number_parse_combine_uncert_auxiv:nnnn
          {#3} {#4} {#5} { #6 #1 }
      }
      {
        \__siunitx_number_parse_combine_uncert_auxiv:nnnn
          {#3} { #4 #1 } {#5} {#6}
      }
  }
\cs_generate_variant:Nn
  \__siunitx_number_parse_combine_uncert_auxiii:nnnnnn { f }
\cs_new_protected:Npn
```

```
\__siunitx_number_parse_combine_uncert_auxiv:nnnn #1#2#3#4
{
  \tl_set:Nx \l__siunitx_number_real_tl
    {
      \tl_head:V \l__siunitx_number_real_tl
        { \exp_not:n {#1} }
        {
          \bool_lazy_and:nnTF
            { \tl_if_blank_p:n {#2} }
            { ! \tl_if_blank_p:n {#4} }
            { 0 }
            { \exp_not:n {#2} }
        }
        {
          \__siunitx_number_parse_combine_uncert_auxv:w #3#4
            \q_recursion_tail \q_recursion_stop
        }
    }
}
```

A short routine to remove any leading zeros in the uncertainty part, which are not needed for the compact representation used by the module.

```
\cs_new:Npn \__siunitx_number_parse_combine_uncert_auxv:w #1
  {
    \quark_if_recursion_tail_stop:N #1
    \str_if_eq:nnTF {#1} { 0 }
      { \__siunitx_number_parse_combine_uncert_auxv:w }
      { \__siunitx_number_parse_combine_uncert_auxvi:w #1 }
  }
\cs_new:Npn \__siunitx_number_parse_combine_uncert_auxvi:w
  #1 \q_recursion_tail \q_recursion_stop
  { \exp_not:n {#1} }
```

(*End definition for* `\__siunitx_number_parse_combine_uncert:` *and others.*)

`\__siunitx_number_parse_comparator:`
`\__siunitx_number_parse_comparator_aux:Nw`

A comparator has to be the very first token in the input. A such, the test for this can be very fast: grab the first token, do a check and if appropriate store the result.

```
\cs_new_protected:Npn \__siunitx_number_parse_comparator:
  {
    \exp_after:wN \__siunitx_number_parse_comparator_aux:Nw
      \l__siunitx_number_arg_tl \q_stop
  }
\cs_new_protected:Npn \__siunitx_number_parse_comparator_aux:Nw #1#2 \q_stop
  {
    \tl_if_in:NnTF \l__siunitx_number_input_comparator_tl {#1}
      {
        \tl_set:Nn \l__siunitx_number_comparator_tl {#1}
        \tl_set:Nn \l__siunitx_number_arg_tl {#2}
      }
      { \tl_clear:N \l__siunitx_number_comparator_tl }
    \tl_if_empty:NF \l__siunitx_number_arg_tl
      { \__siunitx_number_parse_sign: }
  }
```

(*End definition for* `\__siunitx_number_parse_comparator:` *and* `\__siunitx_number_parse_comparator_aux:Nw`.)

__siunitx_number_parse_exponent:
__siunitx_number_parse_exponent_aux:w
__siunitx_number_parse_exponent_aux:nn
__siunitx_number_parse_exponent_aux:Nw
__siunitx_number_parse_exponent_aux:Nn
__siunitx_number_parse_exponent_zero_test:N
__siunitx_number_parse_exponent_check:N
__siunitx_number_parse_exponent_cleanup:N

An exponent part of a number has to come at the end and can only occur once. Thus it is relatively easy to parse. First, there is a check that an exponent part is allowed, and if so a split is made (the previous part of the chain checks that there is some content in `\l__siunitx_number_arg_tl` before calling this function). After splitting, if there is no exponent then simply save a default. Otherwise, check for a sign and then store either this or an assumed `+` and the digits after a check that nothing else is present after the `e`. The only slight complication to all of this is allowing an arbitrary token in the input to represent the exponent: this is done by setting any exponent tokens to the first of the allowed list, then using that in a delimited argument set up. Once an exponent part is found, there is a loop to check that each of the tokens is a digit then a tidy up step to remove any leading zeros.

```
\cs_new_protected:Npn \__siunitx_number_parse_exponent:
  {
    \tl_if_empty:NTF \l__siunitx_number_input_exponent_tl
      { \tl_set:Nn \l__siunitx_number_exponent_tl { +0 } }
      {
        \tl_set:Nx \l__siunitx_number_tmp_tl
          { \tl_head:V \l__siunitx_number_input_exponent_tl }
        \tl_map_inline:Nn \l__siunitx_number_input_exponent_tl
          {
            \tl_replace_all:NnV \l__siunitx_number_arg_tl
              {##1} \l__siunitx_number_tmp_tl
          }
        \use:x
          {
            \cs_set_protected:Npn
              \exp_not:N \__siunitx_number_parse_exponent_aux:w
              ####1 \exp_not:V \l__siunitx_number_tmp_tl
              ####2 \exp_not:V \l__siunitx_number_tmp_tl
              ####3 \exp_not:N \q_stop
          }
            { \__siunitx_number_parse_exponent_aux:nn {##1} {##2} }
        \use:x
          {
            \__siunitx_number_parse_exponent_aux:w
              \exp_not:V \l__siunitx_number_arg_tl
              \exp_not:V \l__siunitx_number_tmp_tl \exp_not:N \q_nil
              \exp_not:V \l__siunitx_number_tmp_tl \exp_not:N \q_stop
          }
      }
  }
\cs_new_protected:Npn \__siunitx_number_parse_exponent_aux:w  { }
\cs_new_protected:Npn \__siunitx_number_parse_exponent_aux:nn #1#2
  {
    \quark_if_nil:nTF {#2}
      { \tl_set:Nn \l__siunitx_number_exponent_tl { +0 } }
      {
        \tl_set:Nn \l__siunitx_number_arg_tl {#1}
        \tl_if_blank:nTF {#2}
          { \tl_clear:N \l__siunitx_number_real_tl }
          { \__siunitx_number_parse_exponent_aux:Nw #2 \q_stop }
      }
    \tl_if_empty:NF \l__siunitx_number_real_tl
```

```
      { \__siunitx_number_parse_loop: }
  }
\cs_new_protected:Npn \__siunitx_number_parse_exponent_aux:Nw #1#2 \q_stop
  {
    \tl_if_in:NnTF \l__siunitx_number_input_sign_tl {#1}
      { \__siunitx_number_parse_exponent_aux:Nn #1 {#2} }
      { \__siunitx_number_parse_exponent_aux:Nn + {#1#2} }
    \tl_if_empty:NT \l__siunitx_number_exponent_tl
      { \tl_clear:N \l__siunitx_number_real_tl }
  }
\cs_new_protected:Npn \__siunitx_number_parse_exponent_aux:Nn #1#2
  {
    \tl_set:Nn \l__siunitx_number_exponent_tl { #1 }
    \tl_if_blank:nTF {#2}
      { \tl_clear:N \l__siunitx_number_real_tl }
      {
        \__siunitx_number_parse_exponent_zero_test:N #2
          \q_recursion_tail \q_recursion_stop
      }
  }
\cs_new_protected:Npn \__siunitx_number_parse_exponent_zero_test:N #1
  {
    \quark_if_recursion_tail_stop_do:Nn #1
      { \tl_set:Nn \l__siunitx_number_exponent_tl { +0 } }
    \str_if_eq:nnTF {#1} { 0 }
      { \__siunitx_number_parse_exponent_zero_test:N }
      { \__siunitx_number_parse_exponent_check:N #1 }
  }
\cs_new_protected:Npn \__siunitx_number_parse_exponent_check:N #1
  {
    \quark_if_recursion_tail_stop:N #1
    \tl_if_in:NnTF \l__siunitx_number_input_digit_tl {#1}
      {
        \tl_put_right:Nn \l__siunitx_number_exponent_tl {#1}
        \__siunitx_number_parse_exponent_check:N
      }
      { \__siunitx_number_parse_exponent_cleanup:wN }
  }
\cs_new_protected:Npn \__siunitx_number_parse_exponent_cleanup:wN
  #1 \q_recursion_stop
  { \tl_clear:N \l__siunitx_number_real_tl }
```

There are two parts to the replacement code. First, any active hyphens signs are normalised: these can come up with some packages and cause issues. Multi-token signs then are converted to the single token equivalents so that everything else can work on a one token basis.

__siunitx_number_parse_replace:
__siunitx_number_parse_replace_aux:nN
__siunitx_number_parse_replace_sign:
\c__siunitx_number_parse_sign_replacement_tl

```
\cs_new_protected:Npn \__siunitx_number_parse_replace:
  {
    \__siunitx_number_parse_replace_minus:
    \exp_last_unbraced:NV \__siunitx_number_parse_replace_aux:nN
      \c__siunitx_number_parse_sign_replacement_tl
      { ? } \q_recursion_tail
        \q_recursion_stop
  }
```

```
\cs_set_protected:Npn \__siunitx_number_parse_replace_aux:nN #1#2
  {
    \quark_if_recursion_tail_stop:N #2
    \tl_replace_all:Nnn \l__siunitx_number_arg_tl {#1} {#2}
    \__siunitx_number_parse_replace_aux:nN
  }
\tl_const:Nn \c__siunitx_number_parse_sign_replacement_tl
  {
    { -+ } \mp
    { +- } \pm
    { << } \ll
    { <= } \le
    { >> } \gg
    { >= } \ge
  }
\group_begin:
  \char_set_catcode_active:N \-
  \cs_new_protected:Npx \__siunitx_number_parse_replace_minus:
    {
      \tl_replace_all:Nnn \exp_not:N \l__siunitx_number_arg_tl
        { \exp_not:N - }  { \token_to_str:N - }
    }
\group_end:
```

(*End definition for* `\__siunitx_number_parse_exponent:` *and others.*)

`\__siunitx_number_parse_finalise:`
`\__siunitx_number_parse_finalise_aux:N`
`\__siunitx_number_parse_finalise_aux:Nw`

Combine all of the bits of a number together: both the real and imaginary parts contain all of the data.

```
\cs_new_protected:Npn \__siunitx_number_parse_finalise:
  {
    \__siunitx_number_parse_finalise_aux:N \l__siunitx_number_real_tl
    \__siunitx_number_parse_finalise_aux:N \l__siunitx_number_imaginary_tl
  }
\cs_new_protected:Npn \__siunitx_number_parse_finalise_aux:N #1
  {
    \tl_if_empty:NF #1
      {
        \tl_set:Nx #1
          {
            { \exp_not:V \l__siunitx_number_comparator_tl }
            \exp_not:V #1
            \exp_after:wN \__siunitx_number_parse_finalise_aux:Nw
              \l__siunitx_number_exponent_tl \q_stop
          }
      }
  }
\cs_new:Npn \__siunitx_number_parse_finalise_aux:Nw #1#2 \q_stop
  {
    \exp_not:N #1
    { \exp_not:n {#2} }
  }
```

(*End definition for* `\__siunitx_number_parse_finalise:`, `\__siunitx_number_parse_finalise_aux:N`, *and* `\__siunitx_number_parse_finalise_aux:Nw`.)

__siunitx_number_parse_loop:
__siunitx_number_parse_loop_first:N
__siunitx_number_parse_loop_main:NNNNN
__siunitx_number_parse_loop_main_end:NN
__siunitx_number_parse_loop_main_digit:NNNNN
__siunitx_number_parse_loop_main_decimal:NN
__siunitx_number_parse_loop_main_uncert:NNN
__siunitx_number_parse_loop_main_complex:N
__siunitx_number_parse_loop_main_sign:NNN
__siunitx_number_parse_loop_main_store:NNN
__siunitx_number_parse_loop_after_decimal:NNN
__siunitx_number_parse_loop_uncert:NNNNN
__siunitx_number_parse_loop_after_uncert:NNN
__siunitx_number_parse_loop_root_swap:NNwNN
__siunitx_number_parse_loop_complex_cleanup:wN
__siunitx_number_parse_loop_break:wN

At this stage, the partial input `\l__siunitx_number_arg_tl` will contain any mantissa, which may contain an uncertainty or complex part. Parsing this and allowing for all of the different formats possible is best done using a token-by-token approach. However, as at each stage only a subset of tokens are valid, the approach take is to use a set of semi-dedicated functions to parse different components along with switches to allow a sensible amount of code sharing.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop:
  {
    \tl_clear:N \l__siunitx_number_partial_tl
    \exp_after:wN \__siunitx_number_parse_loop_first:NNN
      \exp_after:wN \l__siunitx_number_real_tl \exp_after:wN \c_true_bool
        \l__siunitx_number_arg_tl
        \q_recursion_tail \q_recursion_stop
  }
```

The very first token of the input is handled with a dedicated function. Valid cases here are

- Entirely blank if the original input was for example `+e10`: simply clean up if in the integer part of issue an error if in a second part (complex number, *etc.*).
- An integer part digit: pass through to the main collection routine.
- A decimal marker: store an empty integer part and move to the main collection routine for a decimal part.
- A complex root token: shuffle to the end of the input.

Anything else is invalid and sends the code to the abort function.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_first:NNN #1#2#3
  {
    \quark_if_recursion_tail_stop_do:Nn #3
      {
        \bool_if:NTF #2
          { \tl_put_right:Nn #1 { { 1 } { } { } } }
          { \__siunitx_number_parse_loop_break:wN \q_recursion_stop }
      }
    \tl_if_in:NnTF \l__siunitx_number_input_digit_tl {#3}
      {
        \__siunitx_number_parse_loop_main:NNNNN
          #1 \c_true_bool \c_false_bool #2 #3
      }
      {
        \tl_if_in:NnTF \l__siunitx_number_input_decimal_tl {#3}
          {
            \tl_put_right:Nn #1 { { 0 } }
            \__siunitx_number_parse_loop_after_decimal:NNN #1 #2
          }
          {
            \tl_if_in:NnTF \l__siunitx_number_input_complex_tl {#3}
              { \__siunitx_number_parse_loop_root_swap:NNwNN #1 #3 }
              { \__siunitx_number_parse_loop_break:wN }
          }
      }
  }
```

A single function is used to cover the "main" part of numbers: finding real, complex or separated uncertainty parts and covering both the integer and decimal components. This works because these elements share a lot of concepts: a small number of switches can be used to differentiate between them. To keep the code at least somewhat readable, this main function deals with the validity testing but hands off other tasks to dedicated auxiliaries for each case.

The possibilities are

- The number terminates, meaning that some digits were collected and everything is simply tidied up (as far as the loop is concerned).
- A digit is found: this is the common case and leads to a storage auxiliary (which handles non-significant zeros).
- A decimal marker is found: only valid in the integer part and there leading to a store-and-switch situation.
- An open-uncertainty token: switch to the dedicated collector for uncertainties.
- A complex root token: store the current number as an imaginary part and terminate the loop.
- A sign token (if allowed): stop collecting this number and restart collection for the second part.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main:NNNNN #1#2#3#4#5
  {
    \quark_if_recursion_tail_stop_do:Nn #5
      { \__siunitx_number_parse_loop_main_end:NN #1#2 }
    \tl_if_in:NnTF \l__siunitx_number_input_digit_tl {#5}
      { \__siunitx_number_parse_loop_main_digit:NNNNN #1#2#3#4#5 }
      {
        \tl_if_in:NnTF \l__siunitx_number_input_decimal_tl {#5}
          {
            \bool_if:NTF #2
              { \__siunitx_number_parse_loop_main_decimal:NN #1 #4 }
              { \__siunitx_number_parse_loop_break:wN }
          }
          {
            \tl_if_in:NnTF \l__siunitx_number_input_uncert_open_tl {#5}
              { \__siunitx_number_parse_loop_main_uncert:NNN #1#2 #4 }
              {
                \tl_if_in:NnTF \l__siunitx_number_input_complex_tl {#5}
                  {
                    \__siunitx_number_parse_loop_main_store:NNN
                      #1 #2 \c_true_bool
                    \__siunitx_number_parse_loop_main_complex:N #1
                  }
                  {
                    \bool_if:NTF #4
                      {
                        \tl_if_in:NnTF \l__siunitx_number_input_sign_tl {#5}
                          {
                            \__siunitx_number_parse_loop_main_sign:NNN
                              #1#2 #5
```

```
                        }
                        { \__siunitx_number_parse_loop_break:wN }
                    }
                    { \__siunitx_number_parse_loop_break:wN }
                }
            }
        }
    }
}
```

If the main loop finds the end marker then there is a tidy up phase. The current partial number is stored either as the integer or decimal, depending on the setting for the indicator switch. For the integer part, if no number has been collected then one or more non-significant zeros have been dropped. Exactly one zero is therefore needed to make sure the parsed result is correct.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main_end:NN #1#2
  {
    \bool_lazy_and:nnT
      {#2} { \tl_if_empty_p:N \l__siunitx_number_partial_tl }
      { \tl_set:Nn \l__siunitx_number_partial_tl { 0 } }
    \tl_put_right:Nx #1
      {
        { \exp_not:V \l__siunitx_number_partial_tl }
        \bool_if:NT #2 { { } }
        { }
      }
  }
```

The most common case for the main loop collector is to find a digit. Here, in the integer part it is possible that zeros are non-significant: that is handled using a combination of a switch and a string test. Other than that, the situation here is simple: store the input and loop.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main_digit:NNNNN #1#2#3#4#5
  {
    \bool_lazy_or:nnTF
      {#3} { ! \str_if_eq_p:nn {#5} { 0 } }
      {
        \tl_put_right:Nn \l__siunitx_number_partial_tl {#5}
        \__siunitx_number_parse_loop_main:NNNNN #1 #2 \c_true_bool #4
      }
      { \__siunitx_number_parse_loop_main:NNNNN #1 #2 \c_false_bool #4 }
  }
```

When a decimal marker was found, move the integer part to the store and then go back to the loop with the flags set correctly. There is the case of non-significant zeros to cover before that, of course.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main_decimal:NN #1#2
  {
    \__siunitx_number_parse_loop_main_store:NNN #1 \c_false_bool \c_false_bool
    \__siunitx_number_parse_loop_after_decimal:NNN #1 #2
  }
```

Starting an uncertainty part means storing the number to date as in other cases, with the possibility of a blank decimal part allowed for. The uncertainty itself is collected by a dedicated function as it is extremely restricted.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main_uncert:NNN #1#2#3
  {
    \__siunitx_number_parse_loop_main_store:NNN #1 #2 \c_false_bool
    \__siunitx_number_parse_loop_uncert:NNNNN
      #1 \c_true_bool \c_false_bool #3
  }
```

A complex root token has to be at the end of the input (leading ones are dealt with specially). Thus after moving the data to the correct place there is a hand-off to a cleanup function. The case where only the complex root token was given is covered by `\__siunitx_number_parse_loop_root_swap:NNwNN`.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main_complex:N #1
  {
    \tl_set_eq:NN \l__siunitx_number_imaginary_tl #1
    \tl_clear:N #1
    \__siunitx_number_parse_loop_complex_cleanup:wN
  }
```

If a sign is found, terminate the current number, store the sign as the first token of the second part and go back to do the dedicated first-token function.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main_sign:NNN #1#2#3
  {
    \__siunitx_number_parse_loop_main_store:NNN #1 #2 \c_true_bool
    \tl_set:Nn \l__siunitx_number_flex_tl {#3}
    \__siunitx_number_parse_loop_first:NNN
      \l__siunitx_number_flex_tl \c_false_bool
  }
```

A common auxiliary for the various non-digit token functions: tidy up the integer and decimal parts of a number. Here, the two flags are used to indicate if empty decimal and uncertainty parts should be included in the storage cycle.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_main_store:NNN #1#2#3
  {
    \tl_if_empty:NT \l__siunitx_number_partial_tl
      { \tl_set:Nn \l__siunitx_number_partial_tl { 0 } }
    \tl_put_right:Nx #1
      {
        { \exp_not:V \l__siunitx_number_partial_tl }
        \bool_if:NT #2 { { } }
        \bool_if:NT #3 { { } }
      }
    \tl_clear:N \l__siunitx_number_partial_tl
  }
```

After a decimal marker there has to be a digit if there wasn't one before it. That is handled by using a dedicated function, which checks for an empty integer part first then either simply hands off or looks for a digit.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_after_decimal:NNN #1#2#3
  {
    \tl_if_blank:fTF { \exp_after:wN \use_none:n #1 }
      {
        \quark_if_recursion_tail_stop_do:Nn #3
          { \__siunitx_number_parse_loop_break:wN \q_recursion_stop }
        \tl_if_in:NnTF \l__siunitx_number_input_digit_tl {#1}
          {
```

```
            \tl_put_right:Nn \l__siunitx_number_partial_tl {#3}
            \__siunitx_number_parse_loop_main:NNNNN
              #1 \c_false_bool \c_true_bool #2
          }
          { \__siunitx_number_parse_loop_break:wN }
      }
      {
        \__siunitx_number_parse_loop_main:NNNNN
          #1 \c_false_bool \c_true_bool #2 #3
      }
  }
```

Inside the brackets for an uncertainty the range of valid choices is very limited. Either the token is a digit, in which case there is a test to look for non-significant zeros, or it is a closing bracket. The latter is not valid for the very first token, which is handled using a switch (it's a simple enough difference).

