The **namedef** package
Named parameters in TEX definitions*

Phelype H. Oleinik†

Released 2020-06-20

1 Introduction

This package provides a somewhat dubious, however useful way to make TEX definitions using `\def` and the like. The usual way to define a "hello world" macro is `\def\hello#1{Hello #1!}`. Sure, this is easy (for people using this package, at least), but sometimes I stumbled upon the case where I had a macro defined with something like `\def\macro#1#2#3#4#5#6{#6#1#4#3#5#2}` and then I had to, for whatever reason, add another argument before `#1`. Oh, the pain. But OK, occupational hazard. And then I change my mind and needed to revert. This package provides a way to give a semantic meaning to the arguments of a definition such that it becomes clearer in meaning and easier to change when an argument's identifier defines what it means rather than its position in a list.

1.1 Features

This package defines a macro, `\named`, which acts like a prefix for `\def` (the same way as `\long` and the like), and allows one to rewrite the `\hello` macro like this: `\named\def\hello#[who]{Hello #[who]!}`. Sure, a macro with one argument won't see much benefit from it, but as soon as that number increases, a better description of the arguments comes in handy.

The package also defines a macro `\NamedDelim` which allows the user to change the delimiter of the named parameters to their liking. For example, after `\NamedDelim //` the example above changes to: `\named\def\hello#/who/{Hello #/who/!}`. The default is `\NamedDelim []`.

The only dependency of this package is the LATEX3 programming layer, expl3, thus this package works in any format and engine supported by expl3. In LATEX 2$_\varepsilon$, load it with `\usepackage{namedef}`, and in other formats with `\input namedef.sty`.

1.2 Feedback

Bugs, feature requests, or suggestions are most welcome. You can send them by e-mail or to the source code repository at `https://github.com/PhelypeOleinik/namedef`.

---

[*] This file has version number 1.0, last revised 2020-06-20.

[†] E-mail: phelype.oleinik⟨*at*⟩latex-project.org

2 Usage

`\named`

`\named`⟨*other prefixes*⟩⟨`\(e,g,x)def`⟩⟨*parameter text*⟩`{`⟨*replacement text*⟩`}`

The `\named` macro will grab the ⟨*other prefixes*⟩, the `\def` command, the ⟨*parameter text*⟩ and ⟨*replacement text*⟩, and will replace every ocurrence of `#[`⟨*text*⟩`]` by a suitable `#`⟨*number*⟩ for TEX to carry on with the definition.

The ⟨*other prefixes*⟩ can be any combination of `\long`, `\outer`, `\global`, and `\protected`. As for TEX, their relative order doesn't matter, except that `\named` must appear *before* any `\global` prefix. Other prefixes are detected whether placed before or after `\named`.

The `\def` command can be anything. The package will not check the validity of the command, it will simply drag the token along until the point where the definition is made. This allows one to use any of TEX's primitive `\def` commands as well as, for instance, expl3's `\cs_new:Npn` or the like. However `\named` will not work with LATEX's `\newcommand` or xparse's `\NewDocumentCommand` and the like because their syntax is completely different.

The ⟨*parameter text*⟩ is the same ⟨*parameter text*⟩ that would be used if the `\named` prefix wasn't used with the difference that all `#`⟨*number*⟩ must be replaced by a `#[`⟨*name*⟩`]`. The characters `[` and `]` are mandatory and the character `]` cannot appear in ⟨*name*⟩, however `[` and `]` can still appear in the ⟨*parameter text*⟩ without restriction; the special meaning is only applied after a parameter token (`#`). ⟨*name*⟩ can be anything that when fully expanded returns a string of characters (anything valid in `\csname`...`\endcsname`). The ⟨*parameter text*⟩ cannot, however, contain numbered parameters anymore (`#1`, `#2`, ...).

The ⟨*replacement text*⟩ is also the same ⟨*replacement text*⟩ that would be used otherwise, with all `#`⟨*number*⟩ replaced by `#[`⟨*name*⟩`]`.

`\NamedDelim`
`\globalNamedDelim`

`\NamedDelim`⟨*begin-token*⟩⟨*end-token*⟩
`\globalNamedDelim`⟨*begin-token*⟩⟨*end-token*⟩

These macros change the delimiter of the named parameters from the default `#[`⟨*name*⟩`]` to `#`⟨*begin-token*⟩⟨*name*⟩⟨*end-token*⟩. Both delimiters must be one single non-macro token. Valid delimiters are character tokens with catcode 3, 4, 7, 8, 11, 12, or 13 (see section 2.2).

`\globalNamedDelim` is the same as `\NamedDelim` except that the effect of the former has global scope, while the latter is local to a group. While you can use both, you should be careful not to interleave them too much or you might exhaust TEX's save stack.

Delimiters are matched based on their character code (using `\if`) so changes in their category code doesn't matter as long as that change doesn't prevent the character from becoming a token or the category code isn't "too special" :-) (see above).

The choice of delimiter is mostly "what you like better". Neither the delimter tokens nor the name of the parameter make it to the macro itself. They are just code-level abstractions. Thus the delimiter should be something that the person writing the code can easily distinguish from the rest of the code. For example, the code `\NamedDelim xz \named\def\test#xyz{xyz#xyzxyz}` works, but its readability is questionable.

2.1 Limitations

As already stated the command does not work (and I don't intend to make it work) with LATEX 2ε's `\newcommand` and its family because a) the argument specification is by the number of arguments, so you can't "declare" them as with `\def`, and b) because it's supposed to be used for user-level interfaces, which usually (and preferably) have a low argument count, so numbering shouldn't be a problem. That said, see section 4.1.

For `xparse`'s `\NewDocumentCommand` the situation is the same. Other than these, `\named` should work for whatever type of definition that uses TeX's `\def` syntax.