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_uncert:NNNNN #1#2#3#4#5
  {
    \quark_if_recursion_tail_stop_do:Nn #5
      { \__siunitx_number_parse_loop_break:wN \q_recursion_stop }
    \tl_if_in:NnTF \l__siunitx_number_input_digit_tl {#5}
      {
        \bool_lazy_or:nnTF
          {#3} { ! \str_if_eq_p:nn {#5} { 0 } }
          {
            \tl_put_right:Nn \l__siunitx_number_partial_tl {#5}
            \__siunitx_number_parse_loop_uncert:NNNNN
              #1 \c_false_bool \c_true_bool #4
          }
          {
            \__siunitx_number_parse_loop_uncert:NNNNN
              #1 \c_false_bool \c_false_bool #4
          }
      }
      {
        \tl_if_in:NnTF \l__siunitx_number_input_uncert_close_tl {#5}
          {
            \bool_if:NTF #2
              { \__siunitx_number_parse_loop_break:wN }
              {
                \__siunitx_number_parse_loop_main_store:NNN #1
                  \c_false_bool \c_false_bool
                \__siunitx_number_parse_loop_after_uncert:NNN #1 #3
              }
          }
          { \__siunitx_number_parse_loop_break:wN }
      }
  }
```

After a bracketed uncertainty there are only a very small number of valid choices. The number can end, there can be a complex root token or there can be a sign. The latter is only allowed if the part being parsed at the moment was the first part of the number. The case where there is no root symbol but there should have been is cleared up after the loop code, so at this stage there is no check.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_after_uncert:NNN #1#2#3
  {
    \quark_if_recursion_tail_stop:N #3
    \tl_if_in:NnTF \l__siunitx_number_input_complex_tl {#3}
      { \__siunitx_number_parse_loop_main_complex:N #1 }
      {
        \bool_if:NTF #2
          {
            \tl_if_in:NnTF \l__siunitx_number_input_sign_tl {#3}
              {
                \tl_set:Nn \l__siunitx_number_flex_tl {#3}
                \__siunitx_number_parse_loop_first:NNN
                  \l__siunitx_number_flex_tl \c_false_bool
              }
              { \__siunitx_number_parse_loop_break:wN }
          }
          { \__siunitx_number_parse_loop_break:wN }
      }
  }
```

When the complex root symbol comes at the start of the number rather than at the end, the easiest approach is to shuffle it to the "normal" position. As the exponent has already been removed, this must be the last token of the input and any duplication will be picked up. The case where just a complex root token has to be covered: in that situation, there is an implicit `1` to store after which the loop stops.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_root_swap:NNwNN #1#2#3
  \q_recursion_tail \q_recursion_stop
  {
    \tl_if_blank:nTF {#3}
      {
        \tl_set:Nx \l__siunitx_number_imaginary_tl
          {
            \exp_not:V #1
            { 1 } { } { }
          }
        \tl_clear:N #1
      }
      {
        \use:x
          {
            \tl_clear:N \exp_not:N #1
            \tl_set:Nn \exp_not:N \l__siunitx_number_flex_tl { \exp_not:V #1 }
          }
        \__siunitx_number_parse_loop_first:NNN
          \l__siunitx_number_flex_tl \c_false_bool
          #3 #2 \q_recursion_tail \q_recursion_stop
      }
  }
```

Nothing is allowed after a complex root token: check and if there is kill the parsing.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_complex_cleanup:wN
  #1 \q_recursion_tail \q_recursion_stop
  {
    \tl_if_blank:nF {#1}
      { \__siunitx_number_parse_loop_break:wN \q_recursion_stop }
```

```
  }
```

Something is not right: remove all of the remaining tokens from the number and clear the storage areas as a signal for the next part of the code.

```
\cs_new_protected:Npn \__siunitx_number_parse_loop_break:wN
  #1 \q_recursion_stop
  {
    \tl_clear:N \l__siunitx_number_imaginary_tl
    \tl_clear:N \l__siunitx_number_flex_tl
    \tl_clear:N \l__siunitx_number_real_tl
  }
```

(*End definition for* `\__siunitx_number_parse_loop:` *and others.*)

`\__siunitx_number_parse_sign:`
`\__siunitx_number_parse_sign_aux:Nw`

The first token of a number after a comparator could be a sign. A quick check is made and if found stored; if there is no sign then the internal format requires that `+` is used. For the number to be valid it has to be more than just a sign, so the next part of the chain is only called if that is the case.

```
\cs_new_protected:Npn \__siunitx_number_parse_sign:
  {
    \exp_after:wN \__siunitx_number_parse_sign_aux:Nw
      \l__siunitx_number_arg_tl \q_stop
  }
\cs_new_protected:Npn \__siunitx_number_parse_sign_aux:Nw #1#2 \q_stop
  {
    \tl_if_in:NnTF \l__siunitx_number_input_sign_tl {#1}
      {
        \tl_set:Nn \l__siunitx_number_arg_tl {#2}
        \tl_set:Nn \l__siunitx_number_real_tl {#1}
      }
      { \tl_set:Nn \l__siunitx_number_real_tl { + } }
    \tl_if_empty:NTF \l__siunitx_number_arg_tl
      { \tl_clear:N \l__siunitx_number_real_tl }
      { \__siunitx_number_parse_exponent: }
  }
```

(*End definition for* `\__siunitx_number_parse_sign:` *and* `\__siunitx_number_parse_sign_aux:Nw`.)

## 1.4 Processing numbers

`\l__siunitx_number_round_half_up_bool`
`\l__siunitx_number_round_min_tl`
`\l__siunitx_number_round_mode_tl`
`\l__siunitx_number_round_precision_int`

```
\keys_define:nn { siunitx }
  {
    round-half .choice: ,
    round-half / even .code:n =
      { \bool_set_false:N \l__siunitx_number_round_half_up_bool } ,
    round-half / up .code:n =
      { \bool_set_true:N \l__siunitx_number_round_half_up_bool } ,
    round-minimum .tl_set:N =
      \l__siunitx_number_round_min_tl ,
    round-mode .choice: ,
    round-mode / figures .code:n =
      { \tl_set:Nn \__siunitx_number_round_mode_tl { figures } } ,
    round-mode / none .code:n =
```

```
      { \tl_set:Nn \__siunitx_number_round_mode_tl { none } } ,
    round-mode / places .code:n =
      { \tl_set:Nn \__siunitx_number_round_mode_tl { places } } ,
    round-mode / uncertainty .code:n =
      { \tl_set:Nn \__siunitx_number_round_mode_tl { uncertainty } } ,
    round-precision .int_set:N =
      \l__siunitx_number_round_precision_int ,
  }
\bool_new:N \l__siunitx_number_round_half_up_bool
\tl_new:N \__siunitx_number_round_mode_tl
```

(*End definition for* `\l__siunitx_number_round_half_up_bool` *and others.*)

`\__siunitx_number_round:`
`\__siunitx_number_round:N`
`\__siunitx_number_round_none:nNnnnNn`

```
\cs_new_protected:Npn \__siunitx_number_round:
  {
    \__siunitx_number_round:N \l__siunitx_number_real_tl
    \__siunitx_number_round:N \l__siunitx_number_imaginary_tl
  }
\cs_new_protected:Npn \__siunitx_number_round:N #1
  {
    \tl_if_empty:NF #1
      {
        \tl_set:Nx #1
          {
            \cs:w
              __siunitx_number_round_ \__siunitx_number_round_mode_tl :nNnnnNn
              \exp_after:wN
            \cs_end: #1
          }
      }
  }
\cs_new:Npn \__siunitx_number_round_none:nNnnnNn #1#2#3#4#5#6#7
  { \exp_not:n { {#1} #2 {#3} {#4} {#5} #6 {#7} } }
```

(*End definition for* `\__siunitx_number_round:` , `\__siunitx_number_round:N` , *and* `\__siunitx_number_round_none:nNnnnNn`.)

`\__siunitx_number_round_figures:nNnnnNn`

```
\cs_new:Npn \__siunitx_number_round_figures:nNnnnNn #1#2#3#4#5#6#7
  {
    \tl_if_blank:nTF {#5}
      {
        \int_compare:nNnTF \l__siunitx_number_round_precision_int > 0
          { }
          { { } + { 0 } { } { } + { 0 } }
      }
      { \exp_not:n { {#1} #2 {#3} {#4} {#5} #6 {#7} } }
  }
```

(*End definition for* `\__siunitx_number_round_figures:nNnnnNn`.)

`\__siunitx_number_round_places:nNnnnNn`
`\__siunitx_number_round_places_decimal:nNnnnNn`
`\__siunitx_number_round_places_integer:nNnnnNn`

The first step when rounding to a fixed number of places is to establish if this is in the decimal or integer parts. The two require different calculations for how many digits to drop from the input.

```
\cs_new:Npn \__siunitx_number_round_places:nNnnnNn #1#2#3#4#5#6#7
  {
    \tl_if_blank:nTF {#5}
      {
        \int_compare:nNnTF \l__siunitx_number_round_precision_int > 0
          { \__siunitx_number_round_places_decimal:nNnnNn }
          { \__siunitx_number_round_places_integer:nNnnNn }
          {#1} #2 {#3} {#4} #6 {#7}
      }
      { \exp_not:n { {#1} #2 {#3} {#4} {#5} #6 {#7} } }
  }
\cs_new:Npn \__siunitx_number_round_places_decimal:nNnnNn #1#2#3#4#5#6
  { }
\cs_new:Npn \__siunitx_number_round_places_integer:nNnnNn #1#2#3#4#5#6
  { }
```

(*End definition for* `\__siunitx_number_round_places:nNnnnNn`, `\__siunitx_number_round_places_decimal:nNnnNn`, *and* `\__siunitx_number_round_places_integer:nNnnNn`.)

`\__siunitx_number_round_uncertainty:nNnnnNn`

```
\cs_new:Npn \__siunitx_number_round_uncertainty:nNnnnNn #1#2#3#4#5#6#7
  {
    \tl_if_blank:nTF {#5}
      { \exp_not:n { {#1} #2 {#3} {#4} { } #6 {#7} } }
      { }
  }
```

(*End definition for* `\__siunitx_number_round_uncertainty:nNnnnNn`.)

## 1.5 Formatting parsed numbers

`\l__siunitx_number_bracket_negative_bool`
`\l__siunitx_number_bracket_close_tl`
`\l__siunitx_number_explicit_plus_bool`
`\l__siunitx_number_exponent_base_tl`
`\l__siunitx_number_exponent_product_tl`
`\l__siunitx_number_group_decimal_bool`
`\l__siunitx_number_group_integer_bool`
`\l__siunitx_number_group_minimum_int`
`\l__siunitx_number_group_separator_tl`
`\l__siunitx_number_negative_color_tl`
`\l__siunitx_number_bracket_open_tl`
`\l__siunitx_number_output_uncert_close_tl`
`\l__siunitx_number_output_complex_tl`
`\l__siunitx_number_output_decimal_tl`
`\l__siunitx_number_output_uncert_open_tl`
`\l__siunitx_number_uncert_separate_bool`
`\l__siunitx_number_tight_bool`
`\l__siunitx_number_unity_mantissa_bool`
`\l__siunitx_number_zero_exponent_bool`

Keys producing tokens in the output.

```
\keys_define:nn { siunitx }
  {
    bracket-negative .bool_set:N =
      \l__siunitx_number_bracket_negative_bool ,
    explicit-plus .bool_set:N =
      \l__siunitx_number_explicit_plus_bool ,
    exponent-base .tl_set:N =
      \l__siunitx_number_exponent_base_tl ,
    exponent-product .tl_set:N =
      \l__siunitx_number_exponent_product_tl ,
    group-digits .choice: ,
    group-digits / all .code:n =
      {
        \bool_set_true:N \l__siunitx_number_group_decimal_bool
        \bool_set_true:N \l__siunitx_number_group_integer_bool
      } ,
    group-digits / decimal .code:n =
      {
        \bool_set_true:N  \l__siunitx_number_group_decimal_bool
        \bool_set_false:N \l__siunitx_number_group_integer_bool
      } ,
    group-digits / integer .code:n =
```

```
      {
        \bool_set_false:N \l__siunitx_number_group_decimal_bool
        \bool_set_true:N  \l__siunitx_number_group_integer_bool
      } ,
    group-digits / none .code:n =
      {
        \bool_set_false:N \l__siunitx_number_group_decimal_bool
        \bool_set_false:N \l__siunitx_number_group_integer_bool
      } ,
    group-digits .default:n  = all ,
    group-minimum-digits .int_set:N  =
      \l__siunitx_number_group_minimum_int ,
    group-separator .tl_set:N =
      \l__siunitx_number_group_separator_tl ,
    negative-color .tl_set:N =
    \l__siunitx_number_negative_color_tl ,
    number-close-bracket .tl_set:N =
      \l__siunitx_number_bracket_close_tl ,
    number-open-bracket .tl_set:N =
      \l__siunitx_number_bracket_open_tl ,
    output-close-uncertainty .tl_set:N =
      \l__siunitx_number_output_uncert_close_tl ,
    output-complex-root .tl_set:N =
      \l__siunitx_number_output_complex_tl ,
    output-decimal-marker .tl_set:N =
      \l__siunitx_number_output_decimal_tl ,
    output-open-uncertainty .tl_set:N =
      \l__siunitx_number_output_uncert_open_tl ,
    separate-uncertainty .bool_set:N =
      \l__siunitx_number_uncert_separate_bool ,
    tight-spacing .bool_set:N =
      \l__siunitx_number_tight_bool ,
    unity-mantissa .bool_set:N =
      \l__siunitx_number_unity_mantissa_bool ,
    zero-exponent .bool_set:N =
      \l__siunitx_number_zero_exponent_bool ,
  }
\bool_new:N \l__siunitx_number_group_decimal_bool
\bool_new:N \l__siunitx_number_group_integer_bool
```

(*End definition for* `\l__siunitx_number_bracket_negative_bool` *and others.*)

`\__siunitx_number_format:`

```
\cs_new_protected:Npn \__siunitx_number_format:
  {
    \tl_set:Nx \l__siunitx_number_formatted_tl
      {
        \tl_if_empty:NTF \l__siunitx_number_real_tl
          {
            \tl_if_empty:NF \l__siunitx_number_imaginary_tl
              { \__siunitx_number_format:N \l__siunitx_number_imaginary_tl }
          }
          {
            \tl_if_empty:NTF \l__siunitx_number_imaginary_tl
```

```
              { \__siunitx_number_format:N \l__siunitx_number_real_tl }
              { ??? }
          }
      }
  }
```

(*End definition for* `\__siunitx_number_format:`.)

`\__siunitx_number_format:N`
`\__siunitx_number_format:nNnnnNn`
`\__siunitx_number_format_comparator:n`
`\__siunitx_number_format_sign:N`
`\__siunitx_number_format_sign_aux:N`
`\__siunitx_number_format_sign_color:w`
`\__siunitx_number_format_sign_brackets:w`
`\__siunitx_number_format_integer:nnn`
`\__siunitx_number_format_decimal:n`
`\__siunitx_number_format_decimal:f`
`\__siunitx_number_format_digits:nn`
`\__siunitx_number_format_integer_aux:n`
`\__siunitx_number_format_integer_aux_0:n`
`\__siunitx_number_format_integer_aux_1:n`
`\__siunitx_number_format_integer_aux_2:n`
`\__siunitx_number_format_decimal_aux:n`
`\__siunitx_number_format_decimal_loop:NNNN`
`\__siunitx_number_format_integer_first:nnNN`
`\__siunitx_number_format_integer_loop:NNNN`
`\__siunitx_number_format_uncertainty:nn`
`siunitx_number_format_uncertainty_unaligned:`
`\__siunitx_number_format_uncertainty_aux:nn`
`\__siunitx_number_format_uncertainty_aux:fn`
`\__siunitx_number_format_uncertainty:nnw`
`\__siunitx_number_format_uncertainty:fnw`
`\__siunitx_number_format_uncertainty:nw`
`\__siunitx_number_format_exponent:Nnn`
`\__siunitx_number_format_end:`

The approach to formatting a single number is to split into the constituent parts. All of the parts are assembled including inserting tabular alignment markers (which may be empty) for each separate unit.

```
\cs_new:Npn \__siunitx_number_format:N #1
  { \exp_after:wN \__siunitx_number_format:nNnnnNn #1 }
\cs_new:Npn \__siunitx_number_format:nNnnnNn #1#2#3#4#5#6#7
  {
    \__siunitx_number_format_comparator:n {#1}
    \__siunitx_number_format_sign:N #2
    \__siunitx_number_format_integer:nnn {#3} {#4} {#7}
    \__siunitx_number_format_decimal:n {#4}
    \__siunitx_number_format_uncertainty:nn {#5} {#4}
    \__siunitx_number_format_exponent:Nnn #6 {#7} { #3 . #4 }
    \__siunitx_number_format_end:
  }
```

To get the spacing correct this needs to be an ordinary math character.

```
\cs_new:Npn \__siunitx_number_format_comparator:n #1
  {
    \tl_if_blank:nF {#1}
      { \exp_not:n { \mathord {#1} } }
    \exp_not:V \l__siunitx_number_tab_tl
  }
```

Formatting signs has to deal with some additional formatting requirements for negative numbers. Both making such numbers a fixed color and bracketing them needs some rearrangement of the order of tokens, which is set up in the main formatting macro by the dedicated do-nothing end function.

```
\cs_new:Npn \__siunitx_number_format_sign:N #1
  {
    \str_if_eq:nnTF {#1} { + }
      {
        \bool_if:NT \l__siunitx_number_explicit_plus_bool
          { \__siunitx_number_format_sign_aux:N #1 }
      }
      {
        \str_if_eq:nnTF {#1} { - }
          {
            \tl_if_empty:NF \l__siunitx_number_negative_color_tl
              { \__siunitx_number_format_sign_color:w }
            \bool_if:NTF \l__siunitx_number_bracket_negative_bool
              { \__siunitx_number_format_sign_brackets:w }
              { \__siunitx_number_format_sign_aux:N #1 }
          }
          { \__siunitx_number_format_sign_aux:N #1 }
      }
  }
```

```
\cs_new:Npn \__siunitx_number_format_sign_aux:N #1
  {
    \bool_if:NTF \l__siunitx_number_tight_bool
      { \exp_not:n { \mathord {#1} } }
      { \exp_not:n {#1} }
  }
\cs_new:Npn
  \__siunitx_number_format_sign_color:w #1 \__siunitx_number_format_end:
  {
    \exp_not:N \textcolor { \exp_not:V \l__siunitx_number_negative_color_tl }
      {
        #1
        \__siunitx_number_format_end:
      }
  }
\cs_new:Npn
  \__siunitx_number_format_sign_brackets:w #1 \__siunitx_number_format_end:
  {
    \exp_not:V \l__siunitx_number_bracket_open_tl
    #1
    \exp_not:V \l__siunitx_number_bracket_close_tl
    \__siunitx_number_format_end:
  }
```

Digit formatting leads off with separate functions to allow for a few "up front" items before using a common set of tests for some common cases. The code then splits again as the two types of grouping need different strategies.

```
\cs_new:Npn \__siunitx_number_format_integer:nnn #1#2#3
  {
    \bool_lazy_all:nF
      {
        { \str_if_eq_p:nn {#1} { 1 } }
        { \tl_if_blank_p:n {#2} }
        { ! \str_if_eq_p:nn {#3} { 0 } }
        { ! \l__siunitx_number_unity_mantissa_bool }
      }
      { \__siunitx_number_format_digits:nn { integer } {#1} }
  }
\cs_new:Npn \__siunitx_number_format_decimal:n #1
  {
    \exp_not:V \l__siunitx_number_tab_tl
    \tl_if_blank:nF {#1}
      { \exp_not:V \l__siunitx_number_output_decimal_tl }
    \exp_not:V \l__siunitx_number_tab_tl
    \__siunitx_number_format_digits:nn { decimal } {#1}
  }
\cs_generate_variant:Nn \__siunitx_number_format_decimal:n { f }
\cs_new:Npn \__siunitx_number_format_digits:nn #1#2
  {
    \bool_if:cTF { l__siunitx_number_group_ #1 _ bool }
      {
        \int_compare:nNnTF
          { \tl_count:n {#2} } < \l__siunitx_number_group_minimum_int
          { \exp_not:n {#2} }
```

```
          { \use:c { __siunitx_number_format_ #1 _aux:n } {#2} }
      }
      { \exp_not:n {#2} }
  }
```

For integers, we need to know how many digits there are to allow for the correct insertion of separators. That is done using a two-part set up such that there is no separator on the first pass.

```
\cs_new:Npn \__siunitx_number_format_integer_aux:n #1
  {
    \use:c
      {
        __siunitx_number_format_integer_aux_
        \int_eval:n { \int_mod:nn { \tl_count:n {#1} } { 3 } }
        :n
      } {#1}
  }
\cs_new:cpn { __siunitx_number_format_integer_aux_0:n } #1
  { \__siunitx_number_format_integer_first:nnNN #1 \q_nil }
\cs_new:cpn { __siunitx_number_format_integer_aux_1:n } #1
  { \__siunitx_number_format_integer_first:nnNN { } { } #1 \q_nil }
\cs_new:cpn { __siunitx_number_format_integer_aux_2:n } #1
  { \__siunitx_number_format_integer_first:nnNN { } #1 \q_nil }
\cs_new:Npn \__siunitx_number_format_integer_first:nnNN #1#2#3#4
  {
    \exp_not:n {#1#2#3}
    \quark_if_nil:NF #4
      { \__siunitx_number_format_integer_loop:NNNN #4 }
  }
\cs_new:Npn \__siunitx_number_format_integer_loop:NNNN #1#2#3#4
  {
    \exp_not:V \l__siunitx_number_group_separator_tl
    \exp_not:n {#1#2#3}
    \quark_if_nil:NF #4
      { \__siunitx_number_format_integer_loop:NNNN #4 }
  }
```

For decimals, no need to do any counting, just loop using enough markers to find the end of the list. By passing the decimal marker, it is possible not to have to use a check on the content of the rest of the number. The `\use_none:n(n)` mop up the remaining `\q_nil` tokens.

```
\cs_new:Npn \__siunitx_number_format_decimal_aux:n #1
  {
    \__siunitx_number_format_decimal_loop:NNNN \c_empty_tl
      #1 \q_nil \q_nil \q_nil
  }
\cs_new:Npn \__siunitx_number_format_decimal_loop:NNNN #1#2#3#4
  {
    \quark_if_nil:NF #2
      {
        \exp_not:V #1
        \exp_not:n {#2}
        \quark_if_nil:NTF #3
          { \use_none:n }
```

```
      {
        \exp_not:n {#3}
        \quark_if_nil:NTF #4
          { \use_none:nn }
          {
            \exp_not:n {#4}
            \__siunitx_number_format_decimal_loop:NNNN
              \l__siunitx_number_group_separator_tl
          }
      }
  }
```

Uncertainties which are directly attached are easy to deal with. For those that are separated, the first step is to find if they are entirely contained within the decimal part, and to pad if they are. For the case where the boundary is crossed to the integer part, the correct number of digit tokens need to be removed from the start of the uncertainty and the split result sent to the appropriate auxiliaries.

```
\cs_new:Npn \__siunitx_number_format_uncertainty:nn #1#2
  {
    \tl_if_blank:nTF {#1}
      { \__siunitx_number_format_uncertainty_unaligned: }
      {
        \bool_if:NTF \l__siunitx_number_uncert_separate_bool
          {
            \exp_not:V \l__siunitx_number_tab_tl
            \__siunitx_number_format_sign_aux:N \pm
            \exp_not:V \l__siunitx_number_tab_tl
            \__siunitx_number_format_uncertainty_aux:fn
              { \int_eval:n { \tl_count:n {#1} - \tl_count:n {#2} } }
              {#1}
          }
          {
            \exp_not:V \l__siunitx_number_output_uncert_open_tl
            \exp_not:n {#1}
            \exp_not:V \l__siunitx_number_output_uncert_close_tl
            \__siunitx_number_format_uncertainty_unaligned:
          }
      }
  }
\cs_new:Npn \__siunitx_number_format_uncertainty_unaligned:
  {
    \exp_not:V \l__siunitx_number_tab_tl
    \exp_not:V \l__siunitx_number_tab_tl
    \exp_not:V \l__siunitx_number_tab_tl
    \exp_not:V \l__siunitx_number_tab_tl
  }
\cs_new:Npn \__siunitx_number_format_uncertainty_aux:nn #1#2
  {
    \int_compare:nNnTF {#1} > 0
      {
        \__siunitx_number_format_uncertainty_aux:fnw
          { \int_eval:n { #1 - 1 } }
          { }
```

```
          #2 \q_nil
      }
      {
        0
        \__siunitx_number_format_decimal:f
          {
            \prg_replicate:nn { \int_abs:n {#1} } { 0 }
            #2
          }
      }
  }
\cs_generate_variant:Nn \__siunitx_number_format_uncertainty_aux:nn { f }
\cs_new:Npn \__siunitx_number_format_uncertainty_aux:nnw #1#2#3
  {
    \quark_if_nil:NF #3
      {
        \int_compare:nNnTF {#1} = 0
          { \__siunitx_number_format_uncertainty_aux:nw {#2#3} }
          {
            \__siunitx_number_format_uncertainty_aux:fnw
              { \int_eval:n { #1 - 1 } }
              {#2#3}
          }
      }
  }
\cs_generate_variant:Nn \__siunitx_number_format_uncertainty_aux:nnw { f }
\cs_new:Npn \__siunitx_number_format_uncertainty_aux:nw #1#2 \q_nil
  {
    \__siunitx_number_format_digits:nn { integer } {#1}
    \__siunitx_number_format_decimal:n {#2}
  }
```

Setting the exponent part requires some information about the mantissa: was it there or not. This means that whilst only the sign and value for the exponent are typeset here, there is a need to also have access to the combined mantissa part (with a decimal marker). The rest of the work is about picking up the various options and getting the combinations right. For signs, the auxiliary from the main sign routine can be used, but not the main function: negative exponents don't have special handling.

```
\cs_new:Npn \__siunitx_number_format_exponent:Nnn #1#2#3
  {
    \exp_not:V \l__siunitx_number_tab_tl
    \bool_lazy_or:nnTF
      { \l__siunitx_number_zero_exponent_bool }
      { ! \str_if_eq_p:nn {#2} { 0 } }
      {
        \bool_lazy_and:nnTF
          { \str_if_eq_p:nn {#3} { 1. } }
          { ! \l__siunitx_number_unity_mantissa_bool }
          { \exp_not:V \l__siunitx_number_tab_tl }
          {
            \bool_if:NTF \l__siunitx_number_tight_bool
              {
                \exp_not:N \mathord
                  { \exp_not:V \l__siunitx_number_exponent_product_tl }
```

```
            }
            { \exp_not:V \l__siunitx_number_exponent_product_tl }
          \exp_not:V \l__siunitx_number_tab_tl
        }
      \exp_not:V \l__siunitx_number_exponent_base_tl
      ^
        {
          \bool_lazy_or:nnT
            { \l__siunitx_number_explicit_plus_bool }
            { ! \str_if_eq_p:nn {#1} { + } }
            { \__siunitx_number_format_sign_aux:N #1 }
          \__siunitx_number_format_digits:nn { integer } {#2}
        }
    }
    { \exp_not:V \l__siunitx_number_tab_tl }
}
```

A do-nothing marker used to allow shuffling of the output and so expandable operations for formatting.

```
\cs_new:Npn \__siunitx_number_format_end: { }
```

(*End definition for* `\__siunitx_number_format:N` *and others.*)

## 1.6 Miscellaneous tools

`\l__siunitx_number_valid_tl` The list of valid tokens.

```
\tl_new:N \l__siunitx_number_valid_tl
```

(*End definition for* `\l__siunitx_number_valid_tl`.)

`\siunitx_if_number:nTF` Test if an entire number is valid: this means parsing the number but not returning anything.

```
\prg_new_protected_conditional:Npnn \siunitx_if_number:n #1
  { T , F , TF }
  {
    \group_begin:
      \bool_set_true:N \l__siunitx_number_validate_bool
      \__siunitx_number_parse:n {#1}
      \bool_lazy_and:nnTF
        { \tl_if_empty_p:N \l__siunitx_number_real_tl }
        { \tl_if_empty_p:N \l__siunitx_number_imaginary_tl }
        {
          \group_end:
          \prg_return_false:
        }
        {
          \group_end:
          \prg_return_true:
        }
  }
```

(*End definition for* `\siunitx_if_number:nTF`. *This function is documented on page* 7.)

\siunitx_if_number_token:NTF A simple conditional to answer the question of whether a specific token is possibly valid in a number.

```
\prg_new_protected_conditional:Npnn \siunitx_if_number_token:N #1
  { T , F , TF }
  {
    \tl_set:Nx \l__siunitx_number_valid_tl
      {
        \exp_not:V \l__siunitx_number_input_uncert_close_tl
        \exp_not:V \l__siunitx_number_input_complex_tl
        \exp_not:V \l__siunitx_number_input_comparator_tl
        \exp_not:V \l__siunitx_number_input_decimal_tl
        \exp_not:V \l__siunitx_number_input_digit_tl
        \exp_not:V \l__siunitx_number_input_exponent_tl
        \exp_not:V \l__siunitx_number_input_ignore_tl
        \exp_not:V \l__siunitx_number_input_uncert_open_tl
        \exp_not:V \l__siunitx_number_input_sign_tl
        \exp_not:V \l__siunitx_number_input_uncert_sign_tl
      }
    \tl_if_in:VnTF \l__siunitx_number_valid_tl {#1}
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\siunitx_if_number_token:NTF`*. This function is documented on page* 7.)

## 1.7 Messages

```
\msg_new:nnnn { siunitx } { number / invalid-input }
  { Invalid~number~'#1'. }
  {
    The~input~'#1'~could~not~be~parsed~as~a~number~following~the~
    format~defined~in~module~documentation.
  }
```

## 1.8 Standard settings for module options

Some of these follow naturally from the point of definition (*e.g.* boolean variables are always `false` to begin with), but for clarity everything is set here.

```
\keys_set:nn { siunitx }
  {
    bracket-negative         = false                                ,
    evaluate-expression      = false                                ,
    explicit-plus            = false                                ,
    exponent-base            = 10                                   ,
    exponent-product         = \times                               ,
    expression               = #1                                   ,
    group-digits             = all                                  ,
    group-minimum-digits     = 4                                    ,
    group-separator          = \,                                   , % (
    input-close-uncertainty  = )                                    ,
    input-complex-roots      = ij                                   ,
    input-comparators        = { <=>\approx\ge\geq\gg\le\leq\ll\sim } ,
    input-decimal-markers    = { ., }                               ,
    input-digits             = 0123456789                           ,
```

```
    input-exponent-markers   = dDeE                          ,
    input-ignore             = \,                            ,
    input-open-uncertainty   = (                             , % )
    input-signs              = +-\mp\pm                      ,
    input-uncertainty-signs  = \pm                           ,
    negative-color           =                               , % (
    number-close-bracket     = )                             ,
    number-open-bracket      = (                             , % )
    output-close-uncertainty = )                             ,
    output-complex-root      = \mathrm { i }                 ,
    output-decimal-marker    = .                             ,
    output-open-uncertainty  = (                             , % )
    round-half               = up                            ,
    round-minimum            = 0                             ,
    round-mode               = none                          ,
    round-precision          = 2                             ,
    separate-uncertainty     = false                         ,
    tight-spacing            = false                         ,
    unity-mantissa           = false                         ,
    zero-exponent            = false
  }
```

⟨/package⟩

# Part IV
# siunitx-print – Printing material with font control

This submodule is focussed on providing controlled printing for numbers and units. Key to this is control of font: conventions for printing quantities mean that the exact nature of the output is important. At the same time, this module provides flexibility for the user in terms of which aspects of the font are responsive to the surrounding general text. Printing material may also take place in text or math mode.

The printing routines assume that normal LaTeX $2_\varepsilon$ font selection commands are available, in particular `\bfsries`, `\mathrm`, `\mathversion`, `\mdseries`, `\rmfamily` and `\upshape`. It also requires the standard LaTeX $2_\varepsilon$ kernel commands `\ensuremath`, `\textsubscript` and `\textsuperscript` for printing in text mode. The following packages are also required to provide the functionality detailed.

- color: support for color using `\textcolor`
- textcomp: `\textminus` and `\textpm` for printing in text mode
- amstext: the `\text` command for printing in text mode

`\siunitx_print:nn` `\siunitx_print:nn {⟨type⟩} {⟨material⟩}`
`\siunitx_print:nV`

Prints the ⟨*material*⟩ according the the prevailing settings for the submodule as applicable to the ⟨*type*⟩ of content: the latter should be either `number` or `unit`. The ⟨*material*⟩ should comprise normal LaTeX mark-up for numbers or units. In particular, units will typically use `\mathrm` to indicate material to be printed in the current upright roman font, and `^` and `_` will typically be used to indicate super- and subscripts, respectively. These elements will be correctly handled when printing for example using `\mathsf` in math mode, or using only text fonts.

`\l_siunitx_print_series_prop`

This properly list contains mappings from text mode font weights to math mode font versions. The standard settings here cover

- `m` Standard (mid) weight, mapping to math version `normal`.
- `bx` Bold exanded weight, mapping to math version `bold`.
- `lt` Light weight, mapping to math version `light`.
- `l` Light weight, mapping to math version `light`.

## 0.1 Key–value options

The options defined by this submodule are available within the l3keys `siunitx` tree.

color `color = ⟨color⟩`

Color to apply to printed output: the latter should be a named color defined for use with `\textcolor`. The standard setting is empty (no color).

mode `mode = ⟨choice⟩`

Selects which mode (math or text) the output is printed in: a choice from the options `match`, `math` or `text`. The option `match` matches the mode prevailing at the point `\siunitx_print:nn` is called. The `math` and `text` options choose the relevant TEX mode for printing. The standard setting is `math`.

number-color `number-color = ⟨color⟩`

Color to apply to numbers in output: the latter should be a named color defined for use with `\textcolor`. The standard setting is empty (no color).

number-mode `number-mode = ⟨choice⟩`

Selects which mode (math or text) the numbers are printed in: a choice from the options `match`, `math` or `text`. The option `match` matches the mode prevailing at the point `\siunitx_print:nn` is called. The `math` and `text` options choose the relevant TEX mode for printing. The standard setting is `math`.

propagate-math-font `propagate-math-font = true|false`

Switch to determine if the currently-active math font is applied within printed output. This is relevant only when `\siunitx_print:nn` is called from within math mode: in text mode there is not active math font. When not active, math mode material will be typeset using standard math mode fonts without any changes being made to the supplied argument. The standard setting is `false`.

reset-math-version `reset-math-version = true|false`

Switch to determine whether the active `\mathversion` is reset to `normal` when printing in math mode. Note that math version is typically used to select `\boldmath`, though it is also be used by *e.g.* sansmath. The standard setting is `true`.

reset-text-family `reset-text-family = true|false`

Switch to determine whether the active math family is reset to `\rmfamily` when printing in text mode. The standard setting is `true`.

reset-text-series `reset-text-series = true|false`

Switch to determine whether the active math series is reset to `\mdseries` when printing in text mode. The standard setting is `true`.

reset-text-shape `reset-text-shape = true|false`

Switch to determine whether the active math shape is reset to `\upshape` when printing in text mode. The standard setting is `true`.

text-family-to-math `text-family-to-math = true|false`

Switch to determine if the family of the current text font should be applied (where possible) to printing in math mode. The standard setting is `false`.

text-weight-to-math `text-weight-to-math = true|false`

Switch to determine if the weight of the current text font should be applied (where possible) to printing in math mode. This is achieved by setting the `\mathversion`, and so will override `reset-math-version`. The mappings between text and math weight are stored in `\l_siunitx_print_series_prop`. The standard setting is `false`.

unit-color `unit-color = ⟨color⟩`

Color to apply to units in output: the latter should be a named color defined for use with `\textcolor`. The standard setting is empty (no color).

unit-mode `unit-mode = ⟨choice⟩`

Selects which mode (math or text) units are printed in: a choice from the options `match`, `math` or `text`. The option `match` matches the mode prevailing at the point `\siunitx_print:nn` is called. The `math` and `text` options choose the relevant TEX mode for printing. The standard setting is `math`.

# 1 siunitx-print implementation

Start the DocStrip guards.

```
⟨*package⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention): only internal material in this *submodule* should be used directly.

```
⟨@@=siunitx_print⟩
```

## 1.1 Initial set up

The printing routines depend on amstext for text mode working.

```
\RequirePackage { amstext }
```

Color support is always required.

```
\RequirePackage { color }
```

For a sensible `\textminus` we load textcomp if fontspec is not in use.

```
\AtBeginDocument
  {
    \@ifpackageloaded { fontspec }
      { }
      { \RequirePackage { textcomp } }
  }
```

\l__siunitx_print_tmp_box Scratch space.

```
\box_new:N \l__siunitx_print_tmp_box
```

(*End definition for* `\l__siunitx_print_tmp_box`.)

\document
__siunitx_print_store_fam:n
\c__siunitx_print_mathrm_int
\c__siunitx_print_mathsf_int
\c__siunitx_print_mathtt_int

In order to test math fonts, we need information about the \fam used by the various options. This is run as a hook onto \document, rather than using \AtBeginDocument as it has to come after anything that fontspec does (nasty errors arise otherwise). As this is a true one-off, we avoid wasting a box.

```
\tl_put_right:Nn \document
  {
    \__siunitx_print_store_fam:n { rm }
    \__siunitx_print_store_fam:n { sf }
    \__siunitx_print_store_fam:n { tt }
    \ignorespaces
  }
\cs_new_protected:Npn \__siunitx_print_store_fam:n #1
  {
    \group_begin:
      \hbox_set:Nn \l__siunitx_print_tmp_box
        {
          \ensuremath
            {
              \use:c { math #1 }
                { \int_const:cn { c__siunitx_print_math #1 _int } { \fam } }
            }
        }
    \group_end:
  }
```

(*End definition for* \document *and others. This function is documented on page* ??.)

## 1.2 Printing routines

\l__siunitx_print_number_color_tl
\l__siunitx_print_number_mode_tl
\l__siunitx_print_unit_color_tl
\l__siunitx_print_unit_mode_tl
\l__siunitx_print_math_font_bool
\l__siunitx_print_math_version_bool
\l__siunitx_print_math_family_bool
\l__siunitx_print_math_weight_bool
\l__siunitx_print_text_family_tl
\l__siunitx_print_text_series_tl
\l__siunitx_print_text_shape_tl

Options which apply to the main formatting routine, and so are not tied to either symbolic or literal input.

```
\tl_new:N \l__siunitx_print_number_mode_tl
\tl_new:N \l__siunitx_print_unit_mode_tl
\keys_define:nn { siunitx }
  {
    color .meta:n =
      { number-color = #1 , unit-color = #1 } ,
    mode .meta:n =
      { number-mode = #1 , unit-mode = #1 } ,
    number-color .tl_set:N =
      \l__siunitx_print_number_color_tl ,
    number-mode .choices:nn =
      { match , math , text }
      {
        \tl_set_eq:NN
          \l__siunitx_print_number_mode_tl \l_keys_choice_tl
      } ,
    propagate-math-font .bool_set:N =
      \l__siunitx_print_math_font_bool ,
    reset-math-version .bool_set:N =
      \l__siunitx_print_math_version_bool ,
    reset-text-family .bool_set:N =
      \l__siunitx_print_text_family_bool ,
```

```
    reset-text-series .bool_set:N =
      \l__siunitx_print_text_series_bool ,
    reset-text-shape .bool_set:N =
      \l__siunitx_print_text_shape_bool ,
    text-family-to-math .bool_set:N =
      \l__siunitx_print_math_family_bool ,
    text-weight-to-math .bool_set:N =
      \l__siunitx_print_math_weight_bool ,
    unit-color .tl_set:N =
      \l__siunitx_print_unit_color_tl ,
    unit-mode .choices:nn =
      { match , math , text }
      {
        \tl_set_eq:NN
          \l__siunitx_print_unit_mode_tl \l_keys_choice_tl
      }
  }
```

(*End definition for* `\l__siunitx_print_number_color_tl` *and others.*)

`\siunitx_print:nn`
`\siunitx_print:nV`

The main printing function doesn't actually need to do very much: just set the color and select the correct sub-function.

```
\cs_new_protected:Npn \siunitx_print:nn #1#2
  {
    \tl_if_empty:cTF { l__siunitx_print_ #1 _color_tl }
      { \use:n }
      { \exp_args:Nv \textcolor { l__siunitx_print_ #1 _color_tl } }
        {
          \use:c
            {
              __siunitx_print_
              \tl_use:c { l__siunitx_print_ #1 _mode_tl } :n
            }
              {#2}
        }
  }
\cs_generate_variant:Nn \siunitx_print:nn { nV }
```

(*End definition for* `\siunitx_print:nn`. *This function is documented on page* 35.)

`\__siunitx_print_match:n`

When the *output* mode should match the input, a simple selection of route can be made.

```
\cs_new_protected:Npn \__siunitx_print_match:n #1
  {
    \mode_if_math:TF
      { \__siunitx_print_math:n {#1} }
      { \__siunitx_print_text:n {#1} }
  }
```

(*End definition for* `\__siunitx_print_match:n`.)

`\l_siunitx_print_series_prop`

Mapping data to relate text series to math version.

```
\prop_new:N \l_siunitx_print_series_prop
\prop_put:Nnn \l_siunitx_print_series_prop { m }  { normal }
\prop_put:Nnn \l_siunitx_print_series_prop { bx } { bold }
\prop_put:Nnn \l_siunitx_print_series_prop { l }  { light }
\prop_put:Nnn \l_siunitx_print_series_prop { lt } { light }
```

(*End definition for* `\l_siunitx_print_series_prop`. *This variable is documented on page 35.*)

`\__siunitx_print_math:n`
`\__siunitx_print_math_version:nn`
`\__siunitx_print_math_version:Vn`
`\__siunitx_print_math_auxi:n`
`\__siunitx_print_math_auxii:n`
`\__siunitx_print_math_auxiii:n`
`\__siunitx_print_math_auxiv:n`
`\__siunitx_print_math_auxv:n`
`\__siunitx_print_math_aux:Nn`
`\__siunitx_print_math_aux:cn`
`\__siunitx_print_math_sub:n`
`\__siunitx_print_math_super:n`
`\__siunitx_print_math_script:n`
`\__siunitx_print_math_text:n`

The first step in setting in math mode is to check on the math version. The starting point is the question of whether text series needs to propagate to math mode: if so, check on the mapping, otherwise check on the current math version.

```
\cs_new_protected:Npn \__siunitx_print_math:n #1
  {
    \bool_if:NTF \l__siunitx_print_math_weight_bool
      {
        \prop_get:NVNTF \l_siunitx_print_series_prop
          \f@series \l__siunitx_print_tmp_tl
          { \__siunitx_print_math_version:Vn \l__siunitx_print_tmp_tl {#1} }
          { \__siunitx_print_math_auxi:n {#1} }
      }
      { \__siunitx_print_math_auxi:n {#1} }
  }
\cs_new_protected:Npn \__siunitx_print_math_auxi:n #1
  {
    \bool_if:NTF \l__siunitx_print_math_version_bool
      { \__siunitx_print_math_version:nn { normal } {#1} }
      { \__siunitx_print_math_auxii:n {#1} }
  }
```

Any setting which changes the math version can only be set from text mode (as it applies at the level of a formula). As such, the first test is to see if that needs to be to check if the math version has to be set: if so, switch to text mode, sort it out and switch back. That of course means that in such cases, line breaking will not be possible.

```
\cs_new_protected:Npn \__siunitx_print_math_version:nn #1#2
  {
    \str_if_eq_x:nnTF { \math@version } { #1 }
      { \__siunitx_print_math_auxii:n {#2} }
      {
        \mode_if_math:TF
          { \text }
          { \use:n }
            {
              \mathversion {#1}
              \__siunitx_print_math_auxii:n {#2}
            }
       }
  }
\cs_generate_variant:Nn \__siunitx_print_math_version:nn { V }
```

At this point, force math mode then start dealing with setting math font based on text family. If the text family is roman, life is slightly different to if it is sanserif or monospaced. In all cases, the outcomes can be handled using the same routines as for normal math mode treatment.

```
\cs_new_protected:Npn \__siunitx_print_math_auxii:n #1
  { \ensuremath { \__siunitx_print_math_auxiii:n {#1} } }
\cs_new_protected:Npn \__siunitx_print_math_auxiii:n #1
  {
    \bool_if:NTF \l__siunitx_print_math_family_bool
      {
        \str_case_x:nnF { \f@family }
```

```
          {
            { \rmdefault } { \__siunitx_print_math_auxv:n }
            { \sfdefault } { \__siunitx_print_math_aux:Nn \mathsf }
            { \ttdefault } { \__siunitx_print_math_aux:Nn \mathtt }
          }
          { \__siunitx_print_math_auxiv:n }
      }
      { \__siunitx_print_math_auxiv:n }
        {#1}
  }
```

Now we deal with the font selection in math mode. There are two possible cases. First, we are retaining the current math font, and the active one is `\mathsf` or `\mathtt`: that needs to be applied to the argument. Alternatively, if the current font is not retained, ensure that normal math mode rules are active. The parts here are split up to allow reuse when picking up the text family.

```
\cs_new_protected:Npn \__siunitx_print_math_auxiv:n #1
  {
    \bool_if:NTF \l__siunitx_print_math_font_bool
      {
        \int_case:nnF \fam
          {
            \c__siunitx_print_mathsf_int { \__siunitx_print_math_aux:Nn \mathsf }
            \c__siunitx_print_mathtt_int { \__siunitx_print_math_aux:Nn \mathtt }
          }
          { \use:n }
      }
      { \__siunitx_print_math_auxv:n }
        {#1}
  }
\cs_new_protected:Npn \__siunitx_print_math_auxv:n #1
  {
    \bool_lazy_or:nnTF
      { \int_compare_p:nNn \fam = { -1 } }
      { \int_compare_p:nNn \fam = \c__siunitx_print_mathrm_int }
      { \use:n }
      { \mathrm }
        {#1}
  }
```

Search-and-replace fun: deal with any `\mathrm` in the argument and also inside sub/superscripts.

```
\cs_new_protected:Npx \__siunitx_print_math_aux:Nn #1#2
  {
    \group_begin:
      \tl_set:Nn \exp_not:N \l__siunitx_print_tmp_tl {#2}
      \tl_replace_all:Nnn \exp_not:N \l__siunitx_print_tmp_tl
        { \exp_not:N \mathrm } { \exp_not:N \use:n }
      \tl_replace_all:Nnn \exp_not:N \l__siunitx_print_tmp_tl
        { \char_generate:nn { `\_ } { 8 } }
        { \exp_not:N \__siunitx_print_math_sub:n }
      \tl_replace_all:Nnn \exp_not:N \l__siunitx_print_tmp_tl
        { ^ }
        { \exp_not:N \__siunitx_print_math_super:n }
```

```
      #1 { \exp_not:N \tl_use:N \exp_not:N \l__siunitx_print_tmp_tl }
    \group_end:
  }
\cs_generate_variant:Nn \__siunitx_print_math_aux:Nn { c }
\cs_new_protected:Npx \__siunitx_print_math_sub:n #1
  {
    \char_generate:nn { `\_ } { 8 }
      { \exp_not:N \__siunitx_print_math_script:n {#1} }
  }
\cs_new_protected:Npn \__siunitx_print_math_super:n #1
  { ^ { \__siunitx_print_math_script:n {#1} } }
\cs_new_protected:Npn \__siunitx_print_math_script:n #1
  {
    \group_begin:
      \tl_set:Nn \l__siunitx_print_tmp_tl {#1}
      \tl_replace_all:Nnn \l__siunitx_print_tmp_tl { \mathrm } { \use:n }
      \tl_use:N \l__siunitx_print_tmp_tl
    \group_end:
  }
```

To match the text mode font, there is a simple look-up of the current font family. Luckily, mappings to math mode equivalents are easy.