Another limitation that I'd like to change in a future release (but still don't know the best way to make the interface) is to support definition commands which go beyond the ⟨*parameter text*⟩`{`⟨*replacement text*⟩`}` syntax. For instance, in `expl3` a conditional that checks whether its argument is negative can be defined like this (for integers, of course):

```
\prg_new_conditional:Npnn \namedef_if_negative:n #1 { T, F, TF }
  {
    \if_int_compare:w #1 < 0
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

And if one tried to use `\named` in that definition it would fail because this command takes one extra argument before the ⟨*replacement text*⟩. Something could be done to allow one (or more) extra argument before the ⟨*replacement text*⟩.

One serious limitation is when used with definitions that expand their argument, namely `\edef` and `\xdef`. This type of definition expands tokens as it reads them, possibly changing their meaning during the process. `\named`, however, first grabs the definition as argument to process the named arguments before actually performing the definition, so these "unexpected" changes of meaning might make the code misbehave. While writing this manual I could think of two (and a half) situations which will be problematic and how to work around them (sorry, no solution for now; suggestions are welcome :).

2.1.1 \named\edef\test{\string#[arg]}

The normal (no `\named`) counterpart of this one is a perfectly valid definition: `\edef\test{\string#1}`. While expanding the ⟨*replacement text*⟩, `\string` turns the parameter token $\#_6$ into a character $\#_{12}$, thus defining `\test` to expand to the two characters `#1`. When using `\named`, however, the replacement routine doesn't know what `\string` does to the token next to it, so it goes on and treats `#[arg]` as one named argument only to find out that it was never defined in the ⟨*parameter text*⟩, so it aborts the definition with an error.

This will occur in the specific case where the user wishes to have the macro expand to the characters `#[arg]`, without replacement by a parameter. In this case the work-around is to temporarily switch the delimiter tokens of `\named`'s scanner:

```
\NamedDelim||
\named\edef\test{\string#[arg]}
\NamedDelim[]
```

in which case the scanner will still see the `#` as a parameter token but since it is no longer followed by a delimiter, it will be simply passed on to the definition. Afterwards, at the time TeX tries to carry on with the definition, `\string` will do its thing to `#` without further problems.

2.1.2 `\named\edef\test#[arg]{\string}#[arg]}`

This one, as the previous, works fine without `\named`: `\edef\test#1{\string}#1}`. Again, when TeX scans this definition, it will expand `\string` which will turn the end group token `}`$_2$ into a character `}`$_{12}$, which will have TeX end the definition with the next `}`$_2$, after the `#1`. This only works because TeX does not grab the whole thing as argument before expanding. Which is precisely what `\named` does.

When `\named` grabs the definition as argument the first `}` ends the ⟨*replacement text*⟩, so what `\named` effectively sees is `\edef\test#[arg]{\string}`, which is then processed (`#[arg]` is replaced by `#1`) and left back in the input stream for TeX to do its thing, however the replacement `#[arg]` is never replaced: `\edef\test#1{\string}#[arg]}`, then when TeX tries to do the definition it will complain about an "! Illegal parameter number in definition of `\test`."

The work-around in this case is to do a dirty brace trick to lure `\named` into grabbing the whole thing as argument, but which will disappear at the time TeX performs the expansion and definition. One option is `\iffalse{\fi`:

```
\named\edef\test#[arg]{%
  \iffalse{\fi \string}#[arg]}
```

In this example `\named` will process everything, but at the time of the definition the entire `\iffalse{\fi` thing expands to nothing and the final definition becomes the same as the one without `\named`. The `\iffalse{\fi` block can also be left in the definition *without* named and the result will be the same. One could argue that using the brace hack is safer because it doesn't change the definition, yet avoid problems when grabbing the definition as argument in a general situation.

2.1.2,5 `\named\edef\test#[arg]{\string{#[arg]}`

This is rather similar to the previous one, except that the brace later-to-be-detokenized begins a group: `\edef\test#1{\string{#1}`. Here TeX also expands `\string` and makes `{`$_1$ a `{`$_{12}$ which does not count in the brace balancing. `\named`, however, will count it when grabbing the definition as argument and will require one more `}`$_2$. If the code is run as is TeX will probably report a "File ended while scanning use of . . . " error unless there happens to be a matching `}`$_2$ somewhere else, in which case the definition will just go wrong. The work-around here is the same as the one before, with `}` instead:

```
\named\edef\test#[arg]{%
  \string{#[arg]\iffalse}\fi}
```

This will ensure that `\named` will see the `}` it needs to grab the definition correctly and will disappear once the definition is done.

2.2 Invalid delimiters

The delimiters that can be used should be character tokens with catcode 3, 4, 7, 8, 11, 12, or 13. Characters with catcode 0, 5, 9, 14, and 15 don't produce tokens to start with, so they can't possibly be used. The remaining category codes are currently disallowed in the code because they make the input ambiguous or because they make the implementation more complex with no real advantage.

Catcodes 1 and 2 (begin-/end-group) cannot be used because they become indistinguishable from the braces that delimit the ⟨*parameter text*⟩ of the definition, so the input is ambiguous.

Catcode 6 (macro parameter) cannot be used because it gets hard to distinguish a named parameter from some text surrounded by parameter tokens. For example in: `\named\edef\foo#name#{\string#then#name#}`, `namedef` would raise an error on `#then#` (unknown parameter) without knowing that the first `#`$_6$ becomes `#`$_{12}$ and the actual parameter is `#name#`… Or is it? I'm not entirely convinced of my own argument, so this might be implemented in the future.

Catcode 10 (blank space) is possible but it requires a hanful of extra precautions to avoid losing the space when passing arguments around. Since it makes for a strange-looking syntax (our eyes are trained to ignore spaces), this is not supported.

3 Boring examples

The following examples show two definitions each, which are the same, but the second uses `\named`. The third line in each example shows the `\meaning` of the defined macro.

First the basics, replacing a numbered parameter by a named one:

Basics