(*End definition for* `\__siunitx_print_math:n` *and others.*)

`\__siunitx_print_text:n`
`\__siunitx_print_text_replace:n`
`\__siunitx_print_text_replace:N`
`\__siunitx_print_text_replace:NNn`
`\__siunitx_print_text_sub:n`
`\__siunitx_print_text_super:n`
`\__siunitx_print_text_scripts:NnN`
`\__siunitx_print_text_scripts:`
`\__siunitx_print_text_scripts_one:NnN`
`\__siunitx_print_text_scripts_two:NnNn`
`\__siunitx_print_text_scripts_two:nn`
`\__siunitx_print_text_scripts_two:n`

Typesetting in text mode is easy in font control terms but more tricky in the manipulation of the input. The easy part comes first.

```
\cs_new_protected:Npn \__siunitx_print_text:n #1
  {
    \text
      {
        \bool_if:NT \l__siunitx_print_text_family_bool
          { \rmfamily }
        \bool_if:NT \l__siunitx_print_text_series_bool
          { \mdseries }
        \bool_if:NT \l__siunitx_print_text_shape_bool
          { \upshape }
        \__siunitx_print_text_replace:n {#1}
      }
  }
```

To get math mode material to print in text mode, various search-and-replace steps are needed. The `\mathrm` command is also disabled as it is invalid in text mode.

```
\cs_new_protected:Npn \__siunitx_print_text_replace:n #1
  {
    \group_begin:
      \tl_set:Nn \l__siunitx_print_tmp_tl {#1}
      \__siunitx_print_text_replace:N \l__siunitx_print_tmp_tl
      \tl_use:N \l__siunitx_print_tmp_tl
    \group_end:
  }
\cs_new_protected:Npx \__siunitx_print_text_replace:N #1
  {
    \exp_not:N \__siunitx_print_text_replace:NNn #1
```

```
      \exp_not:N \mathrm
        { \exp_not:N \use:n }
      \exp_not:N \pm
        { \exp_not:N \textpm }
      \exp_not:N \mp
        { \exp_not:n { \ensuremath { \mp } } }
      -
        { \exp_not:N \textminus }
      \char_generate:nn { `\_ } { 8 }
        { \exp_not:N \__siunitx_print_text_sub:n }
      ^
        { \exp_not:N \__siunitx_print_text_super:n }
      \exp_not:N \q_recursion_tail
        { ? }
      \exp_not:N \q_recursion_stop
  }
\cs_new_protected:Npn \__siunitx_print_text_replace:NNn #1#2#3
  {
    \quark_if_recursion_tail_stop:N #2
    \tl_replace_all:Nnn #1 {#2} {#3}
    \__siunitx_print_text_replace:NNn #1
  }
```

Sub- and superscripts can be in any order in the source. The first step of handling them is therefore to do a look-ahead to work out whether only one or both are present.

```
\cs_new_protected:Npn \__siunitx_print_text_sub:n #1
  {
    \__siunitx_print_text_scripts:NnN
      \textsubscript {#1} \__siunitx_print_text_super:n
  }
\cs_new_protected:Npn \__siunitx_print_text_super:n #1
  {
    \__siunitx_print_text_scripts:NnN
      \textsuperscript {#1} \__siunitx_print_text_sub:n
  }
\cs_new_protected:Npn \__siunitx_print_text_scripts:NnN #1#2#3
  {
    \cs_set_protected:Npn \__siunitx_print_text_scripts:
      {
        \if_meaning:w \l_peek_token #3
          \exp_after:wN \__siunitx_print_text_scripts_two:NnNn
        \else:
          \exp_after:wN \__siunitx_print_text_scripts_one:Nn
        \fi:
          #1 {#2}
      }
    \peek_after:Nw \__siunitx_print_text_scripts:
  }
\cs_new_protected:Npn \__siunitx_print_text_scripts: { }
```

In the simple case of one script item, we have to do a search-and-replace to deal with anything inside the argument.

```
\cs_new_protected:Npn \__siunitx_print_text_scripts_one:Nn #1#2
  {
    \group_begin:
```

```
      \tl_set:Nn \l__siunitx_print_tmp_tl {#2}
      \__siunitx_print_text_replace:N \l__siunitx_print_tmp_tl
    \exp_args:NNV \group_end:
    #1 \l__siunitx_print_tmp_tl
  }
```

For the two scripts case, we cannot use `\textsubscript`/`\textsuperscript` as they don't stack directly. Instead, we sort out the ordering then use an implementation for both parts that is the same as the kernel text scripts.

```
\cs_new_protected:Npn \__siunitx_print_text_scripts_two:NnNn #1#2#3#4
  {
    \cs_if_eq:NNTF #1 \textsubscript
      { \__siunitx_print_text_scripts_two:nn {#4} {#2} }
      { \__siunitx_print_text_scripts_two:nn {#2} {#4} }
  }
\cs_new_protected:Npx \__siunitx_print_text_scripts_two:nn #1#2
  {
    \group_begin:
      \exp_not:N \m@th
      \exp_not:N \ensuremath
        {
          ^ { \exp_not:N \__siunitx_print_text_scripts_two:n {#1} }
          \char_generate:nn { `\_ } { 8 }
            { \exp_not:N \__siunitx_print_text_scripts_two:n {#2} }
        }
    \group_end:
  }
\cs_new_protected:Npn \__siunitx_print_text_scripts_two:n #1
  {
    \mbox
      {
        \fontsize \sf@size \z@ \selectfont
        \__siunitx_print_text_scripts_one:Nn \use:n {#1}
      }
  }
```

(*End definition for* `\__siunitx_print_text:n` *and others.*)

## 1.3 Standard settings for module options

Some of these follow naturally from the point of definition (*e.g.* boolean variables are always `false` to begin with), but for clarity everything is set here.

```
\keys_set:nn { siunitx }
  {
    color               =       ,
    mode                = math  ,
    number-color        =       ,
    number-mode         = math  ,
    propagate-math-font = false ,
    reset-math-version  = true  ,
    reset-text-shape    = true  ,
    reset-text-series   = true  ,
    reset-text-family   = true  ,
    text-family-to-math = false ,
```

```
    text-weight-to-math = false ,
    unit-color          =       ,
    unit-mode           = math
  }
⟨/package⟩
```

# Part V
# siunitx-table – Formatting numbers in tables

## 1 siunitx-table implementation

Start the DocStrip guards.

```
1 ⟨*package⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention): only internal material in this *submodule* should be used directly.

```
2 ⟨@@=siunitx_table⟩
```

`\l__siunitx_table_tmp_tl` Scratch space.

```
3 \tl_new:N \l__siunitx_table_tmp_tl
```

(*End definition for* `\l__siunitx_table_tmp_tl`.)

### 1.1 Interface functions

`\l__siunitx_table_parse_bool` One top-level setting applies to all number cells.

```
4 \keys_define:nn { siunitx / table }
5   {
6     parse .bool_set:N = \l__siunitx_table_parse_bool
7   }
```

(*End definition for* `\l__siunitx_table_parse_bool`.)

`\siunitx_cell_begin:`
`\siunitx_cell_end:`

```
8 \cs_new_protected:Npn \siunitx_cell_begin:
9   {
10    \bool_if:NTF \l__siunitx_table_parse_bool
11      { \__siunitx_table_collect_begin:N }
12      { \__siunitx_table_direct_begin:N }
13  }
14 \cs_new_protected:Npn \siunitx_cell_end:
15  {
16    \__siunitx_table_collect_end:
17  }
```

(*End definition for* `\siunitx_cell_begin:` *and* `\siunitx_cell_end:`. *These functions are documented on page* ??.)

## 1.2 Collecting tokens

\l__siunitx_table_collect_tl

Space for tokens.

```
\tl_new:N \l__siunitx_table_collect_tl
```

(*End definition for* `\l__siunitx_table_collect_tl`.)

__siunitx_table_collect_begin:N

Collecting a tabular cell means doing a token-by-token collection. In previous versions of siunitx that was done along with picking out the numerical part, but the code flow ends up very tricky. Here, therefore, we just collect up the unchanged tokens first. The `#1` collects the leading `\ignorespaces` from the template: that has to go to avoid issues with the peek-ahead code. The definition of `\cr` is used to allow collection of any tokens inserted after the main content when dealing with the last cell of a row: the "group" around it is needed to avoid issues with the underlying `\halign`. (The approach is based on that in collcell.) Notice that as each cell forms a group there is no need to reset the definition of `\cr`.

```
\cs_new_protected:Npn \__siunitx_table_collect_begin:N #1
  {
    \tl_clear:N \l__siunitx_table_collect_tl
    \if_false: { \fi:
    \cs_set_protected:Npn \cr
      {
        \__siunitx_table_collect_loop:
        \tex_cr:D
      }
    \if_false: } \fi:
    \__siunitx_table_collect_loop:
  }
```

(*End definition for* `\__siunitx_table_collect_begin:N`.)

__siunitx_table_collect_loop:
__siunitx_table_collect_group:n
__siunitx_table_collect_token:N
__siunitx_table_collect_search:NnF
__siunitx_table_collect_search_aux:NNn

Collecting up the cell content needs a loop: this is done using a `peek` approach as it's most natural. (A slower approach is possible using something like the `\tl_lower_case:n` loop code.) The set of possible tokens is somewhat limited compared to an arbitrary cell (*cf.* the approach in collcell): the special cases are pulled out for manual handling. The flexible lookup approach is more-or-less the same idea as in the kernel `case` functions. The `\relax` special case covers the case where `\\` has been expanded in an empty cell.

```
\cs_new_protected:Npn \__siunitx_table_collect_loop:
  {
    \peek_catcode_ignore_spaces:NTF \c_group_begin_token
      { \__siunitx_table_collect_group:n }
      { \__siunitx_table_collect_token:N }
  }
\cs_new_protected:Npn \__siunitx_table_collect_group:n #1
  {
    \tl_put_right:Nn \l__siunitx_table_collect_tl { {#1} }
    \__siunitx_table_collect_loop:
  }
\cs_new_protected:Npn \__siunitx_table_collect_token:N #1
  {
    \__siunitx_table_collect_search:NnF #1
      {
        \unskip            { \__siunitx_table_collect_loop: }
```

```
        \end               { \tabularnewline \end }
        \relax             { \relax }
        \tabularnewline    { \tabularnewline }
        \siunitx_cell_end: { \siunitx_cell_end: }
      }
      {
        \tl_put_right:Nn \l__siunitx_table_collect_tl {#1}
        \__siunitx_table_collect_loop:
      }
  }
\AtBeginDocument
  {
    \@ifpackageloaded { mdwtab }
      {
        \cs_set_protected:Npn \__siunitx_table_collect_token:N #1
          {
            \__siunitx_table_collect_search:NnF #1
              {
                \@maybe@unskip     { \__siunitx_table_collect_loop: }
                \tab@setcr         { \__siunitx_table_collect_loop: }
                \unskip            { \__siunitx_table_collect_loop: }
                \end               { \tabularnewline \end }
                \relax             { \relax }
                \tabularnewline    { \tabularnewline }
                \siunitx_cell_end: { \siunitx_cell_end: }
              }
              {
                \tl_put_right:Nn \l__siunitx_table_collect_tl {#1}
                \__siunitx_table_collect_loop:
              }
          }
      }
      { }
  }
\cs_new_protected:Npn \__siunitx_table_collect_search:NnF #1#2#3
  {
    \__siunitx_table_collect_search_aux:NNn #1
      #2
      #1 {#3}
    \q_stop
  }
\cs_new_protected:Npn \__siunitx_table_collect_search_aux:NNn #1#2#3
  {
    \token_if_eq_meaning:NNTF #1 #2
      { \use_i_delimit_by_q_stop:nw {#3} }
      { \__siunitx_table_collect_search_aux:NNn #1 }
  }
```

(*End definition for* `\__siunitx_table_collect_loop:` *and others.*)

## 1.3 Separating collected material

The input needs to be divided into numerical tokens and those which appear before and after them. This needs a second loop and validation.

\l__siunitx_table_pre_tl
\l__siunitx_table_number_tl
\l__siunitx_table_post_tl

Space for tokens.

```
\tl_new:N \l__siunitx_table_pre_tl
\tl_new:N \l__siunitx_table_number_tl
\tl_new:N \l__siunitx_table_post_tl
```

(*End definition for* `\l__siunitx_table_pre_tl`, `\l__siunitx_table_number_tl`, *and* `\l__siunitx_table_post_tl`.)

__siunitx_table_collect_end:

At the end of the cell, expand all of the content as far as possible then split it up into numerical and non-numerical parts.

```
\cs_new_protected:Npn \__siunitx_table_collect_end:
  {
    \protected@edef \l__siunitx_table_collect_tl
      { \l__siunitx_table_collect_tl }
    \tl_clear:N \l__siunitx_table_pre_tl
    \tl_clear:N \l__siunitx_table_number_tl
    \tl_clear:N \l__siunitx_table_post_tl
    \exp_after:wN \__siunitx_table_split_loop: \l__siunitx_table_collect_tl
      \q_recursion_tail \q_recursion_stop
    \__siunitx_table_split_tidy:N \l__siunitx_table_pre_tl
    \__siunitx_table_split_tidy:N \l__siunitx_table_post_tl
    \tl_if_empty:NTF \l__siunitx_table_number_tl
      { \__siunitx_table_print_text:V \l__siunitx_table_pre_tl }
      {
        \__siunitx_table_print:VVV
          \l__siunitx_table_pre_tl
          \l__siunitx_table_number_tl
          \l__siunitx_table_post_tl
      }
  }
```

(*End definition for* `\__siunitx_table_collect_end:`.)

__siunitx_table_split_loop:
__siunitx_table_split_group:n
__siunitx_table_split_token:N

Splitting into parts uses the fact that numbers cannot contain groups and that we can track where we are up to based on the content of the token lists.

```
\cs_new_protected:Npn \__siunitx_table_split_loop:
  {
    \peek_catcode_ignore_spaces:NTF \c_group_begin_token
      { \__siunitx_table_split_group:n }
      { \__siunitx_table_split_token:N }
  }
\cs_new_protected:Npn \__siunitx_table_split_group:n #1
  {
    \tl_if_empty:NTF \l__siunitx_table_number_tl
      { \tl_put_right:Nn \l__siunitx_table_pre_tl { {#1} } }
      { \tl_put_right:Nn \l__siunitx_table_post_tl { {#1} } }
    \__siunitx_table_split_loop:
  }
\cs_new_protected:Npn \__siunitx_table_split_token:N #1
  {
    \quark_if_recursion_tail_stop:N #1
    \tl_if_empty:NTF \l__siunitx_table_post_tl
      {
        \siunitx_if_number_token:NTF #1
```

```
          { \tl_put_right:Nn \l__siunitx_table_number_tl {#1} }
          {
            \tl_if_empty:NTF \l__siunitx_table_number_tl
              { \tl_put_right \l__siunitx_table_pre_tl {#1} }
              { \tl_put_right \l__siunitx_table_post_tl {#1} }
          }
      }
      { \tl_put_right:Nn \l__siunitx_table_post_tl {#1} }
    \__siunitx_table_split_loop:
  }
```

(*End definition for* `\__siunitx_table_split_loop:` , `\__siunitx_table_split_group:n` , *and* `\__siunitx_table_split_token:N`.)

`\__siunitx_table_split_tidy:N` `\__siunitx_table_split_tidy:Nn` `\__siunitx_table_split_tidy:NV`

A quick test for the entire content being surrounded by a set of braces: rather than look explicitly, use the fact that a string comparison can detect the same thing. The auxiliary is needed to avoid having to go *via* a `:D` function (for the expansion behaviour).

```
\cs_new_protected:Npn \__siunitx_table_split_tidy:N #1
  {
    \tl_if_empty:NF #1
      { \__siunitx_table_split_tidy:NV #1 #1 }
  }
\cs_new_protected:Npn \__siunitx_table_split_tidy:Nn #1#2
  {
    \str_if_eq_x:nnT
      { \exp_not:n {#2} }
      { { \exp_not:o { \use:n #2 } } }
      { \tl_set:No #1 { \use:n #2 } }
  }
\cs_generate_variant:Nn \__siunitx_table_split_tidy:Nn { NV }
```

(*End definition for* `\__siunitx_table_split_tidy:N` *and* `\__siunitx_table_split_tidy:Nn`.)

## 1.4 Printing numbers in cells: spacing

Getting the general alignment correct in tables is made more complex than one would like by the colortbl package. In the original LaTeX $2_\varepsilon$ definition, cell material is centred by a construction of the (primitive) form

```
\hfil
#
\hfil
```

which only uses `fil` stretch. That is altered by colortbl to broadly

```
\hskip 0pt plus 0.5fill
\kern 0pt
#
\hskip 0pt plus 0.5fill
```

which means there is `fill` stretch to worry about and the kern as well.

__siunitx_table_skip:n

To prevent combination of skips, a kern is inserted after each one. This is best handled as a short auxiliary.

```
\cs_new_protected:Npn \__siunitx_table_skip:n #1
  {
    \skip_horizontal:n {#1}
    \tex_kern:D \c_zero_skip
  }
```

(*End definition for* `\__siunitx_table_skip:n`.)

\l__siunitx_table_column_width_dim
\l__siunitx_table_fixed_width_bool

Settings which apply to aligned columns in general.

```
\keys_define:nn { siunitx / table }
  {
    column-width .dim_set:N =
      \l__siunitx_table_column_width_dim ,
    fixed-width .bool_set:N =
      \l__siunitx_table_fixed_width_bool
  }
```

(*End definition for* `\l__siunitx_table_column_width_dim` *and* `\l__siunitx_table_fixed_width_bool`.)

__siunitx_table_align_center:n
__siunitx_table_align_left:n
__siunitx_table_align_right:n
__siunitx_table_align_auxi:nn
__siunitx_table_align_auxii:nn

The beginning and end of each table cell have to adjust the position of the content using glue. When colortbl is loaded the glue is done in two parts: one for our positioning and one to explicitly override that from the package. Using a two-step auxiliary chain avoids needing to repeat any code and the impact of the extra expansion should be trivial.

```
\cs_new_protected:Npn \__siunitx_table_align_center:n #1
  { \__siunitx_table_align_auxi:nn {#1} { 0pt plus 0.5fill } {#1} }
\cs_new_protected:Npn \__siunitx_table_align_left:n #1
  { \__siunitx_table_align_auxi:nn {#1} { 0pt } {#1} }
\cs_new_protected:Npn \__siunitx_table_align_right:n #1
  { \__siunitx_table_align_auxi:nn {#1} { 0pt plus 1fill } {#1} }
\cs_new_protected:Npn \__siunitx_table_align_auxi:nn #1#2
  {
    \bool_if:NTF \l__siunitx_table_fixed_width_bool
      { \hbox_to_wd:nn \l__siunitx_table_column_width_dim }
      { \use:n }
      {
        \__siunitx_table_skip:n {#2}
        #1
        \__siunitx_table_skip:n { 0pt plus 1fill - #2 }
      }
  }
\AtBeginDocument
  {
    \@ifpackageloaded { colortbl }
      {
        \cs_new_eq:NN
          \__siunitx_table_align_auxii:nn
          \__siunitx_table_align_auxi:nn
        \cs_set_protected:Npn \__siunitx_table_align_auxi:nn #1#2
          {
            \__siunitx_table_skip:n{ 0pt plus -0.5fill }
            \__siunitx_table_align_auxii:nn {#1} {#2}
            \__siunitx_table_skip:n { 0pt plus -0.5fill }
```

```
            }
        }
        { }
    }
```

(*End definition for* `\__siunitx_table_align_center:n` *and others.*)

## 1.5 Printing just text

In cases where there is no numerical part, siunitx allows alignment of the "escaped" text independent of the underlying column type.

`\l__siunitx_table_align_text_tl` Alignment is handled using a `tl` as this allows a fast lookup at the point of use.

```
\keys_define:nn { siunitx / table }
  {
    text-alignment .choices:nn =
      { center , left , right }
      { \tl_set:Nn \l__siunitx_table_align_text_tl {#1} } ,
  }
\tl_new:N \l__siunitx_table_align_text_tl
```

(*End definition for* `\l__siunitx_table_align_text_tl`.)

`\__siunitx_table_print_text:n` `\__siunitx_table_print_text:V` Printing escaped text is easy: just place it in correctly in the column.

```
\cs_new_protected:Npn \__siunitx_table_print_text:n #1
  {
    \use:c { __siunitx_table_align_ \l__siunitx_table_align_text_tl :n } {#1}
  }
\cs_generate_variant:Nn \__siunitx_table_print_text:n { V }
```

(*End definition for* `\__siunitx_table_print_text:n`.)

## 1.6 Reserving space: the table format

`\l__siunitx_table_format_tl`

```
\keys_define:nn { siunitx / table }
  {
    format .tl_set:N = \l__siunitx_table_format_tl
  }
```

(*End definition for* `\l__siunitx_table_format_tl`.)

## 1.7 Directly printing without collection

Collecting the number allows for various effects but is not as fast as simply aligning on the first token that is a decimal marker. The strategy here is that used by dcolumn.

`\__siunitx_table_direct_begin:N` `\__siunitx_table_direct_begin_aux:` After removing the `\ignorespaces` at the start of the cell, check to see if there is a `{` and branch as appropriate.

```
\cs_new_protected:Npn \__siunitx_table_direct_begin: #1
  {
    \peek_catcode_ignore_spaces:NTF \c_group_begin_token
      { \__siunitx_table_print_text:n }
      {
```

```
        \m@th
        \__siunitx_table_direct_begin_aux:
      }
  }
\cs_new_protected:Npn \__siunitx_table_direct_begin_aux:
  {
  }
```

(*End definition for* `\__siunitx_table_direct_begin:N` *and* `\__siunitx_table_direct_begin_aux:`.)

## 1.8 Printing numbers in cells: main functions

`\l__siunitx_table_align_comparator_bool`
`\l__siunitx_table_align_exponent_bool`
`\l__siunitx_table_align_uncertainty_bool`
`\l__siunitx_table_alignment_tl`
`\l__siunitx_table_parse_only_bool`

Alignment is handled using a `tl` as this allows a fast lookup at the point of use.