```
      \def\hello#1{Hello #1!}
\named\def\hello#[who]{Hello #[who]!}
> \hello=macro:#1->Hello #1!
```

Prefixes can be added at will after the `\named` prefix:

Prefixes

```
      \protected\long\edef\hello#1{Hello #1!}
\named\protected\long\edef\hello#[who]{Hello #[who]!}
> \hello=\protected\long macro:#1->Hello #1!
```

This example is just to show that the named argument delimiter doesn't interfere with the text in the macro:

Argument delimited by `[` and `]`

```
      \def\hello[#1]{Hello #1!}
\named\def\hello[#[who]]{Hello #[who]!}
> \hello=macro:[#1]->Hello #1!
```

However, for readability, the delimiter can be changed to something else:

Argument delimited by `[` and `]`

```
\NamedDelim{|}{|}
      \def\hello[#1]{Hello #1!}
\named\def\hello[#|who|]{Hello #|who|!}
> \hello=macro:[#1]->Hello #1!
```

This example demonstrates multiple arguments and arbitrary ⟨*parameter text*⟩:

More arguments

```
      \def\cfbox #1 to #2#3{\fbox{\hbox to #2{\color{#1}#3}}}
\named\def\cfbox #[color] to #[width]#[content]{%
        \fbox{\hbox to #[width]{\color{#[color]}#[content]}}}
> \cfbox=macro:#1 to #2#3-> \fbox {\hbox to #2{\color {#1}#3}}
```

TeX's weird `#{` argument can be used as well:

Weird arguments

```
      \def\cbox #1 to #2#{\hbox to #2\bgroup\color{#1}\let\next= }
\named\def\cbox #[color] to #[width]#{%
        \hbox to #[width]\bgroup\color{#[color]}\let\next= }
> \cbox=macro:#1 to #2{->\hbox to #2\bgroup \color {#1}\let \next = {
```

`\edef` workaround 2.1.1

```
      \edef\test{\string#[arg]}
\NamedDelim ||
\named\edef\test{\string#[arg]}
\NamedDelim []
> \test=macro:->#[arg]
```

`\edef` workaround 2.1.2

```
      \edef\test#1{\string}#1}
\named\edef\test#[arg]{\iffalse{\fi\string}#[arg]}
> \test=macro:#1->}#1
```

`\edef` workaround 2.1.2,5

```
      \edef\test#1{\string{#1}
\named\edef\test#[arg]{\string{#[arg]\iffalse}\fi}
> \test=macro:#1->{#1
```

4 Interesting[1] examples

These examples shows a few more elaborate ways to use namedef.

4.1 Extended `\newcommand`

Here's an implementation to allow the syntax of namedef in `\newcommand`. It uses xparse to handle optional arguments, and uses `\newcommand` itself to define (possibly) optional argument handling, so the resulting command uses LaTeX 2$_\varepsilon$'s command parsing machinery.

The syntax of the defined command is:

`\newnamedcommand`⟨*⟩`\`⟨*cmd*⟩`[`⟨*arg-list*⟩`][`⟨*opt*⟩=⟨*default*⟩`]{`⟨*definition*⟩`}`

where everything is the same as in regular `\newcommand`, except for the optional arguments ⟨*arg-list*⟩ and ⟨*opt*⟩=⟨*default*⟩. ⟨*arg-list*⟩ should be a comma-separated list of named parameters as `\named` would expect, and ⟨*opt*⟩ is a named parameter, and ⟨*default*⟩ is its default value. The usage would be something like:

```
\newnamedcommand\foo[#[one],#[two]][#[opt]=default]%
  {#[one], #[two], and #[opt]}
```

which translates to:

```
\newcommand\foo[3][default]%
  {#2, #3, and #1}
```

---

[1] Terms and Conditions may apply.

First, load xparse and namedef, and define the top-level commands to use a common `\NNC_newcommand:NnNnnn`. `\NNC_newcommand:NnNnnn` will store the mandatory arguments in a `seq` variable for easier access, then call `\__NNC_newcommand:NnNn` to do the `\newcommand` part of the job, and `\__NNC_named_def:nNnn` to `\named` part. `\new...`, `\renew...`, and `\provide...` versions are defined, but since a `\def` is used later with no checking, the behaviour is not exactly the same as you'd get with `\newcommand` in this regard.

```
\usepackage{namedef}
\usepackage{xparse}
\ExplSyntaxOn
\seq_new:N \l__NNC_args_seq
\scan_new:N \s__NNC
\NewDocumentCommand \newnamedcommand { s m o o m }
  { \NNC_newcommand:NnNnnn \newcommand {#1} #2 {#3} {#4} {#5} }
\NewDocumentCommand \renewnamedcommand { s m o o m }
  { \NNC_newcommand:NnNnnn \renewcommand {#1} #2 {#3} {#4} {#5} }
\NewDocumentCommand \providenamedcommand { s m o o m }
  { \NNC_newcommand:NnNnnn \providecommand {#1} #2 {#3} {#4} {#5} }
\named \cs_new_protected:Npn \NNC_newcommand:NnNnnn
    #[newcmd] #[star] #[cmd] #[args] #[opt] #[defn]
  {
    \seq_clear:N \l__NNC_args_seq
    \IfValueT {#[args]}
      { \seq_set_from_clist:Nn \l__NNC_args_seq {#[args]} }
    \__NNC_newcommand:NnNn #[newcmd] {#[star]} #[cmd] {#[opt]}
    \__NNC_named_def:nNnn {#[star]} #[cmd] {#[opt]} {#[defn]}
  }
```

`\__NNC_newcommand:NnNn` does the `\newcommand` part of the job. It takes the arguments read in by xparse, and translates them into the `\newcommand` syntax. The number of items in the `#[args]` parameter is counted and left within brackets, and the default value of the optional argument is also left within another pair of brackets. This step is executed with an empty definition because the named parameters will cause havoc in `\newcommand`.