```
\keys_define:nn { siunitx / table }
  {
    align-comparator .bool_set:N =
      \l__siunitx_table_align_comparator_bool ,
    align-exponent .bool_set:N =
      \l__siunitx_table_align_exponent_bool ,
    align-uncertainty .bool_set:N =
      \l__siunitx_table_align_uncertainty_bool ,
    alignment .choices:nn =
      { center , left , right }
      { \tl_set:Nn \l__siunitx_table_alignment_tl {#1} } ,
    parse-only .bool_set:N =
      \l__siunitx_table_parse_only_bool
  }
\tl_new:N \l__siunitx_table_alignment_tl
```

(*End definition for* `\l__siunitx_table_align_comparator_bool` *and others.*)

`\__siunitx_table_print:nnn`
`\__siunitx_table_print:VVV`
`\__siunitx_table_print_non_aligned:nnn`
`\__siunitx_table_print_aligned:nnn`

```
\cs_new_protected:Npn \__siunitx_table_print:nnn #1#2#3
  {
    \use:c { __siunitx_table_align_ \l__siunitx_table_alignment_tl :n }
      {
        \bool_if:NTF \l__siunitx_table_parse_only_bool
          { \__siunitx_table_print_non_aligned:nnn }
          { \__siunitx_table_print_aligned:nnn }
          {#1} {#2} {#3}
      }
  }
\cs_generate_variant:Nn \__siunitx_table_print:nnn { VVV }
\cs_new_protected:Npn \__siunitx_table_print_non_aligned:nnn #1#2#3
  {
    #1
    \siunitx_number_format:nN {#2} \l__siunitx_table_tmp_tl
    \siunitx_print:nV { number } \l__siunitx_table_tmp_tl
    #3
  }
\cs_new_protected:Npn \__siunitx_table_print_aligned:nnn #1#2#3
  {
  }
```

(*End definition for* `\__siunitx_table_print:nnn`, `\__siunitx_table_print_non_aligned:nnn`, *and* `\__siunitx_table_print_aligned:nnn`.)

## 1.9 Standard settings for module options

Some of these follow naturally from the point of definition (*e.g.* boolean variables are always `false` to begin with), but for clarity everything is set here.

```
\keys_set:nn { siunitx / table }
  {
    align-comparator  = false  ,
    align-exponent    = false  ,
    align-uncertainty = false  ,
    alignment         = center ,
    column-width      = 0pt    ,
    fixed-width       = false  ,
    format            =        ,
    parse             = true   ,
    parse-only        = false  ,
    text-alignment    = center
  }
```

⟨/package⟩

# Part VI
# siunitx-unit – Parsing and formatting units

This submodule is dedicated to formatting physical units. The main function, `\siunitx_unit_format:nN`, takes user input specify physical units and converts it into a formatted token list suitable for typesetting in math mode. While the formatter will deal correctly with "literal" user input, the key strength of the module is providing a method to describe physical units in a "symbolic" manner. The output format of these symbolic units can then be controlled by a number of key–value options made available by the module.

A small number of LaTeX $2_\varepsilon$ math mode commands are assumed to be available as part of the formatted output. The `\mathchoice` command (normally the TeX primitive) is needed when using `per-mode = symbol-or-fraction`. The command `\mathrm` is used for wrapping the text (letter) part of units. The commands `\frac`, `\mbox`, `\␣` and `\,` are used by the standard module settings, and `\ensuremath`, `\hbar`, `\mathit` and `\mathrm` in some standard unit definitions (for atomic and natural units). For the display of colored (highlighted) and cancelled units, the commands `\textcolor` and `\cancel` are assumed to be available.

## 1 Formatting units

`\siunitx_unit_format:nN` `\siunitx_unit_format:nN {⟨units⟩} ⟨tl var⟩`

This function converts the input ⟨*units*⟩ into a processed ⟨*tl var*⟩ which can then be inserted in math mode to typeset the material. Where the ⟨*units*⟩ are given in symbolic form, described elsewhere, this formatting process takes place in two stages: the ⟨*units*⟩ are parsed into a structured form before the generation of the appropriate output form based on the active settings. When the ⟨*units*⟩ are given as literals, processing is minimal: the characters `.` and `~` are converted to unit products (boundaries). In both cases, the result is a series of tokens intended to be typeset in math mode with appropriate choice of font for typesetting of the textual parts.

For example,

```
\siunitx_unit_format:nN { \kilo \metre \per \second } \l_tmpa_tl
```

will, with standard settings, result in `\l_tmpa_tl` being set to

```
\mathrm{km}\,\mathrm{s}^{-1}
```

\siunitx_unit_format:nNN `\siunitx_unit_format:nNN {⟨units⟩} ⟨tl var⟩ ⟨fp var⟩`

This function formats the ⟨*units*⟩ in the same way as described for `\siunitx_unit_format:nN`. When the input is given in symbolic form, any decimal unit prefixes will be extracted and the overall power of ten that these represent will be stored in the ⟨*fp var*⟩.

For example,

```
\siunitx_unit_format:nNN { \kilo \metre \per \second }
  \l_tmpa_tl \l_tmpa_fp
```

will, with standard settings, result in `\l_tmpa_tl` being set to

```
\mathrm{m}\,\mathrm{s}^{-1}
```

with `\l_tmpa_fp` taking value 3. Note that the latter is a floating point variable: it is possible for non-integer values to be obtained here.

# 2 Defining symbolic units

\siunitx_declare_prefix:Nnn `\siunitx_declare_prefix:Nnn ⟨prefix⟩ {⟨symbol⟩} {⟨power⟩}`

Defines a symbolic ⟨*prefix*⟩ (which should be a control sequence such as `\kilo`) to be converted by the parser to the ⟨*symbol*⟩. The latter should consist of literal content (*e.g.* `k`). In literal mode the ⟨*symbol*⟩ will be typeset directly. The prefix should represent an integer ⟨*power*⟩ of 10, and this information may be used to convert from one or more ⟨*prefix*⟩ symbols to an overall power applying to a unit. See also `\siunitx_declare_prefix:Nn`.

\siunitx_declare_prefix:Nn `\siunitx_declare_prefix:Nn ⟨prefix⟩ {⟨symbol⟩}`

Defines a symbolic ⟨*prefix*⟩ (which should be a control sequence such as `\kilo`) to be converted by the parser to the ⟨*symbol*⟩. The latter should consist of literal content (*e.g.* `k`). In literal mode the ⟨*symbol*⟩ will be typeset directly. In contrast to `\siunitx_declare_prefix:Nnn`, there is no assumption about the mathematical nature of the ⟨*prefix*⟩, *i.e.* the prefix may represent a power of any base. As a result, no conversion of the ⟨*prefix*⟩ to a numerical power will be possible.

\siunitx_declare_power:NNn `\siunitx_declare_power:NnN ⟨pre-power⟩ ⟨post-power⟩ {⟨value⟩}`

Defines *two* symbolic ⟨*powers*⟩ (which should be control sequences such as `\squared`) to be converted by the parser to the ⟨*value*⟩. The latter should be an integer or floating point number in the format defined for l3fp. Powers may precede a unit or be give after it: both forms are declared at once, as indicated by the argument naming. In literal mode, the ⟨*value*⟩ will be applied as a superscript to either the next token in the input (for the ⟨*pre-power*⟩) or appended to the previously-typeset material (for the ⟨*post-power*⟩).

\siunitx_declare_qualifier:Nn `\siunitx_declare_qualifier:Nn ⟨qualifier⟩ {⟨meaning⟩}`

Defines a symbolic ⟨*qualifier*⟩ (which should be a control sequence such as `\catalyst`) to be converted by the parser to the ⟨*meaning*⟩. The latter should consist of literal content (*e.g.* `cat`). In literal mode the ⟨*meaning*⟩ will be typeset following a space after the unit to which it applies.

\siunitx_declare_unit:Nn
\siunitx_declare_unit:Nx

`\siunitx_declare_unit:Nn` ⟨*unit*⟩ {⟨*meaning*⟩}

Defines a symbolic ⟨*unit*⟩ (which should be a control sequence such as `\kilogram`) to be converted by the parser to the ⟨*meaning*⟩. The latter may consist of literal content (*e.g.* `kg`), other symbolic unit commands (*e.g.* `\kilo\gram`) or a mixture of the two. In literal mode the ⟨*meaning*⟩ will be typeset directly.

`\l_siunitx_unit_symbolic_seq`

This sequence contains all of the symbolic ⟨*unit*⟩ names defined : these will be in the form of control sequences such as `\kilogram`. The order of the sequence is unimportant.

# 3 Pre-defined symbolic unit components

The unit parser is defined to recognise a number of pre-defined units, prefixes and powers, and also interpret a small selection of "generic" symbolic parts.

Broadly, the pre-defined units are those defined by the BIPM in the documentation for the *International System of Units* (SI) [1]. As far as possible, the names given to the command names for units are those used by the BIPM, omitting spaces and using only ASCII characters. The standard symbols are also taken from the same documentation. In the following documentation, the order of the description of units broadly follows the SI Brochure.

`\kilogram`
`\metre`
`\meter`
`\mole`
`\kelvin`
`\candela`
`\second`
`\ampere`

The base units as defined in Section 2.1 of the SI Brochure [2]. Notice that `\meter` is defined as an alias for `\metre` as the former spelling is common in the US (although the latter is the official spelling).

`\gram`

The base unit `\kilogram` is defined using an SI prefix: as such the (derived) unit `\gram` is required by the module to correctly produce output for the `\kilogram`.

\yocto
\zepto
\atto
\femto
\pico
\nano
\micro
\milli
\centi
\deci
\deca
\deka
\hecto
\kilo
\mega
\giga
\tera
\peta
\exa
\zetta
\yotta

Prefixes, all of which are integer powers of 10: the powers are stored internally by the module and can be used for conversion from prefixes to their numerical equivalent. These prefixes are documented in Section 3.1 of the SI Brochure [4].

Note that the `\kilo` prefix is required to define the base `\kilogram` unit. Also note the two spellings available for `\deca`/`\deka`.

\becquerel
\degreeCelsius
\coulomb
\farad
\gray
\hertz
\henry
\joule
\katal
\lumen
\lux
\newton
\ohm
\pascal
\radian
\siemens
\sievert
\steradian
\tesla
\volt
\watt
\weber

The defined SI units with defined names and symbols, as given in Section 2.2.2 of the SI Brochure [3]. Notice that the names of the units are lower case with the exception of `\degreeCelsius`, and that this unit name includes "degree".

\day
\hectare
\hour
\litre
\liter
\minute
\tonne

Units accepted for use with the SI: here `\minute` is a unit of time not of plane angle. These units are taken from Table 4.1 of the SI Brochure [6].

For the unit `\litre`, both `l` and `L` are listed as acceptable symbols: the latter is the standard setting of the module. The alternative spelling `\liter` is also given for this unit for US users (as with `\metre`, the official spelling is "re").

`\arcminute`
`\arcsecond`
`\degree`

Units for plane angles accepted for use with the SI: to avoid a clash with units for time, here `\arcminute` and `\arcsecond` are used in place of `\minute` and `\second`. These units are taken from Table 4.1 of the SI Brochure [6].

`\astronomicalunit`
`\atomicmassunit`
`\auaction`
`\aucharge`
`\auenergy`
`\aulength`
`\aumass`
`\autime`
`\bohr`
`\dalton`
`\electronvolt`
`\hartree`
`\nuaction`
`\numass`
`\nuspeed`
`\nutime`

Non-SI where values must be determined experimentally. These units are taken from Table 7 of the SI Brochure [7]. Where no better name is given for the unit in the SI Brochure, the prefixes `nu` (natural unit) and `au` (atomic unit) are used.

Note that the value of the natural unit of speed (the speed of light) is used to define the second and is thus not determined by experiment: it is however included in this set of units.

`\angstrom`
`\bar`
`\barn`
`\bel`
`\decibel`
`\knot`
`\millimetremercury`
`\nauticalmile`
`\neper`

Non-SI units accepted for use with the SI. These units are taken from Table 8 of the SI Brochure [8].

`\dyne`
`\erg`
`\gal`
`\gauss`
`\maxwell`
`\oersted`
`\phot`
`\poise`
`\stilb`
`\stokes`

Non-SI units associated with the CGS and the CGS-Gaussian system of units. These units are taken from Table 9 of the SI Brochure [9].

`\percent`

The mathematical concept of percent, usable with the SI as detailed in Section 5.3.7 of the SI Brochure [5].

`\square`
`\cubic`

`\square` ⟨*prefix*⟩ ⟨*unit*⟩
`\cubic` ⟨*prefix*⟩ ⟨*unit*⟩

Pre-defined unit powers which apply to the next ⟨*prefix*⟩/⟨*unit*⟩ combination.

\squared ⟨*prefix*⟩ ⟨*unit*⟩ \squared
\cubed ⟨*prefix*⟩ ⟨*unit*⟩ \cubed

Pre-defined unit powers which apply to the preceding ⟨*prefix*⟩/⟨*unit*⟩ combination.

\per \per ⟨*prefix*⟩ ⟨*unit*⟩ ⟨*power*⟩

Indicates that the next ⟨*prefix*⟩/⟨*unit*⟩/⟨*power*⟩ combination is reciprocal, *i.e.* raises it to the power −1. This symbolic representation may be applied in addition to a \power, and will work correctly if the \power itself is negative. In literal mode \per will print a slash ("/").

\cancel \cancel ⟨*prefix*⟩ ⟨*unit*⟩ ⟨*power*⟩

Indicates that the next ⟨*prefix*⟩/⟨*unit*⟩/⟨*power*⟩ combination should be "cancelled out". In the parsed output, the entire unit combination will be given as the argument to a function \cancel, which is assumed to be available at a higher level. In literal mode, the same higher-level \cancel will be applied to the next token. It is the responsibility of the calling code to provide an appropriate definition for \cancel outside of the scope of the unit parser.

\highlight \highlight {⟨*color*⟩} ⟨*prefix*⟩ ⟨*unit*⟩ ⟨*power*⟩

Indicates that the next ⟨*prefix*⟩/⟨*unit*⟩/⟨*power*⟩ combination should be highlighted in the specified ⟨*color*⟩. In the parsed output, the entire unit combination will be given as the argument to a function \textcolor, which is assumed to be available at a higher level. In literal mode, the same higher-level \textcolor will be applied to the next token. It is the responsibility of the calling code to provide an appropriate definition for \textcolor outside of the scope of the unit parser.

\of ⟨*prefix*⟩ ⟨*unit*⟩ ⟨*power*⟩ \of {⟨*qualifier*⟩}

Indicates that the ⟨*qualifier*⟩ applies to the current ⟨*prefix*⟩/⟨*unit*⟩/⟨*power*⟩ combination. In parsed mode, the display of the result will depend upon module options. In literal mode, the ⟨*qualifier*⟩ will be printed in parentheses following the preceding ⟨*unit*⟩ and a full-width space.

\raiseto \raiseto {⟨*power*⟩} ⟨*prefix*⟩ ⟨*unit*⟩
\tothe ⟨*prefix*⟩ ⟨*unit*⟩ \tothe {⟨*power*⟩}

Indicates that the ⟨*power*⟩ applies to the current ⟨*prefix*⟩/⟨*unit*⟩ combination. As shown, \raiseto applies to the next ⟨*unit*⟩ whereas \tothe applies to the preceding unit. In literal mode the \power will be printed as a superscript attached to the next token (\raiseto) or preceding token (\tothe) as appropriate.

## 3.1 Key–value options

The options defined by this submodule are available within the l3keys siunitx tree.

bracket-denominator bracket-denominator = true|false

Switch to determine whether brackets are added to the denominator part of a unit when printed using inline fractional form (with per-mode as repeated-symbol, symbol or symbol-or-fraction). The standard setting is true.

**fraction-command** `fraction-command =` ⟨*command*⟩

Command used to create fractional output when `per-mode` is set to `fraction`. The standard setting is `\frac`.

**parse-units** `parse-units = true|false`

Determines whether parsing of unit symbols is attempted or literal mode is used directly. The standard setting is `true`.

**per-mode** `per-mode =` ⟨*choice*⟩

Selects how the negative powers (`\per`) are formatted: a choice from the options `fraction`, `power`, `power-positive-first`, `repeated-symbol`, `symbol` and `symbol-or-fraction`. The option `fraction` generates fractional output when appropriate using the command specified by the `fraction-command` option. The setting `power` uses reciprocal powers leaving the units in the order of input, while `power-positive-first` uses the same display format but sorts units such that the positive powers come before negative ones. The `symbol` setting uses a symbol (specified by `per-symbol`) between positive and negative powers, while `repeated-symbol` uses the same symbol but places it before *every* unit with a negative power (this is mathematically "wrong" but often seen in real work). Finally, `symbol-or-fraction` acts like `symbol` for inline output and like `fraction` when the output is used in a display math environment. The standard setting is `power`.

**per-symbol** `per-symbol =` ⟨*symbol*⟩

Specifies the symbol to be used to denote negative powers when the option `per-mode` is set to `repeated-symbol`, `symbol` or `symbol-or-fraction`. The standard setting is `/`.

**qualifier-mode** `qualifier-mode =` ⟨*choice*⟩

Selects how qualifiers are formatted: a choice from the options `brackets`, `combine`, `phrase` and `subscript`. The option `bracket` wraps the qualifier in parenthesis, `combine` joins the qualifier with the unit directly, `phrase` inserts the content stored by the option `qualifier-phrase` between the unit and qualifier, and `subscript` formats the qualifier as a subscript. The standard setting is `subscript`.

**qualifier-phrase** `qualifier-phrase =` ⟨*choice*⟩

Defines the text to be inserted between a unit and qualifier when `qualifier-mode` is set to `phrase`. This material is inserted without any font control and so if text mode is required it should be included in the setting, for example `\␣\mbox{of}\␣`. The standard setting is a full width space (`\␣`).

**sticky-per** `sticky-per = true|false`

Used to determine whether `\per` should be applied one a unit-by-unit basis (when `false`) or should apply to all following units (when `true`). The latter mode is somewhat akin conceptually to the TEX `\over` primitive. The standard setting is `false`.

**unit-close-bracket** `unit-close-bracket =` ⟨*symbol*⟩

Bracket symbol used to close a matched pair around units when once is required to maintain mathematical logic. The standard setting is `)`.

unit-open-bracket `unit-open-bracket = ⟨symbol⟩`

Bracket symbol used to open a matched pair around units when once is required to maintain mathematical logic. The standard setting is `(`.

unit-product `unit-product = ⟨separator⟩`

Inserted between unit combinations in parsed mode, and used to replace `.` and `~` in literal mode. The standard setting is `\,`.

# 4 siunitx-unit implementation

Start the DocStrip guards.

```
1 ⟨*package⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention): only internal material in this *submodule* should be used directly.

```
2 ⟨@@=siunitx_unit⟩
```

## 4.1 Initial set up

The mechanisms defined here need a few variables to exist and to be correctly set: these don't belong to one subsection and so are created in a small general block.

Variants not provided by expl3.

```
3 \cs_generate_variant:Nn \tl_replace_all:Nnn { NnV }
```

\l__siunitx_unit_tmp_fp
\l__siunitx_unit_tmp_int
\l__siunitx_unit_tmp_tl

Scratch space.

```
4 \fp_new:N \l__siunitx_unit_tmp_fp
5 \int_new:N \l__siunitx_unit_tmp_int
6 \tl_new:N  \l__siunitx_unit_tmp_tl
```

(*End definition for* `\l__siunitx_unit_tmp_fp`, `\l__siunitx_unit_tmp_int`, *and* `\l__siunitx_unit_tmp_tl`.)

\c__siunitx_unit_math_subscript_tl

Useful tokens with awkward category codes.

```
7 \tl_const:Nx \c__siunitx_unit_math_subscript_tl
8   { \char_generate:nn { `\_ } { 8 } }
```

(*End definition for* `\c__siunitx_unit_math_subscript_tl`.)

\l__siunitx_unit_parsing_bool

A boolean is used to indicate when the symbolic unit functions should produce symbolic or literal output. This is used when the symbolic names are used along with literal input, and ensures that there is a sensible fall-back for these cases.

```
9 \bool_new:N \l__siunitx_unit_parsing_bool
```

(*End definition for* `\l__siunitx_unit_parsing_bool`.)

\l__siunitx_unit_test_bool

A switch used to indicate that the code is testing the input to find if there is any typeset output from individual unit macros. This is needed to allow the "base" macros to be found, and also to pick up the difference between symbolic and literal unit input.

```
10 \bool_new:N \l__siunitx_unit_test_bool
```

(*End definition for* `\l__siunitx_unit_test_bool`.)

__siunitx_unit_if_symbolic:nTF

The test for symbolic units is needed in two places. First, there is the case of "pre-parsing" input to check if it can be parsed. Second, when parsing there is a need to check if the current unit is built up from others (symbolic) or is defined in terms of some literals. To do this, the approach used is to set all of the symbolic unit commands expandable and to do nothing, with the few special cases handled manually. If an `f`-type definition then yields nothing at all then the assumption is that the input is symbolic. (We use `f`-type expansion since it will turn the symbolic unit names into nothing at all but doesn't require them to be expandable.)

```
\prg_new_protected_conditional:Npnn \__siunitx_unit_if_symbolic:n #1 { TF }
  {
    \group_begin:
      \bool_set_true:N \l__siunitx_unit_test_bool
      \tl_set:Nf \l__siunitx_unit_tmp_tl {#1}
    \exp_args:NNV \group_end:
    \tl_if_blank:nTF \l__siunitx_unit_tmp_tl
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\__siunitx_unit_if_symbolic:nTF`.)

## 4.2 Defining symbolic unit

Unit macros and related support are created here. These exist only within the scope of the unit processor code, thus not polluting document-level namespace and allowing overlap with other areas in the case of useful short names (for example `\pm`). Setting up the mechanisms to allow this requires a few additional steps on top of simply saving the data given by the user in creating the unit.

\l_siunitx_unit_symbolic_seq

A list of all of the symbolic units, *etc.*, set up. This is needed to allow the symbolic names to be defined within the scope of the unit parser but not elsewhere using simple mappings.

```
\seq_new:N \l_siunitx_unit_symbolic_seq
```

(*End definition for* `\l_siunitx_unit_symbolic_seq`. *This variable is documented on page* 57.)

__siunitx_unit_set_symbolic:Nnn
__siunitx_unit_set_symbolic:Npnn
__siunitx_unit_set_symbolic:NNpnn

The majority of the work for saving each symbolic definition is the same irrespective of the item being defined (unit, prefix, power, qualifier). This is therefore all carried out in a single internal function which does the common tasks. The three arguments here are the symbolic macro name, the literal output and the code to insert when doing full unit parsing. To allow for the "special cases" (where arguments are required) the entire mechanism is set up in a two-part fashion allowing for flexibility at the slight cost of additional functions.

Importantly, notice that the unit macros are declared as expandable. This is required so that literals can be correctly converted into a token list of material which does not depend on local redefinitions for the unit macros. That is required so that the unit formatting system can be grouped.

```
\cs_new_protected:Npn \__siunitx_unit_set_symbolic:Nnn #1
  { \__siunitx_unit_set_symbolic:NNnnn \cs_set:cpn #1 { } }
\cs_new_protected:Npn \__siunitx_unit_set_symbolic:Npnn #1#2#
  { \__siunitx_unit_set_symbolic:NNnnn \cs_set:cpn #1 {#2} }
\cs_new_protected:Npn \__siunitx_unit_set_symbolic:NNnnn #1#2#3#4#5
```

```
  {
    \seq_put_right:Nn \l_siunitx_unit_symbolic_seq {#2}
    #1 { units ~ > ~ \token_to_str:N #2 } #3
      {
        \bool_if:NF \l__siunitx_unit_test_bool
          {
            \bool_if:NTF \l__siunitx_unit_parsing_bool
              {#5}
              {#4}
          }
      }
  }
```

(*End definition for* `\__siunitx_unit_set_symbolic:Nnn`, `\__siunitx_unit_set_symbolic:Npnn`, *and* `\__siunitx_unit_set_symbolic:NNpnn`.)

`\siunitx_declare_power:NNn`

Powers can come either before or after the unit. As they always come (logically) in matching, we handle this by declaring two commands, and setting each up separately.

```
\cs_new_protected:Npn \siunitx_declare_power:NNn #1#2#3
  {
    \__siunitx_unit_set_symbolic:Nnn #1
      { \__siunitx_unit_literal_power:nN {#1} }
      { \__siunitx_unit_parse_power:nnN {#1} {#3} \c_true_bool }
    \__siunitx_unit_set_symbolic:Nnn #2
      { ^ {#3} }
      { \__siunitx_unit_parse_power:nnN {#2} {#3} \c_false_bool }
  }
```

(*End definition for* `\siunitx_declare_power:NNn`. *This function is documented on page* 56.)

`\siunitx_declare_prefix:Nn`
`\siunitx_declare_prefix:Nnn`
`\l__siunitx_unit_prefixes_forward_prop`
`\l__siunitx_unit_prefixes_reverse_prop`

For prefixes there are a couple of options. In all cases, the basic requirement is to set up to parse the prefix using the appropriate internal function. For prefixes which are powers of 10, there is also the need to be able to do conversion to/from the numerical equivalent. That is handled using two properly lists which can be used to supply the conversion data later.

```
\cs_new_protected:Npn \siunitx_declare_prefix:Nn #1#2
  {
    \__siunitx_unit_set_symbolic:Nnn #1
      {#2}
      { \__siunitx_unit_parse_prefix:Nn #1 {#2} }
  }
\cs_new_protected:Npn \siunitx_declare_prefix:Nnn #1#2#3
  {
    \siunitx_declare_prefix:Nn #1 {#2}
    \prop_put:Nnn \l__siunitx_unit_prefixes_forward_prop {#2} {#3}
    \prop_put:Nnn \l__siunitx_unit_prefixes_reverse_prop {#3} {#2}
  }
\prop_new:N \l__siunitx_unit_prefixes_forward_prop
\prop_new:N \l__siunitx_unit_prefixes_reverse_prop
```

(*End definition for* `\siunitx_declare_prefix:Nn` *and others. These functions are documented on page* 56.)

\siunitx_declare_qualifier:Nn

Qualifiers are relatively easy to handle: nothing to do other than save the input appropriately.

```
\cs_new_protected:Npn \siunitx_declare_qualifier:Nn #1#2
  {
    \__siunitx_unit_set_symbolic:Nnn #1
      { \  ( #2 ) }
      { \__siunitx_unit_parse_qualifier:nn {#1} {#2} }
  }
```

(*End definition for* `\siunitx_declare_qualifier:Nn`*. This function is documented on page 56.*)

\siunitx_declare_unit:Nn
\siunitx_declare_unit:Nx

For the unit parsing, allowing for variations in definition order requires that a test is made for the output of each unit at point of use.

```
\cs_new_protected:Npn \siunitx_declare_unit:Nn #1#2
  {
    \__siunitx_unit_set_symbolic:Nnn #1
      {#2}
      {
        \__siunitx_unit_if_symbolic:nTF {#2}
          {#2}
          { \__siunitx_unit_parse_unit:Nn #1 {#2} }
      }
  }
\cs_generate_variant:Nn \siunitx_declare_unit:Nn { Nx }
```

(*End definition for* `\siunitx_declare_unit:Nn`*. This function is documented on page 57.*)

## 4.3 Non-standard symbolic units

A few of the symbolic units require non-standard definitions: these are created here. They all use parts of the more general code but have particular requirements which can only be addressed by hand. Some of these could in principle be used in place of the dedicated definitions above, but at point of use that would then require additional expansions for each unit parsed: as the macro names would still be needed, this does not offer any real benefits.

\per

The `\per` symbolic unit is a bit special: it has a mechanism entirely different from everything else, so has to be set up by hand. In literal mode it is represented by a very simple symbol!

```
\__siunitx_unit_set_symbolic:Nnn \per
  { / }
  { \__siunitx_unit_parse_per: }
```

(*End definition for* `\per`*. This function is documented on page 60.*)

\cancel
\highlight

The two special cases, `\cancel` and `\highlight`, are easy to deal with when parsing. When not parsing, a precaution is taken to ensure that the user level equivalents always get a braced argument.

```
\__siunitx_unit_set_symbolic:Npnn \cancel
  { \__siunitx_unit_literal_special:nN { \cancel } }
  { \__siunitx_unit_parse_special:n { \cancel } }
\__siunitx_unit_set_symbolic:Npnn \highlight #1
  { \__siunitx_unit_literal_special:nN { \textcolor {#1} } }
  { \__siunitx_unit_parse_special:n { \textcolor {#1} } }
```

(*End definition for* `\cancel` *and* `\highlight`*. These functions are documented on page 60.*)

`\of` The generic qualifier is simply the same as the dedicated ones except for needing to grab an argument.

```
\__siunitx_unit_set_symbolic:Npnn \of #1
  { \  ( #1 ) }
  { \__siunitx_unit_parse_qualifier:nn { \of {#1} } {#1} }
```

(*End definition for* `\of`*. This function is documented on page 60.*)

`\raiseto` `\tothe` Generic versions of the pre-defined power macros. These require an argument and so cannot be handled using the general approach. Other than that, the code here is very similar to that in `\siunitx_unit_power_set:NnN`.

```
\__siunitx_unit_set_symbolic:Npnn \raiseto #1
  { \__siunitx_unit_literal_power:nN {#1} }
  { \__siunitx_unit_parse_power:nnN { \raiseto {#1} } {#1} \c_true_bool }
\__siunitx_unit_set_symbolic:Npnn \tothe #1
  { ^ {#1} }
  { \__siunitx_unit_parse_power:nnN { \tothe {#1} } {#1} \c_false_bool }
```

(*End definition for* `\raiseto` *and* `\tothe`*. These functions are documented on page 60.*)

## 4.4 Main formatting routine

Unit input can take two forms, "literal" units (material to be typeset directly) or "symbolic" units (macro-based). Before any parsing or typesetting is carried out, a small amount of pre-parsing has to be carried out to decide which of these cases applies.

`\l__siunitx_unit_product_tl` Options which apply to the main formatting routine, and so are not tied to either symbolic or literal input.

```
\keys_define:nn { siunitx }
  {
    unit-product .tl_set:N = \l__siunitx_unit_product_tl
  }
```

(*End definition for* `\l__siunitx_unit_product_tl`.)

`\l__siunitx_unit_formatted_tl` A token list for the final formatted result: may or may not be generated by the parser, depending on the nature of the input.

```
\tl_new:N \l__siunitx_unit_formatted_tl
```

(*End definition for* `\l__siunitx_unit_formatted_tl`.)

`\siunitx_unit_format:nN` `\siunitx_unit_format:nNN` `\__siunitx_unit_format:nNN` `\__siunitx_unit_format_aux:` Formatting parsed units can take place either with the prefixes printed or separated out into a power of ten. This variation is handled using two separate functions: as this submodule does not really deal with numbers, formatting the numeral part here would be tricky and it is better therefore to have a mechanism to return a simple numerical power. At the same time, most uses will no want this more complex return format and so a version of the code which does not do this is also provided.

The main unit formatting routine groups all of the parsing/formatting, so that the only value altered will be the return token list. As definitions for the various unit macros are not globally created, the first step is to map over the list of names and active the unit definitions: these do different things depending on the switches set. There is then

a decision to be made: is the unit input one that can be parsed ("symbolic"), or is is one containing one or more literals. In the latter case, there is a still the need to convert the input into an expanded token list as some parts of the input could still be using unit macros.

Notice that for `\siunitx_unit_format:nN` a second return value from the auxiliary has to be allowed for, but is simply discarded.

```
\cs_new_protected:Npn \siunitx_unit_format:nN #1#2
  {
    \bool_set_false:N \l__siunitx_unit_prefix_power_bool
    \__siunitx_unit_format:nNN {#1} #2 \l__siunitx_unit_tmp_fp
  }
\cs_new_protected:Npn \siunitx_unit_format:nNN #1#2#3
  {
    \bool_set_true:N \l__siunitx_unit_prefix_power_bool
    \__siunitx_unit_format:nNN {#1} #2 #3
  }
\cs_new_protected:Npn \__siunitx_unit_format:nNN #1#2#3
  {
    \group_begin:
      \seq_map_inline:Nn \l_siunitx_unit_symbolic_seq
        { \cs_set_eq:Nc ##1 { units ~ > ~ \token_to_str:N ##1 } }
      \tl_clear:N \l__siunitx_unit_formatted_tl
      \fp_zero:N \l__siunitx_unit_prefix_fp
      \bool_if:NTF \l__siunitx_unit_parse_bool
        {
          \__siunitx_unit_if_symbolic:nTF {#1}
            {
              \__siunitx_unit_parse:n {#1}
              \prop_if_empty:NF \l__siunitx_unit_parsed_prop
                { \__siunitx_unit_format_parsed: }
            }
            { \__siunitx_unit_format_literal:n {#1} }
        }
        { \__siunitx_unit_format_literal:n {#1} }
      \cs_set_protected:Npx \__siunitx_unit_format_aux:
        {
          \tl_set:Nn \exp_not:N #2
            { \exp_not:V \l__siunitx_unit_formatted_tl }
          \fp_set:Nn \exp_not:N #3
            { \fp_use:N \l__siunitx_unit_prefix_fp }
        }
    \exp_after:wN \group_end:
    \__siunitx_unit_format_aux:
  }
\cs_new_protected:Npn \__siunitx_unit_format_aux: { }
```

(*End definition for* `\siunitx_unit_format:nN` *and others. These functions are documented on page* 55.)

## 4.5 Formatting literal units

While in literal mode no parsing occurs, there is a need to provide a few auxiliary functions to handle one or two special cases.

__siunitx_unit_literal_power:nN For printing literal units which are given before the unit they apply to, there is a slight rearrangement.

```
\cs_new_protected:Npn \__siunitx_unit_literal_power:nN #1#2 { #2 ^ {#1} }
```

(*End definition for* `\__siunitx_unit_literal_power:nN`.)

__siunitx_unit_literal_special:nN When dealing with the special cases, there is an argument to absorb. This should be braced to be passed up to the user level, which is dealt with here.

```
\cs_new_protected:Npn \__siunitx_unit_literal_special:nN #1#2 { #1 {#2} }
```

(*End definition for* `\__siunitx_unit_literal_special:nN`.)

__siunitx_unit_format_literal:n
__siunitx_unit_format_literal_auxi:w
__siunitx_unit_format_literal_auxii:w
__siunitx_unit_format_literal_auxiii:w
__siunitx_unit_format_literal_auxiv:w
__siunitx_unit_format_literal_auxv:w
\l__siunitx_unit_separator_tl

To format literal units, there are two tasks to do. The input is `x`-type expanded to force any symbolic units to be converted into their literal representation: this requires setting the appropriate switch. In the resulting token list, all `.` and `~` tokens are then replaced by the current unit product token list. To enable this to happen correctly with a normal (active) `~`, a small amount of "protection" is needed first.

As with other code dealing with user input, `\protected@edef` is used here rather than `\tl_set:Nx` as LaTeX $2_\varepsilon$ robust commands may be present.

```
\group_begin:
  \char_set_catcode_active:n { `\~ }
  \cs_new_protected:Npx \__siunitx_unit_format_literal:n #1
    {
      \group_begin:
        \exp_not:n { \bool_set_false:N \l__siunitx_unit_parsing_bool }
        \tl_set:Nn \exp_not:N \l__siunitx_unit_tmp_tl {#1}
        \tl_replace_all:Nnn \exp_not:N \l__siunitx_unit_tmp_tl
          { \exp_not:N ~ } { . }
        \tl_replace_all:Nnn \exp_not:N \l__siunitx_unit_tmp_tl
          { \token_to_str:N ^ } { ^ }
        \tl_replace_all:Nnn \exp_not:N \l__siunitx_unit_tmp_tl
          { \token_to_str:N _ } { \c__siunitx_unit_math_subscript_tl }
        \exp_not:n
          {
              \protected@edef \l__siunitx_unit_tmp_tl
                { \l__siunitx_unit_tmp_tl }
              \tl_clear:N \l__siunitx_unit_formatted_tl
              \tl_if_empty:NF \l__siunitx_unit_tmp_tl
                {
                  \exp_after:wN \__siunitx_unit_format_literal_auxi:w
                    \l__siunitx_unit_tmp_tl .
                    \q_recursion_tail . \q_recursion_stop
                }
            \exp_args:NNNV \group_end:
            \tl_set:Nn \l__siunitx_unit_formatted_tl
              \l__siunitx_unit_formatted_tl
          }
    }
\group_end:
```

To introduce the font changing commands while still allowing for line breaks in literal units, a loop is needed to replace one `.` at a time. To also allow for division, a second loop is used within that to handle `/`: as a result, the separator between parts has to be tracked.

```
\cs_new_protected:Npn \__siunitx_unit_format_literal_auxi:w #1 .
  {
    \quark_if_recursion_tail_stop:n {#1}
    \__siunitx_unit_format_literal_auxii:n {#1}
    \tl_set_eq:NN \l__siunitx_unit_separator_tl \l__siunitx_unit_product_tl
    \__siunitx_unit_format_literal_auxi:w
  }
\cs_set_protected:Npn \__siunitx_unit_format_literal_auxii:n #1
  {
    \__siunitx_unit_format_literal_auxiii:w
      #1 / \q_recursion_tail / \q_recursion_stop
  }
\cs_new_protected:Npn \__siunitx_unit_format_literal_auxiii:w #1 /
  {
    \quark_if_recursion_tail_stop:n {#1}
    \__siunitx_unit_format_literal_auxiv:w #1 ^ ^ \q_stop
    \tl_set:Nn \l__siunitx_unit_separator_tl { / }
    \__siunitx_unit_format_literal_auxiii:w
  }
```

Within each unit any sub- and superscript parts need to be separated out: wrapping everything in the font command is incorrect. The superscript part is relatively easy as there is no catcode issue or font command to add, while the subscript part needs a bit more work. As the user might have the two parts in either order, picking up the subscript needs to look in two places. We assume that only one is given: odd input here is simply accepted.

```
\use:x
  {
    \cs_new_protected:Npn \exp_not:N \__siunitx_unit_format_literal_auxiv:w
      ##1 ^ ##2 ^ ##3 \exp_not:N \q_stop
      {
        \exp_not:N \__siunitx_unit_format_literal_auxv:w
          ##1
          \c__siunitx_unit_math_subscript_tl
          \c__siunitx_unit_math_subscript_tl
          \exp_not:N \q_mark
          ##2
          \c__siunitx_unit_math_subscript_tl
          \c__siunitx_unit_math_subscript_tl
          \exp_not:N \q_stop
      }
    \cs_new_protected:Npn \exp_not:N \__siunitx_unit_format_literal_auxv:w
      ##1 \c__siunitx_unit_math_subscript_tl
      ##2 \c__siunitx_unit_math_subscript_tl ##3
      \exp_not:N \q_mark
      ##4 \c__siunitx_unit_math_subscript_tl
      ##5 \c__siunitx_unit_math_subscript_tl ##6
      \exp_not:N \q_stop
      {
        \tl_set:Nx \exp_not:N \l__siunitx_unit_formatted_tl
          {
            \exp_not:N \exp_not:V
              \exp_not:N \l__siunitx_unit_formatted_tl
            \exp_not:N \tl_if_empty:NF
```

```
              \exp_not:N \l__siunitx_unit_formatted_tl
              {
                \exp_not:N \exp_not:V
                  \exp_not:N \l__siunitx_unit_separator_tl
              }
            \exp_not:N \tl_if_blank:nF {##1}
              {
                \exp_not:N \exp_not:N
                  \exp_not:N \mathrm
                  { \exp_not:N \exp_not:n {##1} }
              }
            \exp_not:N \tl_if_blank:nF {##4}
              { ^ { \exp_not:N \exp_not:n {##4} } }
            \exp_not:N \tl_if_blank:nF {##2##5}
              {
                \c__siunitx_unit_math_subscript_tl
                  {
                    \exp_not:N \exp_not:N
                      \exp_not:N \mathrm
                      { \exp_not:N \exp_not:n {##2##5} }
                  }
              }
          }
      }
  }
\tl_new:N \l__siunitx_unit_separator_tl
```

(*End definition for* `\__siunitx_unit_format_literal:n` *and others.*)

## 4.6 Parsing symbolic units

Parsing units takes place by storing information about each unit in a `prop`. As well as the unit itself, there are various other optional data points, for example a prefix or a power. Some of these can come before the unit, others only after. The parser therefore tracks the number of units read and uses the current position to allocate data to individual units.

The result of parsing is a property list (`\l__siunitx_unit_parsed_prop`) which contains one or more entries for each unit:

- `prefix-`$n$ The symbol for the prefix which applies to this unit, *e.g.* for `\kilo` with (almost certainly) would be `k`.
- `unit-`$n$ The symbol for the unit itself, *e.g.* for `\metre` with (almost certainly) would be `m`.
- `power-`$n$ The power which a unit is raised to. During initial parsing this will (almost certainly) be positive, but is combined with `per-`$n$ to give a "fully qualified" power before any formatting takes place
- `per-`$n$ Indicates that `per` applies to the current unit: stored during initial parsing then combined with `power-`$n$ (and removed from the list) before further work.
- `qualifier-`$n$ Any qualifier which applies to the current unit.
- `special-`$n$ Any "special effect" to apply to the current unit.

\l__siunitx_unit_sticky_per_bool There is one option when *parsing* the input (as opposed to *formatting* for output): how to deal with \per.

```
\keys_define:nn { siunitx }
  {
    sticky-per .bool_set:N = \l__siunitx_unit_sticky_per_bool
  }
```

(*End definition for* \l__siunitx_unit_sticky_per_bool.)

\l__siunitx_unit_parsed_prop
\l__siunitx_unit_per_bool
\l__siunitx_unit_position_int Parsing units requires a small number of variables are available: a prop for the parsed units themselves, a bool to indicate if \per is active and an int to track how many units have be parsed.

```
\prop_new:N \l__siunitx_unit_parsed_prop
\bool_new:N \l__siunitx_unit_per_bool
\int_new:N \l__siunitx_unit_position_int
```

(*End definition for* \l__siunitx_unit_parsed_prop, \l__siunitx_unit_per_bool, *and* \l__siunitx_unit_position_int.)

__siunitx_unit_parse:n The main parsing function is quite simple. After initialising the variables, each symbolic unit is set up. The input is then simply inserted into the input stream: the symbolic units themselves then do the real work of placing data into the parsing system. There is then a bit of tidying up to ensure that later stages can rely on the nature of the data here.

```
\cs_new_protected:Npn \__siunitx_unit_parse:n #1
  {
    \prop_clear:N \l__siunitx_unit_parsed_prop
    \bool_set_true:N \l__siunitx_unit_parsing_bool
    \bool_set_false:N \l__siunitx_unit_per_bool
    \bool_set_false:N \l__siunitx_unit_test_bool
    \int_zero:N \l__siunitx_unit_position_int
    #1
    \int_step_inline:nnnn 1 1 \l__siunitx_unit_position_int
      { \__siunitx_unit_parse_finalise:n {##1} }
    \__siunitx_unit_parse_finalise:
  }
```

(*End definition for* __siunitx_unit_parse:n.)

__siunitx_unit_parse_add:nnnn In all cases, storing a data item requires setting a temporary tl which will be used as the key, then using this to store the value. The tl is set using x-type expansion as this will expand the unit index and any additional calculations made for this.

```
\cs_new_protected:Npn \__siunitx_unit_parse_add:nnnn #1#2#3#4
  {
    \tl_set:Nx \l__siunitx_unit_tmp_tl { #1 - #2 }
    \prop_if_in:NVTF \l__siunitx_unit_parsed_prop
      \l__siunitx_unit_tmp_tl
      {
        \msg_error:nnxx { siunitx } { unit / duplicate-part }
          { \exp_not:n {#1} } { \token_to_str:N #3 }
      }
      {
        \prop_put:NVn \l__siunitx_unit_parsed_prop
          \l__siunitx_unit_tmp_tl {#4}
```

```
      }
  }
```

(*End definition for* `\__siunitx_unit_parse_add:nnnn`.)

`\__siunitx_unit_parse_prefix:Nn`
`\__siunitx_unit_parse_power:nnN`
`\__siunitx_unit_parse_qualifier:nn`
`\__siunitx_unit_parse_special:n`

Storage of the various optional items follows broadly the same pattern in each case. The data to be stored is passed along with an appropriate key name to the underlying storage system. The details for each type of item should be relatively clear. For example, prefixes have to come before their "parent" unit and so there is some adjustment to do to add them to the correct unit.

```
\cs_new_protected:Npn \__siunitx_unit_parse_prefix:Nn #1#2
  {
    \int_set:Nn \l__siunitx_unit_tmp_int { \l__siunitx_unit_position_int + 1 }
    \__siunitx_unit_parse_add:nnnn { prefix }
      { \int_use:N \l__siunitx_unit_tmp_int } {#1} {#2}
  }
\cs_new_protected:Npn \__siunitx_unit_parse_power:nnN #1#2#3
  {
    \tl_set:Nx \l__siunitx_unit_tmp_tl
      { unit- \int_use:N \l__siunitx_unit_position_int }
    \bool_lazy_or:nnTF
      {#3}
      {
        \prop_if_in_p:NV
          \l__siunitx_unit_parsed_prop \l__siunitx_unit_tmp_tl
      }
      {
        \__siunitx_unit_parse_add:nnnn { power }
          {
            \int_eval:n
              { \l__siunitx_unit_position_int \bool_if:NT #3 { + 1 } }
          }
          {#1} {#2}
      }
      {
        \msg_error:nnxx { siunitx }
          { unit / part-before-unit } { power } { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \__siunitx_unit_parse_qualifier:nn #1#2
  {
    \tl_set:Nx \l__siunitx_unit_tmp_tl
      { unit- \int_use:N \l__siunitx_unit_position_int }
    \prop_if_in:NVTF \l__siunitx_unit_parsed_prop \l__siunitx_unit_tmp_tl
      {
        \__siunitx_unit_parse_add:nnnn { qualifier }
          { \int_use:N \l__siunitx_unit_position_int } {#1} {#2}
      }
      {
        \msg_error:nnnn { siunitx }
          { unit / part-before-unit } { qualifier } { \token_to_str:N #1 }
      }
  }
```

Special (exceptional) items should always come before the relevant units.