```
\named \cs_new_protected:Npn \__NNC_newcommand:NnNn
    #[newcmd] #[star] #[cmd] #[opt]
  {
    \use:x
      {
        \exp_not:N #[newcmd]
        \IfBooleanT {#[star]} { * }
        \exp_not:N #[cmd]
        \seq_if_empty:NF \l__NNC_args_seq
          { [ \seq_count:N \l__NNC_args_seq ] }
        \IfValueT {#[opt]} { [ \__NNC_opt_value:w #[opt] \s__NNC ] }
          { }
      }
  }
\cs_new:Npn \__NNC_opt_value:w #1 = #2 \s__NNC {#2}
```

Now `\__NNC_named_def:nNnn` will do the `\named` part. First the `#[star]` argument (if not present) becomes `\long`, and then comes `\named` and `\def`. Then, if an optional argument was given, the command we need to define is `\\foo` rather than `\foo`, so use take care of that with `\token_to_str:N`, and then leave the named parameter given for the optional argument within brackets. If there's no optional argument, we just define `#[cmd]` (pretty boring). Then we call `\seq_use:Nn` on the mandatory arguments to lay them flat for `\named`, and then the parameter text (`#[defn]`), unexpanded.

```
\named \cs_new_protected:Npn \__NNC_named_def:nNnn
    #[star] #[cmd] #[opt] #[defn]
  {
    \use:x
      {
        \IfBooleanF {#[star]} { \long }
        \named \def
        \IfValueTF {#[opt]}
          {
            \exp_not:c { \token_to_str:N #[cmd] }
              [ \exp_not:o { \__NNC_opt_name:w #[opt] \s__NNC } ]
          }
          { \exp_not:N #[cmd] }
        \seq_use:Nn \l__NNC_args_seq { }
        { \exp_not:n {#[defn]} }
      }
  }
\cs_new:Npn \__NNC_opt_name:w #1 = #2 \s__NNC {#1}
\ExplSyntaxOff
```

5 namedef Implementation

```
⟨*package⟩
⟨@@=namedef⟩
```

5.1 Loading

`\__namedef_end_package_hook:`

Load expl3, either through `\RequirePackage` or through inputting the generic loader, depending on the format in use (copied from Bruno Le Floch's gtl).

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname RequirePackage\endcsname\relax
  \input expl3-generic.tex
\else
  \RequirePackage{expl3}[2018-05-15]
\fi
\ExplSyntaxOn
\cs_if_exist:NTF \ProvidesExplPackage
  {
    \cs_new_eq:NN \__namedef_end_package_hook: \prg_do_nothing:
```

```
    \ExplSyntaxOff
    \ProvidesExplPackage
  }
  {
    \cs_new_eq:NN \__namedef_end_package_hook: \ExplSyntaxOff
    \group_begin:
    \ExplSyntaxOff
    \cs_set_protected:Npn \__namedef_tmp:w #1#2#3#4
      {
        \group_end:
        \tl_gset:cx { ver @ #1 . sty } { #2 ~ v#3 ~ #4 }
        \cs_if_exist_use:NF \wlog { \iow_log:x }
          { Package: ~ #1 ~ #2 ~ v#3 ~ #4 }
      }
    \__namedef_tmp:w
  }
      {namedef} {\namedefDate} {\namedefVersion}
      {Named parameters in TeX definitions (PHO)}
```

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

5.2 Declarations

`\flag␣__namedef_parm_count`

A flag (mis)used as a counter to keep track of the parameter number.

```
\flag_new:n { __namedef_parm_count }
```

(*End definition for* `\flag __namedef_parm_count`.)

`\c__namedef_prefix_tl`

A prefix to use as name space for temporary macros.

```
\tl_const:Nn \c__namedef_prefix_tl { namedef~parm~-> }
```

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

`\l__namedef_macro_tl`

A token list to store the name of the macro meing defined for error messages.

```
\tl_new:N \l__namedef_macro_tl
```

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

`\q__namedef_mark`
`\q__namedef_stop`

Quarks used throughout the package.

```
\quark_new:N \q__namedef_mark
\quark_new:N \q__namedef_stop
```

(*End definition for* `\q__namedef_mark` *and* `\q__namedef_stop`.)

`\s__namedef`

Scan mark used to skip code.

```
\scan_new:N \s__namedef
```

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

`\__namedef_skip_to_scan_mark:w`
`\__namedef_skip_to_scan_mark:nw`

Consume everything up to `\s__namedef`.

```
\cs_new:Npn \__namedef_skip_to_scan_mark:w  #1    \s__namedef {  }
\cs_new:Npn \__namedef_skip_to_scan_mark:nw #1 #2 \s__namedef {#1}
```

(*End definition for* `\__namedef_skip_to_scan_mark:w` *and* `\__namedef_skip_to_scan_mark:nw`.)

`\__namedef_tmp:w`

A scratch macro.

```
\cs_new_eq:NN \__namedef_tmp:w ?
```

(*End definition for* `\__namedef_tmp:w`.)

5.3 The top-level `\named` macro

`\named`
`\__namedef_grab_prefix:nN`

Starts scanning ahead for prefixes and the definition command. Once finished the scanning of prefixes, call `\__namedef_replace_named:nNnn` to do the heavy lifting.

```
\cs_new_protected:Npn \named { \__namedef_grab_prefix:nN { } }
\cs_new_protected:Npn \__namedef_grab_prefix:nN #1 #2
  {
    \__namedef_if_prefix:NTF #2
      { \__namedef_grab_prefix:nN }
      { \__namedef_detect_prefixes:Nn \__namedef_kill_outer:nN }
    { #1#2 }
  }
```

(*End definition for* `\named` *and* `\__namedef_grab_prefix:nN`. *This function is documented on page 2.*)

`\__namedef_if_prefix_p:N`
`\__namedef_if_prefix:NTF`

Checks against a list of valid prefixes and returns `true` or `false` accordingly.

```
\prg_new_conditional:Npnn \__namedef_if_prefix:N #1 { TF }
  {
    \if_int_compare:w 0
        \if_meaning:w \tex_protected:D #1 1 \fi:
        \if_meaning:w \tex_global:D    #1 1 \fi:
        \if_meaning:w \tex_outer:D     #1 1 \fi:
        \if_meaning:w \tex_long:D      #1 1 \fi:
        \if_meaning:w \scan_stop:      #1 1 \fi:
        = 1 \exp_stop_f:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\__namedef_if_prefix:NTF`.)

`\__namedef_detect_prefixes:Nn`
`\__namedef_extract_prefixes:w`
`\__namedef_extract_protected:n`
`\__namedef_extract_protected_aux:w`
`\__namedef_extract_long:n`
`\__namedef_extract_long_aux:w`
`\__namedef_extract_outer:n`
`\__namedef_extract_outer_aux:w`

Defines a scratch macro `\__namedef_tmp:w` and queries its prefixes, then forwards them to the the next macro to perform the parameter replacement and definition.

This code would be quite a lot simpler if `\outer` didn't exist. First extract the meaning of `\__namedef_tmp:w`, and pass the prefixes (before "`macro:`") to `\__namedef_extract_prefixes:w`, and then to `\__namedef_extract_protected:n`, `\__namedef_extract_long:n`, and `\__namedef_extract_outer:n` in turn to check if each of these prefixes is there.

`\global` can't be checked this way because it's different from other prefixes in the sense that it affects the definition *at the time* of the definition, rather than at the time it is used. I don't know if it's possible to detect a `\global` after it's already consumed by TeX.

```
\cs_new_protected:Npn \__namedef_detect_prefixes:Nn #1 #2
  {
    \cs_set_nopar:Npn \__namedef_tmp:w { }
    \use:x
      {
        \exp_not:N #1
          {
            \exp_after:wN \exp_after:wN
              \exp_after:wN \__namedef_extract_prefixes:w
```

```
                \exp_after:wN \token_to_meaning:N
                  \cs:w __namedef_tmp:w \cs_end: \s__namedef
            \exp_not:n {#2}
          }
      }
  }
\use:x
  {
    \cs_new:Npn \exp_not:N \__namedef_extract_prefixes:w ##1
        \tl_to_str:n { macro: } ##2 \s__namedef
      {
        \exp_not:N \__namedef_extract_protected:n {##1}
        \exp_not:N \__namedef_extract_long:n {##1}
        \exp_not:N \__namedef_extract_outer:n {##1}
      }
  }
\cs_set_protected:Npn \__namedef_tmp:w #1 #2
  {
    \use:x
      {
        \cs_new:cpn { __namedef_extract_#1:n } ####1
          {
            \exp_not:c { __namedef_extract_#1_aux:w } ####1
              \token_to_str:N #2 \scan_stop: \token_to_str:N #2 \s__namedef
          }
        \cs_new:cpn { __namedef_extract_#1_aux:w } ####1
            \token_to_str:N #2 ####2 \token_to_str:N #2 ####3 \s__namedef
          {
            \exp_not:N \if_meaning:w \scan_stop: ####2
            \exp_not:N \else:
              \exp_not:c { tex_#1:D }
            \exp_not:N \fi:
          }
      }
  }
\__namedef_tmp:w { protected } { \protected }
\__namedef_tmp:w { long } { \long }
\__namedef_tmp:w { outer } { \outer }
```

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

5.4 Main routine

`\__namedef_kill_outer:nN`
`\__namedef_start:nNp`
`\__namedef_replace_named:nNnn`
`\__namedef_replace_parameter:Nn`
`\__namedef_parameter_output:nnw`
`\__namedef_handle_parameter:nN`
`\__namedef_define:nnnN`

Here we play dirty: abuse the fact that `\exp_not:N` temporarily makes the `\noexpand`ed control sequence temporarily equal to `\relax`. But rather than using it in an `\edef` or whatnot, hit `\exp_not:N` with `\exp_after:wN`, and then quickly grab it with `\__namedef_start:nNp`, so it's safe to grab it, even if it's `\outer`. If that wasn't bad enough, do it once again to make it equal to `\relax` in the scope of `\__namedef_replace_named:nNnn` so that it doesn't blow up.

```
\cs_new_protected:Npn \__namedef_kill_outer:nN #1
  {
    \cs_set:Npn \__namedef_tmp:w { \__namedef_start:nNp {#1} }
    \exp_after:wN \__namedef_tmp:w \exp_not:N
  }
\cs_new_protected:Npn \__namedef_start:nNp #1 #2 #3 #
  {
    \group_begin:
      \int_set:Nn \tex_escapechar:D { `\\ }
      \exp_after:wN \cs_set_eq:NN \exp_not:N #2 \scan_stop:
      \__namedef_replace_named:nNnn {#1} #2 {#3}
  }
```

Here the actual replacement of named parameters by numbered ones takes place. A group is started to revert the flag and all the defined temporary macros.

`\__namedef_replace_parameter:Nn \__namedef_in_parameter:nN` starts replacing a dummy macro in the generic parameter replacement routine by the macro which counts the parameters and aliases the named parameters with numbered ones. Finally it starts `\__namedef_replace_parm:Nn`, which scans the ⟨*parameter text*⟩ for the named parameters and replaces them by numbered ones. The second output argument of `\__namedef_replace_parm:Nn` is a list of definitions which assign a number to each named parameter so that they can be used in the next step.

`\__namedef_replace_parameter:Nn \__namedef_in_replacement:nN` then starts by replacing the same dummy macro by one which will replace the named parameter by its number. Again `\__namedef_replace_parm:Nn` is started, and its output is the already-processed part of the ⟨*replacement text*⟩.

The output of both steps is inserted after `\__namedef_define:nnnN` (it's missing two arguments in the definition of `\__namedef_replace_named:nNnn`). After all that is done, all the named parameters were replaced by numbered ones, so `\__namedef_define:nnnN` can do its thing.

A final quark is put in the input stream for recovery from errors. In a successful run this quark is removed by `\__namedef_define:nnnN`.