```
\cs_new_protected:Npn \__siunitx_unit_parse_special:n #1
  {
    \__siunitx_unit_parse_add:nnnn { special }
      { \int_eval:n { \l__siunitx_unit_position_int + 1 } }
      {#1} {#1}
  }
```

(*End definition for* `\__siunitx_unit_parse_prefix:Nn` *and others.*)

`\__siunitx_unit_parse_unit:Nn` Parsing units is slightly more involved than the other cases: this is the one place where the tracking value is incremented. If the switch `\l__siunitx_unit_per_bool` is set true then the current unit is also reciprocal: this can only happen if `\l__siunitx_unit_sticky_per_bool` is also true, so only one test is required.

```
\cs_new_protected:Npn \__siunitx_unit_parse_unit:Nn #1#2
  {
    \int_incr:N \l__siunitx_unit_position_int
    \__siunitx_unit_parse_add:nnnn { unit }
      { \int_use:N \l__siunitx_unit_position_int }
      {#1} {#2}
    \bool_if:NT \l__siunitx_unit_per_bool
      {
        \__siunitx_unit_parse_add:nnnn { per }
          { \int_use:N \l__siunitx_unit_position_int }
          { \per } { true }
      }
  }
```

(*End definition for* `\__siunitx_unit_parse_unit:Nn`.)

`\__siunitx_unit_parse_per:` Storing the `\per` command requires adding a data item separate from the power which applies: this makes later formatting much more straight-forward. This data could in principle be combined with the `power`, but depending on the output format required that may make life more complex. Thus this information is stored separately for later retrieval. If `\per` is set to be "sticky" then after parsing the first occurrence, any further uses are in error.

```
\cs_new_protected:Npn \__siunitx_unit_parse_per:
  {
    \bool_if:NTF \l__siunitx_unit_sticky_per_bool
      {
        \bool_set_true:N \l__siunitx_unit_per_bool
        \cs_set_protected:Npn \per
          { \msg_error:nn { siunitx } { unit / duplicate-sticky-per } }
      }
      {
        \__siunitx_unit_parse_add:nnnn
          { per } { \int_eval:n { \l__siunitx_unit_position_int + 1 } }
          { \per } { true }
      }
  }
```

(*End definition for* `\__siunitx_unit_parse_per:`.)

__siunitx_unit_parse_finalise:n If \per applies to the current unit, the power needs to be multiplied by −1. That is done using an fp operation so that non-integer powers are supported. The flag for \per is also removed as this means we don't have to check that the original power was positive. To be on the safe side, there is a check for a trivial power at this stage.

```
\cs_new_protected:Npn \__siunitx_unit_parse_finalise:n #1
  {
    \tl_set:Nx \l__siunitx_unit_tmp_tl { per- #1 }
    \prop_if_in:NVT \l__siunitx_unit_parsed_prop \l__siunitx_unit_tmp_tl
      {
        \prop_remove:NV \l__siunitx_unit_parsed_prop
          \l__siunitx_unit_tmp_tl
        \tl_set:Nx \l__siunitx_unit_tmp_tl { power- #1 }
        \prop_get:NVNTF
          \l__siunitx_unit_parsed_prop
          \l__siunitx_unit_tmp_tl
          \l__siunitx_unit_part_tl
          {
            \tl_set:Nx \l__siunitx_unit_part_tl
              { \fp_eval:n { \l__siunitx_unit_part_tl * -1 } }
            \fp_compare:nNnTF \l__siunitx_unit_part_tl = 1
              {
                \prop_remove:NV \l__siunitx_unit_parsed_prop
                  \l__siunitx_unit_tmp_tl
              }
              {
                \prop_put:NVV \l__siunitx_unit_parsed_prop
                  \l__siunitx_unit_tmp_tl \l__siunitx_unit_part_tl
              }
          }
          {
            \prop_put:NVn \l__siunitx_unit_parsed_prop
              \l__siunitx_unit_tmp_tl { -1 }
          }
      }
  }
```

(*End definition for* __siunitx_unit_parse_finalise:n.)

__siunitx_unit_parse_finalise: The final task is to check that there is not a "dangling" power or prefix: these are added to the "next" unit so are easy to test for.

```
\cs_new_protected:Npn \__siunitx_unit_parse_finalise:
  {
    \clist_map_inline:nn { per , power , prefix }
      {
        \tl_set:Nx \l__siunitx_unit_tmp_tl
          { ##1 - \int_eval:n { \l__siunitx_unit_position_int + 1 } }
        \prop_if_in:NVT \l__siunitx_unit_parsed_prop \l__siunitx_unit_tmp_tl
          { \msg_error:nnn { siunitx } { unit / dangling-part } { ##1 } }
      }
  }
```

(*End definition for* __siunitx_unit_parse_finalise:.)

## 4.7 Formatting parsed units

\l__siunitx_unit_denominator_bracket_bool
\l__siunitx_unit_fraction_function_tl
\l__siunitx_unit_bracket_close_tl
\l__siunitx_unit_bracket_open_tl
\l__siunitx_unit_parse_bool
\l__siunitx_unit_per_symbol_tl
\l__siunitx_unit_qualifier_mode_tl
\l__siunitx_unit_qualifier_phrase_tl

Set up the options which apply to formatting.

```
\keys_define:nn { siunitx }
  {
    bracket-denominator .bool_set:N =
      \l__siunitx_unit_denominator_bracket_bool ,
    fraction-command .tl_set:N =
      \l__siunitx_unit_fraction_function_tl ,
    parse-units .bool_set:N =
      \l__siunitx_unit_parse_bool ,
    per-mode .choice: ,
    per-mode / fraction .code:n =
      {
        \bool_set_false:N \l__siunitx_unit_autofrac_bool
        \bool_set_false:N \l__siunitx_unit_per_symbol_bool
        \bool_set_true:N  \l__siunitx_unit_powers_positive_bool
        \bool_set_true:N  \l__siunitx_unit_two_part_bool
      } ,
    per-mode / power .code:n =
      {
        \bool_set_false:N \l__siunitx_unit_autofrac_bool
        \bool_set_false:N \l__siunitx_unit_per_symbol_bool
        \bool_set_false:N \l__siunitx_unit_powers_positive_bool
        \bool_set_false:N \l__siunitx_unit_two_part_bool
      } ,
    per-mode / power-positive-first .code:n =
      {
        \bool_set_false:N \l__siunitx_unit_autofrac_bool
        \bool_set_false:N \l__siunitx_unit_per_symbol_bool
        \bool_set_false:N \l__siunitx_unit_powers_positive_bool
        \bool_set_true:N  \l__siunitx_unit_two_part_bool
      } ,
    per-mode / repeated-symbol .code:n =
      {
        \bool_set_false:N \l__siunitx_unit_autofrac_bool
        \bool_set_true:N  \l__siunitx_unit_per_symbol_bool
        \bool_set_true:N  \l__siunitx_unit_powers_positive_bool
        \bool_set_false:N \l__siunitx_unit_two_part_bool
      } ,
    per-mode / symbol .code:n =
      {
        \bool_set_false:N \l__siunitx_unit_autofrac_bool
        \bool_set_true:N  \l__siunitx_unit_per_symbol_bool
        \bool_set_true:N  \l__siunitx_unit_powers_positive_bool
        \bool_set_true:N  \l__siunitx_unit_two_part_bool
      } ,
    per-mode / symbol-or-fraction .code:n =
      {
        \bool_set_true:N \l__siunitx_unit_autofrac_bool
        \bool_set_true:N \l__siunitx_unit_per_symbol_bool
        \bool_set_true:N \l__siunitx_unit_powers_positive_bool
        \bool_set_true:N \l__siunitx_unit_two_part_bool
      } ,
```

```
    per-symbol .tl_set:N =
      \l__siunitx_unit_per_symbol_tl ,
    qualifier-mode .choice: ,
    qualifier-mode / bracket .code:n =
      { \tl_set:Nn \l__siunitx_unit_qualifier_mode_tl { bracket  } }  ,
    qualifier-mode / combine .code:n =
      { \tl_set:Nn \l__siunitx_unit_qualifier_mode_tl { combine } }   ,
    qualifier-mode / phrase .code:n =
      { \tl_set:Nn \l__siunitx_unit_qualifier_mode_tl { phrase } }    ,
    qualifier-mode / subscript .code:n =
      { \tl_set:Nn \l__siunitx_unit_qualifier_mode_tl { subscript } } ,
    qualifier-phrase .tl_set:N =
      \l__siunitx_unit_qualifier_phrase_tl ,
    unit-close-bracket .tl_set:N =
      \l__siunitx_unit_bracket_close_tl ,
    unit-open-bracket .tl_set:N =
      \l__siunitx_unit_bracket_open_tl
  }
```

(*End definition for* `\l__siunitx_unit_denominator_bracket_bool` *and others.*)

`\l__siunitx_unit_bracket_bool` A flag to indicate that the unit currently under construction will require brackets if a power is added.

```
\bool_new:N \l__siunitx_unit_bracket_bool
```

(*End definition for* `\l__siunitx_unit_bracket_bool`.)

`\l__siunitx_unit_font_bool` A flag to control when font wrapping is applied to the output.

```
\bool_new:N \l__siunitx_unit_font_bool
```

(*End definition for* `\l__siunitx_unit_font_bool`.)

`\l__siunitx_unit_autofrac_bool` `\l__siunitx_unit_powers_positive_bool` `\l__siunitx_unit_per_symbol_bool` `\l__siunitx_unit_two_part_bool` Dealing with the various ways that reciprocal (`\per`) can be handled requires a few different switches.

```
\bool_new:N \l__siunitx_unit_autofrac_bool
\bool_new:N \l__siunitx_unit_per_symbol_bool
\bool_new:N \l__siunitx_unit_powers_positive_bool
\bool_new:N \l__siunitx_unit_two_part_bool
```

(*End definition for* `\l__siunitx_unit_autofrac_bool` *and others.*)

`\l__siunitx_unit_numerator_bool` Indicates that the current unit should go into the numerator when splitting into two parts (fractions or other "sorted" styles).

```
\bool_new:N \l__siunitx_unit_numerator_bool
```

(*End definition for* `\l__siunitx_unit_numerator_bool`.)

`\l__siunitx_unit_qualifier_mode_tl` For storing the text of options which are best handled by picking function names.

```
\tl_new:N \l__siunitx_unit_qualifier_mode_tl
```

(*End definition for* `\l__siunitx_unit_qualifier_mode_tl`.)

`\l__siunitx_unit_prefix_power_bool` Used to determine if prefixes are converted into powers. Note that while this may be set as an option "higher up", at this point it is handled as an internal switch (see the two formatting interfaces for reasons).

```
\bool_new:N \l__siunitx_unit_prefix_power_bool
```

(*End definition for* `\l__siunitx_unit_prefix_power_bool`.)

`\l__siunitx_unit_prefix_fp` When converting prefixes to powers, the calculations are done as an `fp`.

```
\fp_new:N \l__siunitx_unit_prefix_fp
```

(*End definition for* `\l__siunitx_unit_prefix_fp`.)

`\l__siunitx_unit_current_tl` `\l__siunitx_unit_part_tl` Building up the (partial) formatted unit requires some token list storage. Each part of the unit combination that is recovered also has to be placed in a token list: this is a dedicated one to leave the scratch variables available.

```
\tl_new:N \l__siunitx_unit_current_tl
\tl_new:N \l__siunitx_unit_part_tl
```

(*End definition for* `\l__siunitx_unit_current_tl` *and* `\l__siunitx_unit_part_tl`.)

`\l__siunitx_unit_denominator_tl` For fraction-like units, space is needed for the denominator as well as the numerator (which is handled using `\l__siunitx_unit_formatted_tl`).

```
\tl_new:N \l__siunitx_unit_denominator_tl
```

(*End definition for* `\l__siunitx_unit_denominator_tl`.)

`\l__siunitx_unit_total_int` The formatting routine needs to know both the total number of units and the current unit. Thus an `int` is required in addition to `\l__siunitx_unit_position_int`.

```
\int_new:N \l__siunitx_unit_total_int
```

(*End definition for* `\l__siunitx_unit_total_int`.)

`\__siunitx_unit_format_parsed:` `\__siunitx_unit_format_parsed_aux:n` The main formatting routine is essentially a loop over each position, reading the various parts of the unit to build up complete unit combination.

```
\cs_new_protected:Npn \__siunitx_unit_format_parsed:
  {
    \int_set_eq:NN \l__siunitx_unit_total_int \l__siunitx_unit_position_int
    \tl_clear:N \l__siunitx_unit_denominator_tl
    \tl_clear:N \l__siunitx_unit_formatted_tl
    \fp_zero:N \l__siunitx_unit_prefix_fp
    \int_zero:N \l__siunitx_unit_position_int
    \int_do_while:nNnn
      \l__siunitx_unit_position_int < \l__siunitx_unit_total_int
      {
        \bool_set_false:N \l__siunitx_unit_bracket_bool
        \tl_clear:N \l__siunitx_unit_current_tl
        \bool_set_false:N \l__siunitx_unit_font_bool
        \bool_set_true:N \l__siunitx_unit_numerator_bool
        \int_incr:N \l__siunitx_unit_position_int
        \clist_map_inline:nn { prefix , unit , qualifier , power , special }
          { \__siunitx_unit_format_parsed_aux:n {##1} }
        \__siunitx_unit_format_output:
      }
    \__siunitx_unit_format_finalise:
  }
\cs_new_protected:Npn \__siunitx_unit_format_parsed_aux:n #1
  {
    \tl_set:Nx \l__siunitx_unit_tmp_tl
      { #1 - \int_use:N \l__siunitx_unit_position_int }
```

```
    \prop_get:NVNT \l__siunitx_unit_parsed_prop
      \l__siunitx_unit_tmp_tl \l__siunitx_unit_part_tl
      { \use:c { __siunitx_unit_format_ #1 : } }
  }
```

(*End definition for* `\__siunitx_unit_format_parsed:` *and* `\__siunitx_unit_format_parsed_aux:n`.)

`\__siunitx_unit_format_bracket:N` A quick utility function which wraps up a token list variable in brackets if they are required.

```
\cs_new:Npn \__siunitx_unit_format_bracket:N #1
  {
    \bool_if:NTF \l__siunitx_unit_bracket_bool
      {
        \exp_not:V \l__siunitx_unit_bracket_open_tl
        \exp_not:V #1
        \exp_not:V \l__siunitx_unit_bracket_close_tl
      }
      { \exp_not:V #1 }
  }
```

(*End definition for* `\__siunitx_unit_format_bracket:N`.)

`\__siunitx_unit_format_power:` `\__siunitx_unit_format_power_aux:wTF` `\__siunitx_unit_format_power_positive:` `\__siunitx_unit_format_power_negative:` `\__siunitx_unit_format_power_negative_aux:w` `\__siunitx_unit_format_power_superscript:` Formatting powers requires a test for negative numbers and depending on output format requests some adjustment to the stored value. This could be done using an `fp` function, but that would be slow compared to a dedicated if lower-level approach based on delimited arguments.

```
\cs_new_protected:Npn \__siunitx_unit_format_power:
  {
    \__siunitx_unit_format_font:
    \exp_after:wN \__siunitx_unit_format_power_aux:wTF
      \l__siunitx_unit_part_tl - \q_stop
      { \__siunitx_unit_format_power_negative: }
      { \__siunitx_unit_format_power_positive: }
  }
\cs_new:Npn \__siunitx_unit_format_power_aux:wTF #1 - #2 \q_stop
  { \tl_if_empty:nTF {#1} }
```

In the case of positive powers, there is little to do: add the power as a subscript (must be required as the parser ensures it's $\neq 1$).

```
\cs_new_protected:Npn \__siunitx_unit_format_power_positive:
  { \__siunitx_unit_format_power_superscript: }
```

Dealing with negative powers starts by flipping the switch used to track where in the final output the current part should get added to. For the case where the output is fraction-like, strip off the `-` then ensure that the result is not the trivial power 1. Assuming all is well, addition to the current unit combination goes ahead.

```
\cs_new_protected:Npn \__siunitx_unit_format_power_negative:
  {
    \bool_set_false:N \l__siunitx_unit_numerator_bool
    \bool_if:NTF \l__siunitx_unit_powers_positive_bool
      {
        \tl_set:Nx \l__siunitx_unit_part_tl
          {
            \exp_after:wN \__siunitx_unit_format_power_negative_aux:w
              \l__siunitx_unit_part_tl \q_stop
```

```
          }
        \str_if_eq_x:nnF { \exp_not:V \l__siunitx_unit_part_tl } { 1 }
          { \__siunitx_unit_format_power_superscript: }
      }
      { \__siunitx_unit_format_power_superscript: }
  }
\cs_new:Npn \__siunitx_unit_format_power_negative_aux:w - #1 \q_stop
  { \exp_not:n {#1} }
```

Adding the power as a superscript has the slight complication that there is the possibility of needing some brackets. The superscript itself uses `\sp` as that avoids any category code issues and also allows redirection at a higher level more readily.

```
\cs_new_protected:Npn \__siunitx_unit_format_power_superscript:
  {
    \tl_set:Nx \l__siunitx_unit_current_tl
      {
        \__siunitx_unit_format_bracket:N \l__siunitx_unit_current_tl
        ^ { \exp_not:V \l__siunitx_unit_part_tl }
      }
    \bool_set_false:N \l__siunitx_unit_bracket_bool
  }
```

(*End definition for* `\__siunitx_unit_format_power:` *and others.*)

`\__siunitx_unit_format_prefix:`
`\__siunitx_unit_format_prefix_power:`
`\__siunitx_unit_format_prefix_symbol:`

Formatting for prefixes depends on whether they are to be expressed as symbols or collected up to be returned as a power of 10. The latter case requires a bit of processing, which includes checking that the conversion is possible and allowing for any power that applies to the current unit.

```
\cs_new_protected:Npn \__siunitx_unit_format_prefix:
  {
    \bool_if:NTF \l__siunitx_unit_prefix_power_bool
      { \__siunitx_unit_format_prefix_power: }
      { \__siunitx_unit_format_prefix_symbol: }
  }
\cs_new_protected:Npn \__siunitx_unit_format_prefix_power:
  {
    \prop_get:NVNTF \l__siunitx_unit_prefixes_forward_prop
      \l__siunitx_unit_part_tl \l__siunitx_unit_part_tl
      {
        \tl_set:Nx \l__siunitx_unit_tmp_tl
          { power- \int_use:N \l__siunitx_unit_position_int }
        \prop_get:NVNF \l__siunitx_unit_parsed_prop
          \l__siunitx_unit_tmp_tl \l__siunitx_unit_tmp_tl
          { \tl_set:Nn \l__siunitx_unit_tmp_tl { 1 } }
        \fp_add:Nn \l__siunitx_unit_prefix_fp
          { \l__siunitx_unit_tmp_tl * \l__siunitx_unit_part_tl }
      }
      { \__siunitx_unit_format_prefix_symbol: }
  }
\cs_new_protected:Npn \__siunitx_unit_format_prefix_symbol:
  { \tl_set_eq:NN \l__siunitx_unit_current_tl \l__siunitx_unit_part_tl }
```

(*End definition for* `\__siunitx_unit_format_prefix:`, `\__siunitx_unit_format_prefix_power:`, *and* `\__siunitx_unit_format_prefix_symbol:`.)

__siunitx_unit_format_qualifier:
__siunitx_unit_format_qualifier_bracket␣:
__siunitx_unit_format_qualifier_combine:
__siunitx_unit_format_qualifier_phrase:
__siunitx_unit_format_qualifier_subscript:

There are various ways that a qualifier can be added to the output. The idea here is to modify the "base" text appropriately and then add to the current unit. In the case that a linking phrase is in use, the addition of spaces means that the unit may end up ambiguous, and brackets are therefore required *if* there is a power. Notice that when the qualifier is just treated as "text", the auxiliary is actually a no-op.

```
\cs_new_protected:Npn \__siunitx_unit_format_qualifier:
  {
    \use:c
      {
        __siunitx_unit_format_qualifier_
        \l__siunitx_unit_qualifier_mode_tl :
      }
    \tl_put_right:NV \l__siunitx_unit_current_tl \l__siunitx_unit_part_tl
  }
\cs_new_protected:Npn \__siunitx_unit_format_qualifier_bracket:
  {
    \__siunitx_unit_format_font:
    \tl_set:Nx \l__siunitx_unit_part_tl
      {
        \exp_not:V \l__siunitx_unit_bracket_open_tl
        \exp_not:N \mathrm
          { \exp_not:V \l__siunitx_unit_part_tl }
        \exp_not:V \l__siunitx_unit_bracket_close_tl
      }
  }
\cs_new_protected:Npn \__siunitx_unit_format_qualifier_combine: { }
\cs_new_protected:Npn \__siunitx_unit_format_qualifier_phrase:
  {
    \__siunitx_unit_format_font:
    \tl_set:Nx \l__siunitx_unit_part_tl
      {
        \exp_not:V \l__siunitx_unit_qualifier_phrase_tl
        \exp_not:N \mathrm
          { \exp_not:V \l__siunitx_unit_part_tl }
      }
  }
\cs_new_protected:Npn \__siunitx_unit_format_qualifier_subscript:
  {
    \__siunitx_unit_format_font:
    \tl_set:Nx \l__siunitx_unit_part_tl
      {
        \c__siunitx_unit_math_subscript_tl
          {
            \exp_not:N \mathrm
              { \exp_not:V \l__siunitx_unit_part_tl }
          }
      }
  }
```

(*End definition for* `\__siunitx_unit_format_qualifier:` *and others.*)

__siunitx_unit_format_special:

Any special odds and ends are handled by simply making the current combination into an argument for the recovered code.

```
\cs_new_protected:Npn \__siunitx_unit_format_special:
```

```
  {
    \tl_set:Nx \l__siunitx_unit_current_tl
      {
        \exp_not:V \l__siunitx_unit_part_tl
          { \exp_not:V \l__siunitx_unit_current_tl }
      }
  }
```

(*End definition for* `\__siunitx_unit_format_special:`.)

`\__siunitx_unit_format_unit:` A very simple task: add the unit to the output currently being constructed.