```
\cs_new_protected:Npn \__namedef_replace_named:nNnn #1 #2 #3 #4
  {
      \tl_set:Nx \l__namedef_macro_tl { \token_to_str:N #2 }
      \__namedef_replace_parameter:Nn \__namedef_in_parameter:nN {#3}
      \__namedef_replace_parameter:Nn \__namedef_in_replacement:nN {#4}
      \__namedef_define:nnnN     {#1} #2
      \s__namedef
  }
\cs_new_protected:Npn \__namedef_define:nnnN #1 #2 #3 #4
  {
    \group_end:
    #3#4#2{#1}
  }
\cs_new_protected:Npn \__namedef_replace_parameter:Nn #1 #2
  {
    \cs_set_eq:NN \__namedef_handle_parameter:nN #1
    \__namedef_replace_parm:Nn \__namedef_parameter_output:nnw {#2}
  }
\cs_new_eq:NN \__namedef_handle_parameter:nN ?
\cs_new_protected:Npn \__namedef_parameter_output:nnw #1 #2
    #3 \__namedef_define:nnnN
  { #2 #3 \__namedef_define:nnnN {#1} }
```

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

`\__namedef_in_parameter:nN`
`\__namedef_in_replacement:nN`

These two functions handle the named parameters when they are found in the ⟨*parameter text*⟩ and ⟨*replacement text*⟩, respectively.

`\__namedef_in_parameter:nN` checks if the named parameter already exists (with `\relax` being `true`) and, in such case, throws an error and inserts the number already set for that named parameter. Otherwise the parameter is `\let` to `\relax` so that if it is found later an error is issued. Setting a macro to `\relax` is an expandable way to define it (the same approach as in l3flag). After that, the `\flag __namedef_parm_count` is raised once and its height is used as the parameter number. The current parameter tokens and the parameter number are flushed to the first (left) output slot, and a definition `\cs_set:cpn {\c__namedef_prefix_tl` ⟨*name*⟩`} {`⟨*number*⟩`}` is appended to the second (right) output slot so that the names can be used in the ⟨*replacement text*⟩.

In case of a repeated parameter it is tricky to do anything sensible. In a normal definition, when TEX sees a repeated parameter number (like in `\def\foo#1#1{}`) it just uses the wrong number to a parameter number not yet taken, or ignores the parameter if there's already nine before that. However here we can't guess the name of the next parameter, so we can't do much. The easiest way out is to just use the same parameter number as before and go on with out job: at the end, TEX will complain again about this.

```
\cs_new:Npn \__namedef_in_parameter:nN #1
  {
    \if_cs_exist:w \c__namedef_prefix_tl #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      { \msg_expandable_error:nnn { namedef } { repeated-parm } {#1} }
      {
        \exp_after:wN \use_none:n \cs:w \c__namedef_prefix_tl #1 \cs_end:
        \flag_raise:n { __namedef_parm_count }
      }
    \exp_args:Nf \__namedef_append_output:nnNwnn
      { \flag_height:n { __namedef_parm_count } }
      {#1}
  }
```

`\__namedef_in_replacement:nN` also checks if the named parameter exists, however now it will be *not* be `\relax`, but the number defined earlier, so `\cs_if_exist:cTF` can be safely used. If the parameter does not exist it was never declared in the ⟨*parameter text*⟩ (somewhat like `\def#1{#2}`), then raise an error and abort. Otherwise just flush `#`⟨*number*⟩.

```
\cs_new:Npn \__namedef_in_replacement:nN #1 #2
  {
    \cs_if_exist:cTF { \c__namedef_prefix_tl #1 }
      {
        \exp_args:Nf \__namedef_append_output:nnNwnn
          { \use:c { \c__namedef_prefix_tl #1 } }
          { }
      }
      {
        \msg_expandable_error:nnn { namedef } { unknown-parm } {#1}
        \exp_args:Ne \__namedef_append_output:nnNwnn
          { #2 \__namedef_begin_name_token: #1 \__namedef_end_name_token: }
          { \cs_end: { } \use_none:nn }
      }
        #2
  }
```

(*End definition for* `\__namedef_in_parameter:nN` *and* `\__namedef_in_replacement:nN`*.*)

5.5 Scanning routine

`\__namedef_replace_parm:Nn`
`\__namedef_replace_loop:w`
`\__namedef_replace_end:wnn`
`\__namedef_flush:nw`
`\__namedef_append_output:nnNwnn`
`\__namedef_abort_definition:w`

`\__namedef_replace_parm:Nn` uses the same looping principle as in l3tl's `\__tl_act:NNNnn`. It scans the input (here, the ⟨*parameter text*⟩ and ⟨*replacement text*⟩, separately) token by token, differentiating spaces, braced tokens (groups), and "normal" tokens.

```
\cs_new:Npn \__namedef_replace_parm:Nn #1 #2
  {
    \exp_after:wN #1
      \exp:w
      \__namedef_replace_loop:w #2
        \q__namedef_mark \q__namedef_stop { } { }
  }
\cs_new:Npn \__namedef_replace_loop:w #1 \q__namedef_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__namedef_replace_normal:N }
      {
        \tl_if_head_is_group:nTF {#1}
          { \__namedef_replace_group:n }
          { \__namedef_replace_space:w }
      }
    #1 \q__namedef_stop
  }
\cs_new:Npn \__namedef_replace_end:wnn \q__namedef_stop #1 #2
  { \exp_end: {#1} {#2} }
\cs_new:Npn \__namedef_flush:nw #1
    #2 \q__namedef_stop #3 #4
  { \__namedef_replace_loop:w #2 \q__namedef_stop { #3 #1 } {#4} }
```

`\__namedef_append_output:nnNwnn` takes three arguments (a parameter number, a parameter name, and a parameter token) and the two output slots as `#5` and `#6`. It appends the parameter token and number to the first output slot, and a definition `\cs_set:cpn {\c__namedef_prefix_tl` ⟨*name*⟩`}` `{`⟨*number*⟩`}` to the second output slot.

```
\cs_new:Npn \__namedef_append_output:nnNwnn #1 #2 #3
    #4 \q__namedef_stop #5 #6
  {
    \__namedef_replace_loop:w #4 \q__namedef_stop
      { #5 #3#1 }
      { #6 \cs_set:cpn { \c__namedef_prefix_tl #2 } {#1} }
  }
```

This macro doesn't really abort the definition at the time it's called because it's called from within an `f`-expansion context, so an attempt to end that prematurely will hardly end well. Instead it hijacks the process by inserting `\__namedef_skip_to_scan_mark:w` in the second output slot, so that the definition end as soon as the scanning ends.

```
\cs_new:Npn \__namedef_abort_definition:w
    #1 \q__namedef_stop #2 #3
```

```
  {
    \__namedef_replace_loop:w #1 \q__namedef_stop
      {#2} { #3 \group_end: \__namedef_skip_to_scan_mark:w }
  }
```

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

__namedef_replace_normal:N
__namedef_replace_group:n
__namedef_flush_group:nnw
__namedef_replace_space:w

Spaces are just passed through: they aren't parameter tokens nor valid delimiters, so need no special treatment.

Braced tokens are recursively scanned by `\__namedef_replace_parm:Nn`, and the output is flushed inside a pair of braces (explicit catcode 1 and 2 tokens are normalised to {$_1$ and }$_2$, respectively)

The remaining tokens are examined for their meaning. If the token is the quark `\q__namedef_mark`, the scanning stops; if the token is a parameter token, what follows is examined with `\__namedef_grab_parm:Nw` to check if a replacement should be done; otherwise it's fluhsed to the output.

```
\cs_new:Npn \__namedef_replace_normal:N #1
  {
    \token_if_eq_meaning:NNTF \q__namedef_mark #1
      { \__namedef_replace_end:wnn }
      {
        \token_if_parameter:NTF #1
          { \__namedef_grab_parm:Nw }
          { \__namedef_flush:nw }
            {#1}
      }
  }
\cs_new:Npn \__namedef_replace_group:n #1
  { \__namedef_replace_parm:Nn \__namedef_flush_group:nnw {#1} }
\cs_new:Npn \__namedef_flush_group:nnw #1 #2
  { \__namedef_flush:nw { {#1} } }
\exp_last_unbraced:NNo
\cs_new:Npn \__namedef_replace_space:w \c_space_tl
  { \__namedef_flush:nw { ~ } }
```

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

5.6 Parsing a parameter

__namedef_grab_parm:Nw
__namedef_grab_parm_aux:NNw
__namedef_grab_parm_noop:NNw
__namedef_grab_parm_scan:NNw
__namedef_grab_parm_loop:nw
__namedef_grab_parm_end:nw
__namedef_parm_get_normal:nN
__namedef_parm_get_group:nn
__namedef_parm_get_space:nw

These macros are the final pieces of the parameter replacement routine. `\__namedef_grab_parm:Nw` checks if the next token in the stream is a valid `N`-type. If it is, then `\__namedef_grab_parm_aux:NNw` checks if its character code is equal to `\__namedef_begin_name_token:`, and if it is, then call `\__namedef_grab_parm_scan:NNw` to scan ahead for the named parameter. In all other cases, the tokens grabbed are not named parameters, so they are flushed to the output.

```
\cs_new:Npn \__namedef_grab_parm:Nw #1 #2 \q__namedef_stop
  {
    \tl_if_head_is_N_type:nTF {#2}
      { \__namedef_grab_parm_aux:NNw }
      { \__namedef_flush:nw }
        #1 #2 \q__namedef_stop
  }
\cs_new:Npn \__namedef_grab_parm_aux:NNw #1 #2
```

```
  {
    \exp_args:No \token_if_eq_charcode:NNTF
        { \__namedef_begin_name_token: } #2
      { \__namedef_grab_parm_scan:NNw }
      { \__namedef_grab_parm_noop:NNw }
        #1 #2
  }
```

Here we have to take care not to flush `\q__namedef_mark` to the output.

```
\cs_new:Npn \__namedef_grab_parm_noop:NNw #1 #2
  {
    \token_if_eq_meaning:NNTF \q__namedef_mark #2
      { \__namedef_flush:nw { #1 } #2 }
      { \__namedef_flush:nw { #1 #2 } }
  }
```

Here's the actual scanning routine. It would be a lot faster to just define a delimiter macro with the right tokens, however this would have two consequences: first, missing delimiters would be rather catastrophic, and second, the catcode of the `end` delimiter would need to match. With a manual scanning, we can kill off those two items at the cost of some performance.

The scanning routine is pretty standard: a looping macro, an output slot, the tokens to be scanned, `\q__namedef_stop` to delimit the whole thing (`\q__namedef_mark` is redundant here: the one from the main scanning rountine is already in place), and the parameter token safely stored at the end:

```
\cs_new:Npn \__namedef_grab_parm_scan:NNw #1 #2 #3 \q__namedef_stop
  { \__namedef_grab_parm_loop:nw { } #3 \q__namedef_stop {#1} }
\cs_new:Npn \__namedef_grab_parm_loop:nw #1 #2 \q__namedef_stop
  {
    \tl_if_head_is_N_type:nTF {#2}
      { \__namedef_parm_get_normal:nN }
      {
        \tl_if_head_is_group:nTF {#2}
          { \__namedef_parm_get_group:nn }
          { \__namedef_parm_get_space:nw }
      }
    {#1} #2 \q__namedef_stop
  }
```

If the end of the token list was reached (signalled by `\q__namedef_mark`), the `end` delimiter is missing. If so, raise an error and recover as gracefully as possible. Otherwise, if the current token is the same character as the `\__namedef_end_name_token:`, then the scaning is complete.