```
\cs_new_protected:Npn \__siunitx_unit_format_unit:
  {
    \tl_put_right:NV
      \l__siunitx_unit_current_tl \l__siunitx_unit_part_tl
  }
```

(*End definition for* `\__siunitx_unit_format_unit:`.)

`\__siunitx_unit_format_output:`
`\__siunitx_unit_format_output_aux:`
`\__siunitx_unit_format_output_denominator:`
`\__siunitx_unit_format_output_aux:nn`
`\__siunitx_unit_format_output_aux:nV`
`\__siunitx_unit_format_output_aux:nv`

The first step here is to make a choice based on whether the current part should be stored as part of the numerator or denominator of a fraction. In all cases, if the switch `\l__siunitx_unit_numerator_bool` is true then life is simple: add the current part to the numerator with a standard separator

```
\cs_new_protected:Npn \__siunitx_unit_format_output:
  {
    \__siunitx_unit_format_font:
    \bool_set_false:N \l__siunitx_unit_bracket_bool
    \use:c
      {
        __siunitx_unit_format_output_
        \bool_if:NTF \l__siunitx_unit_numerator_bool
          { aux: }
          { denominator: }
      }
  }
\cs_new_protected:Npn \__siunitx_unit_format_output_aux:
  {
    \__siunitx_unit_format_output_aux:nV { formatted }
      \l__siunitx_unit_product_tl
  }
```

There are a few things to worry about at this stage if the current part is in the denominator. Powers have already been dealt with and some formatting outcomes only need a branch at the final point of building the entire unit. That means that there are three possible outcomes here: if collecting two separate parts, add to the denominator with a product separator, or if only building one token list there may be a need to use a symbol separator. When the `repeated-symbol` option is in use there may be a need to add a leading `1` to the output in the case where the first unit is in the denominator: that can be picked up by looking for empty output in combination with the flag for using a symbol in the output but not a two-part strategy.

```
\cs_new_protected:Npn \__siunitx_unit_format_output_denominator:
  {
    \bool_if:NTF \l__siunitx_unit_two_part_bool
      {
```

```
        \bool_lazy_and:nnT
          { \l__siunitx_unit_denominator_bracket_bool }
          { ! \tl_if_empty_p:N \l__siunitx_unit_denominator_tl }
          { \bool_set_true:N \l__siunitx_unit_bracket_bool }
        \__siunitx_unit_format_output_aux:nV { denominator }
          \l__siunitx_unit_product_tl
      }
      {
        \bool_lazy_and:nnT
          { \l__siunitx_unit_per_symbol_bool }
          { \tl_if_empty_p:N \l__siunitx_unit_formatted_tl }
          { \tl_set:Nn \l__siunitx_unit_formatted_tl { 1 } }
        \__siunitx_unit_format_output_aux:nv { formatted }
          {
            l__siunitx_unit_
            \bool_if:NTF \l__siunitx_unit_per_symbol_bool
              { per_symbol }
              { product }
            _tl
          }
      }
  }
\cs_new_protected:Npn \__siunitx_unit_format_output_aux:nn #1#2
  {
    \tl_set:cx { l__siunitx_unit_ #1 _tl }
      {
         \exp_not:v { l__siunitx_unit_ #1 _tl }
         \tl_if_empty:cF { l__siunitx_unit_ #1 _tl }
           { \exp_not:n {#2} }
         \exp_not:V \l__siunitx_unit_current_tl
      }
  }
\cs_generate_variant:Nn \__siunitx_unit_format_output_aux:nn { nV , nv }
```

(*End definition for* `\__siunitx_unit_format_output:` *and others.*)

`\__siunitx_unit_format_font:` A short auxiliary which checks if the font has been applied to the main part of the output: if not, add it and set the flag.

```
\cs_new_protected:Npn \__siunitx_unit_format_font:
  {
    \bool_if:NF \l__siunitx_unit_font_bool
      {
        \tl_set:Nx \l__siunitx_unit_current_tl
          {
            \exp_not:N \mathrm
              { \exp_not:V \l__siunitx_unit_current_tl }
          }
        \bool_set_true:N \l__siunitx_unit_font_bool
      }
  }
```

(*End definition for* `\__siunitx_unit_format_font:`.)

`\__siunitx_unit_format_finalise:` `\__siunitx_unit_format_finalise_autofrac:` `\__siunitx_unit_format_finalise_fractional:` `\__siunitx_unit_format_finalise_power:` Finalising the unit format is really about picking up the cases involving fractions: these require assembly of the parts with the need to add additional material in some cases

```
\cs_new_protected:Npn \__siunitx_unit_format_finalise:
  {
    \tl_if_empty:NF \l__siunitx_unit_denominator_tl
      {
        \bool_if:NTF \l__siunitx_unit_powers_positive_bool
          { \__siunitx_unit_format_finalise_fractional: }
          { \__siunitx_unit_format_finalise_power: }
      }
  }
```

For fraction-like output, there are three possible choices and two actual styles. In all cases, if the numerator is empty then it is set here to `1`. To deal with the "auto-format" case, the two styles (fraction and symbol) are handled in auxiliaries: this allows both to be used at the same time! Beyond that, the key here is to use a single `\tl_set:Nx` to keep down the number of assignments.

```
\cs_new_protected:Npn \__siunitx_unit_format_finalise_fractional:
  {
    \tl_if_empty:NT \l__siunitx_unit_formatted_tl
      { \tl_set:Nn \l__siunitx_unit_formatted_tl { 1 } }
    \bool_if:NTF \l__siunitx_unit_autofrac_bool
      { \__siunitx_unit_format_finalise_autofrac: }
      {
        \bool_if:NTF \l__siunitx_unit_per_symbol_bool
          { \__siunitx_unit_format_finalise_symbol: }
          { \__siunitx_unit_format_finalise_fraction: }
      }
  }
```

For the "auto-selected" fraction method, the two other auxiliary functions are used to do both forms of formatting. So that everything required is available, this needs one group so that the second auxiliary receives the correct input. After that it is just a case of applying `\mathchoice` to the formatted output.

```
\cs_new_protected:Npn \__siunitx_unit_format_finalise_autofrac:
  {
    \group_begin:
      \__siunitx_unit_format_finalise_fraction:
    \exp_args:NNNV \group_end:
    \tl_set:Nn \l__siunitx_unit_tmp_tl \l__siunitx_unit_formatted_tl
    \__siunitx_unit_format_finalise_symbol:
    \tl_set:Nx \l__siunitx_unit_formatted_tl
      {
        \mathchoice
          { \exp_not:V \l__siunitx_unit_tmp_tl }
          { \exp_not:V \l__siunitx_unit_formatted_tl }
          { \exp_not:V \l__siunitx_unit_formatted_tl }
          { \exp_not:V \l__siunitx_unit_formatted_tl }
      }
  }
```

When using a fraction function the two parts are now assembled.

```
\cs_new_protected:Npn \__siunitx_unit_format_finalise_fraction:
  {
    \tl_set:Nx \l__siunitx_unit_formatted_tl
      {
        \exp_not:V \l__siunitx_unit_fraction_function_tl
```

```
          { \exp_not:V \l__siunitx_unit_formatted_tl }
          { \exp_not:V \l__siunitx_unit_denominator_tl }
      }
  }
\cs_new_protected:Npn \__siunitx_unit_format_finalise_symbol:
  {
    \tl_set:Nx \l__siunitx_unit_formatted_tl
      {
        \exp_not:V \l__siunitx_unit_formatted_tl
        \exp_not:V \l__siunitx_unit_per_symbol_tl
        \__siunitx_unit_format_bracket:N \l__siunitx_unit_denominator_tl
      }
  }
```

In the case of sorted powers, there is a test to make sure there was at least one positive power, and if so a simple join of the two parts with the appropriate product.

```
\cs_new_protected:Npn \__siunitx_unit_format_finalise_power:
  {
    \tl_if_empty:NTF \l__siunitx_unit_formatted_tl
      {
        \tl_set_eq:NN
          \l__siunitx_unit_formatted_tl
          \l__siunitx_unit_denominator_tl
      }
      {
        \tl_set:Nx \l__siunitx_unit_formatted_tl
          {
            \exp_not:V \l__siunitx_unit_formatted_tl
            \exp_not:V \l__siunitx_unit_product_tl
            \exp_not:V \l__siunitx_unit_denominator_tl
          }
      }
  }
```

(*End definition for* `\__siunitx_unit_format_finalise:` *and others.*)

## 4.8 Pre-defined unit components

Quite a number of units can be predefined: while this is a code-level module, there is little point having a unit parser which does not start off able to parse any units!

`\kilogram` `\metre` `\meter` `\mole` `\kelvin` `\candela` `\second` `\ampere`

The basic SI units: technically the correct spelling is `\metre` but US users tend to use `\meter`.

```
\siunitx_declare_unit:Nn \kilogram { \kilo \gram }
\siunitx_declare_unit:Nn \metre    { m }
\siunitx_declare_unit:Nn \meter    { \metre }
\siunitx_declare_unit:Nn \mole     { mol }
\siunitx_declare_unit:Nn \second   { s }
\siunitx_declare_unit:Nn \ampere   { A }
\siunitx_declare_unit:Nn \kelvin   { K }
\siunitx_declare_unit:Nn \candela  { cd }
```

(*End definition for* `\kilogram` *and others. These functions are documented on page* 57.)

\gram The gram is an odd unit as it is needed for the base unit kilogram.

```
\siunitx_declare_unit:Nn \gram { g }
```

(*End definition for* `\gram`. *This function is documented on page 57.*)

\yocto \zepto \atto \femto \pico \nano \micro \milli \centi \deci The various SI multiple prefixes are defined here: first the small ones.

```
\siunitx_declare_prefix:Nnn \yocto { y } { -24 }
\siunitx_declare_prefix:Nnn \zepto { z } { -21 }
\siunitx_declare_prefix:Nnn \atto  { a } { -18 }
\siunitx_declare_prefix:Nnn \femto { f } { -15 }
\siunitx_declare_prefix:Nnn \pico  { p } { -12 }
\siunitx_declare_prefix:Nnn \nano  { n } { -9 }
\siunitx_declare_prefix:Nnn \micro { [micro] } { -6 }
\siunitx_declare_prefix:Nnn \milli { m } { -3 }
\siunitx_declare_prefix:Nnn \centi { c } { -2 }
\siunitx_declare_prefix:Nnn \deci  { d } { -1 }
```

(*End definition for* `\yocto` *and others. These functions are documented on page 58.*)

\deca \deka \hecto \kilo \mega \giga \tera \peta \exa \zetta \yotta Now the large ones.

```
\siunitx_declare_prefix:Nnn \deca  { da } { 1 }
\siunitx_declare_prefix:Nnn \deka  { da } { 1 }
\siunitx_declare_prefix:Nnn \hecto { h }  { 2 }
\siunitx_declare_prefix:Nnn \kilo  { k }  { 3 }
\siunitx_declare_prefix:Nnn \mega  { M }  { 6 }
\siunitx_declare_prefix:Nnn \giga  { G }  { 9 }
\siunitx_declare_prefix:Nnn \tera  { T }  { 12 }
\siunitx_declare_prefix:Nnn \peta  { P }  { 15 }
\siunitx_declare_prefix:Nnn \exa   { E }  { 18 }
\siunitx_declare_prefix:Nnn \zetta { Z }  { 21 }
\siunitx_declare_prefix:Nnn \yotta { Y }  { 24 }
```

(*End definition for* `\deca` *and others. These functions are documented on page 58.*)

\becquerel \degreeCelsius \coulomb \farad \gray \hertz \henry \joule \katal \lumen \lux Named derived units: first half of alphabet.

```
\siunitx_declare_unit:Nn \becquerel { Bq }
\siunitx_declare_unit:Nn \degreeCelsius
  { \ensuremath { { } ^ { \circ } } \kern -\scriptspace C }
\siunitx_declare_unit:Nn \coulomb   { C }
\siunitx_declare_unit:Nn \farad     { F }
\siunitx_declare_unit:Nn \gray      { Gy }
\siunitx_declare_unit:Nn \hertz     { Hz }
\siunitx_declare_unit:Nn \henry     { H }
\siunitx_declare_unit:Nn \joule     { J }
\siunitx_declare_unit:Nn \katal     { kat }
\siunitx_declare_unit:Nn \lumen     { lm }
\siunitx_declare_unit:Nn \lux       { lx }
```

(*End definition for* `\becquerel` *and others. These functions are documented on page 58.*)

\newton \ohm \pascal \radian \siemens \sievert \steradian \tesla \volt \watt \weber Named derived units: second half of alphabet.

```
\siunitx_declare_unit:Nn \newton    { N }
\siunitx_declare_unit:Nn \ohm       { \ensuremath { \Omega } }
\siunitx_declare_unit:Nn \pascal    { Pa }
\siunitx_declare_unit:Nn \radian    { rad }
```

```
\siunitx_declare_unit:Nn \siemens   { S }
\siunitx_declare_unit:Nn \sievert   { Sv }
\siunitx_declare_unit:Nn \steradian { sr }
\siunitx_declare_unit:Nn \tesla     { T }
\siunitx_declare_unit:Nn \volt      { V }
\siunitx_declare_unit:Nn \watt      { W }
\siunitx_declare_unit:Nn \weber     { Wb }
```

(*End definition for* `\newton` *and others. These functions are documented on page 58.*)

`\day` `\hectare` `\hour` `\litre` `\liter` `\minute` `\tonne`

Non-SI, but accepted for general use. Once again there are two spellings, here for litre and with different output in this case.

```
\siunitx_declare_unit:Nn \day     { d }
\siunitx_declare_unit:Nn \hectare { ha }
\siunitx_declare_unit:Nn \hour    { h }
\siunitx_declare_unit:Nn \litre   { L }
\siunitx_declare_unit:Nn \liter   { \litre }
\siunitx_declare_unit:Nn \minute  { min }
\siunitx_declare_unit:Nn \tonne   { t }
```

(*End definition for* `\day` *and others. These functions are documented on page 58.*)

`\arcminute` `\arcsecond` `\degree`

Arc units: again, non-SI, but accepted for general use.

```
\siunitx_declare_unit:Nn \arcminute { \ensuremath { ^ { \prime } } }
\siunitx_declare_unit:Nn \arcsecond { \ensuremath { ^ { \prime \prime } } }
\siunitx_declare_unit:Nn \degree    { \ensuremath { ^ { \circ } } }
```

(*End definition for* `\arcminute`*,* `\arcsecond`*, and* `\degree`*. These functions are documented on page 59.*)

`\astronomicalunit` `\atomicmassunit` `\dalton` `\electronvolt`

A few units based on physical measurements exist: these ones are accepted for use with the International System.

```
\siunitx_declare_unit:Nn \astronomicalunit { au }
\siunitx_declare_unit:Nn \atomicmassunit   { u }
\siunitx_declare_unit:Nn \dalton           { Da }
\siunitx_declare_unit:Nn \electronvolt     { eV }
```

(*End definition for* `\astronomicalunit` *and others. These functions are documented on page 59.*)

`\nuaction` `\numass` `\nuspeed` `\nutime`

Natural units based on physical constants.

```
\siunitx_declare_unit:Nn \nuaction { \ensuremath { \mathit { \hbar } } }
\siunitx_declare_unit:Nx \numass
  {
    \exp_not:N \ensuremath
      {
        \exp_not:N \mathit { m }
          \c__siunitx_unit_math_subscript_tl { \exp_not:N \mathrm { e } }
      }
  }
\siunitx_declare_unit:Nx \nuspeed
  {
    \exp_not:N \ensuremath
      { \exp_not:N \mathit { c } \c__siunitx_unit_math_subscript_tl { 0 } }
  }
\siunitx_declare_unit:Nn \nutime
  { \numass \per \numass \per \nuspeed \squared }
```

(*End definition for* `\nuaction` *and others. These functions are documented on page 59.*)

`\auaction` `\aucharge` `\auenergy` `\aulength` `\aumass` `\autime` `\bohr` `\hartree`

Atomic units based on physical constants.

```
\siunitx_declare_unit:Nn \auaction { \ensuremath { \mathit { \hbar } } }
\siunitx_declare_unit:Nn \aucharge { \ensuremath { \mathit { e } } }
\siunitx_declare_unit:Nx \auenergy
  {
    \exp_not:N \ensuremath
      {
        \exp_not:N \mathit { E }
          \c__siunitx_unit_math_subscript_tl { \exp_not:N \mathrm { h } }
      }
  }
\siunitx_declare_unit:Nx \aulength
  {
    \exp_not:N  \ensuremath
      { \exp_not:N \mathit { a } \c__siunitx_unit_math_subscript_tl { 0 } }
  }
\siunitx_declare_unit:Nx \aumass
  {
    \exp_not:N \ensuremath
      {
        \exp_not:N \mathit { m }
          \c__siunitx_unit_math_subscript_tl { \exp_not:N \mathrm { e } }
      }
  }
\siunitx_declare_unit:Nn \autime   { \auaction \per \auenergy }
\siunitx_declare_unit:Nn \bohr     { \aulength }
\siunitx_declare_unit:Nn \hartree  { \auenergy }
```

(*End definition for* `\auaction` *and others. These functions are documented on page 59.*)

`\angstrom` `\bar` `\barn` `\bel` `\decibel` `\knot` `\millimetremercury` `\nauticalmile` `\neper`

There are then some day-to-day units which are accepted for use with SI, but are not part of the official specification.

```
\siunitx_declare_unit:Nn \angstrom          { \mbox { \AA } }
\siunitx_declare_unit:Nn \bar               { bar }
\siunitx_declare_unit:Nn \barn              { b }
\siunitx_declare_unit:Nn \bel               { B }
\siunitx_declare_unit:Nn \decibel           { \deci \bel }
\siunitx_declare_unit:Nn \knot              { kn }
\siunitx_declare_unit:Nn \millimetremercury { mmHg }
\siunitx_declare_unit:Nn \nauticalmile      { M }
\siunitx_declare_unit:Nn \neper             { Np }
```

(*End definition for* `\angstrom` *and others. These functions are documented on page 59.*)

`\dyne` `\erg` `\gal` `\gauss` `\maxwell` `\oersted` `\phot` `\poise` `\stilb` `\stokes`

CGS units: similar to the set immediately above, these may be used for specific applications.

```
\siunitx_declare_unit:Nn \dyne    { dyn  }
\siunitx_declare_unit:Nn \erg     { erg }
\siunitx_declare_unit:Nn \gal     { Gal }
\siunitx_declare_unit:Nn \gauss   { G }
\siunitx_declare_unit:Nn \maxwell { Mx }
\siunitx_declare_unit:Nn \oersted { Oe }
```

```
\siunitx_declare_unit:Nn \phot    { ph }
\siunitx_declare_unit:Nn \poise   { P }
\siunitx_declare_unit:Nn \stilb   { sb }
\siunitx_declare_unit:Nn \stokes  { St }
```

(*End definition for* `\dyne` *and others. These functions are documented on page 59.*)

`\percent` For percent, the raw character is the most flexible way of handling output.

```
\siunitx_declare_unit:Nn \percent  { \char "25 ~ }
```

(*End definition for* `\percent`*. This function is documented on page 59.*)

`\square` `\squared` `\cubic` `\cubed` Basic powers.

```
\siunitx_declare_power:NNn \square \squared { 2 }
\siunitx_declare_power:NNn \cubic  \cubed   { 3 }
```

(*End definition for* `\square` *and others. These functions are documented on page 59.*)

## 4.9 Messages

```
\msg_new:nnnn { siunitx } { unit / dangling-part }
  { Found~#1~part~with~no~unit. }
  {
    Each~#1~part~must~be~associated~with~a~unit:~a~#1~part~was~found~
    but~no~following~unit~was~given.
  }
\msg_new:nnnn { siunitx } { unit / duplicate-part }
  { Duplicate~#1~part:~#2. }
  {
    Each~unit~may~have~only~one~#1:\\
    the~additional~#1~part~'#2'~will~be~ignored.
  }
\msg_new:nnnn { siunitx } { unit / duplicate-sticky-per }
  { Duplicate~\token_to_str:N \per. }
  {
    When~the~'sticky-per'~option~is~active,~only~one~
    \token_to_str:N \per \  may~appear~in~a~unit.
  }
\msg_new:nnnn { siunitx } { unit / part-before-unit }
  { Found~#1~part~before~first~unit:~#2. }
  {
    The~#1~part~'#2'~must~follow~after~a~unit:~
    it~cannot~appear~before~any~units~and~will~therefore~be~ignored.
  }
```

## 4.10 Standard settings for module options

Some of these follow naturally from the point of definition (*e.g.* boolean variables are always `false` to begin with), but for clarity everything is set here.

```
\keys_set:nn { siunitx }
  {
    bracket-denominator = true      ,
    fraction-command    = \frac     ,
    parse-units         = true      ,
    per-mode            = power     ,
```

```
    per-symbol          = /         ,
    qualifier-mode      = subscript ,
    qualifier-phrase    = \         ,
    sticky-per          = false     ,
    unit-close-bracket  = )         , % (
    unit-open-bracket   = (         , % )
    unit-product        = \,
  }
```

⟨/package⟩

# References


[1] *The International System of Units (SI)*, https://www.bipm.org/en/measurement-units/.

[2] *SI base units*, https://www.bipm.org/en/publications/si-brochure/section2-1.html.

[3] *Units with special names and symbols; units that incorporate special names and symbols*, https://www.bipm.org/en/publications/si-brochure/section2-2-2.html.

[4] *SI Prefixes*, https://www.bipm.org/en/publications/si-brochure/chapter3.html.

[5] *Stating values of dimensionless quantities, or quantities of dimension one*, https://www.bipm.org/en/publications/si-brochure/section5-3-7.html.

[6] *Non-SI units accepted for use with the International System of Units*, https://www.bipm.org/en/publications/si-brochure/table6.html.

[7] *Non-SI units whose values in SI units must be obtained experimentally*, https://www.bipm.org/en/publications/si-brochure/table7.html.

[8] *Other non-SI units*, https://www.bipm.org/en/publications/si-brochure/table8.html.

[9] *Non-SI units associated with the CGS and the CGS-Gaussian system of units*, https://www.bipm.org/en/publications/si-brochure/table9.html.

# Part VII
# siunitx-v1 – Version 1 compatibility

## 1 siunitx-v1 implementation

Start the DocStrip guards.

```
1 ⟨*cfg⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention). In contrast to other parts of the bundle, the functions here may need to redefine those from various submodules.

```
2 ⟨@@=siunitx⟩
3 ⟨/cfg⟩
```

# Part VIII
# siunitx-v2 – Version 2 compatibility

## 1 siunitx-v2 implementation

Start the DocStrip guards.

```
1 ⟨*cfg⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention). In contrast to other parts of the bundle, the functions here may need to redefine those from various submodules.

```
2 ⟨@@=siunitx⟩
```

The old `s` column type is handled by using the functionality of collcell.

```
3 \RequirePackage { collcell }
4 \AtBeginDocument
5   {
6     \__siunitx_declare_column:Nnn s
7       { \collectcell \unit }
8       { \endcollectcell }
9   }
10 ⟨/cfg⟩
```

# 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.

## D



## E

## N



## O



## P



## Q

## R



## S

## T

## U



## V



## W



## Y



## Z