```
\cs_new:Npn \__namedef_parm_get_normal:nN #1 #2
  {
    \token_if_eq_meaning:NNTF \q__namedef_mark #2
      {
        \msg_expandable_error:nn { namedef } { missing-end }
        \__namedef_grab_parm_end:nw {#1} #2
      }
      {
        \exp_args:No \token_if_eq_charcode:NNTF
            { \__namedef_end_name_token: } #2
          { \__namedef_grab_parm_end:nw {#1} }
          { \__namedef_grab_parm_loop:nw {#1#2} }
      }
  }
\cs_new:Npn \__namedef_parm_get_group:nn #1 #2
  { \__namedef_grab_parm_loop:nw { #1{#2} } }
\cs_new:Npn \__namedef_parm_get_space:nw #1 ~
  { \__namedef_grab_parm_loop:nw { #1~ } }
\cs_new:Npn \__namedef_grab_parm_end:nw #1 #2 \q__namedef_stop #3
  { \__namedef_handle_parameter:nN {#1} #3 #2 \q__namedef_stop }
```

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

5.7 Changing delimiters

`\__namedef_begin_name_token:`
`\__namedef_end_name_token:`

These two hold the delimiters for named parameters. They are initialised here so that we can use `\named` (just to show off) ahead, and they are redefined every time `\NamedDelim` is used.

```
\cs_new:Npn \__namedef_begin_name_token: { [ }
\cs_new:Npn \__namedef_end_name_token: { ] }
```

(*End definition for* `\__namedef_begin_name_token:` *and* `\__namedef_end_name_token:`*.*)

`\NamedDelim`
`\globalNamedDelim`
`\__namedef_named_delim_set:Nnn`
`\__namedef_check_delimiter:n`

At this point everything for the `\named` macro is set up, so we can start using it. Now just some syntactic sugar to allow the modification of the named argument delimiters.

Both `\NamedDelim` and `\globalNamedDelim` take two arguments, an initial and final delimiters for the named argument. Both delimters should be single non-control sequence tokens. Some of these restrictions could be lifted, but it's not really necessary because the choice of delimiter should not influence the working of the code, only the readability. A code with `\NamedDelim[]` and `\def\test#[1][#[2]]{[#[1]][#[2]]}` should work without problems; the only restriction is that `\__namedef_end_name_token:` (*i.e.*, the second argument of `\NamedDelim`) cannot appear in the parameter name.

```
\cs_new_protected:Npn \NamedDelim
  { \__namedef_named_delim_set:Nnn \cs_set:Npn  }
\cs_new_protected:Npn \globalNamedDelim
  { \__namedef_named_delim_set:Nnn \cs_gset:Npn }
\named \cs_new_protected:Npn \__namedef_named_delim_set:Nnn
    #[def] #[begin] #[end]
  {
    \tl_trim_spaces_apply:nN {#[begin]} \__namedef_check_delimiter:n
    \tl_trim_spaces_apply:nN {#[end]}   \__namedef_check_delimiter:n
    #[def] \__namedef_begin_name_token: {#[begin]}
    #[def] \__namedef_end_name_token: {#[end]}
    \s__namedef
  }
```

Here the ⟨*token*⟩ is checked against a bunch of forbidden cases.

```
\named \cs_new_protected:Npn \__namedef_check_delimiter:n #[token]
  {
```

It can't be empty (nor a space: they were trimmed above):

```
    \tl_if_empty:nT {#[token]}
      {
        \msg_error:nn { namedef } { blank-delim }
        \__namedef_skip_to_scan_mark:w
      }
```

It can't be multiple tokens:

```
    \tl_if_single_token:nF {#[token]}
      {
        \msg_error:nnn { namedef } { multi-token-delim } {#[token]}
        \__namedef_skip_to_scan_mark:w
      }
```

It can't be an implicit begin- or end-group token:

```
    \bool_lazy_or:nnT
        { \token_if_group_begin_p:N #[token] }
        { \token_if_group_end_p:N #[token] }
      {
        \msg_error:nnx { namedef } { group-delim }
          { \cs_to_str:N #[token] }
        \__namedef_skip_to_scan_mark:w
      }
```

It can't be a parameter token:

```
    \token_if_parameter:NT #[token]
      {
        \msg_error:nnx { namedef } { param-delim }
          { \cs_to_str:N #[token] }
        \__namedef_skip_to_scan_mark:w
      }
```

It can't be a control sequence:

```
    \token_if_cs:NT #[token]
      {
        \msg_error:nnx { namedef } { macro-delim }
          { \c_backslash_str \cs_to_str:N #[token] }
        \__namedef_skip_to_scan_mark:w
      }
  }
```

(*End definition for* `\NamedDelim` *and others. These functions are documented on page 2.*)

5.8 Messages

Now we define the messages used throughout the package.

```
\msg_new:nnn { namedef } { repeated-parm }
  {
    Parameter~\iow_char:N\#[#1]~duplicated~in~
    definition~of~\l__namedef_macro_tl.
  }
\msg_new:nnn { namedef } { unknown-parm }
  {
    Unknown~parameter~\iow_char:N\#[#1]~in~
    definition~of~\l__namedef_macro_tl.
  }
\msg_new:nnn { namedef } { multi-token-delim }
  {
    Invalid~\iow_char:N\\named~parameter~delimiter~`#1'.~
    Delimiters~for~named~parameters~must~be~single~tokens.
  }
\msg_new:nnn { namedef } { macro-delim }
```

```
  {
    Invalid~\iow_char:N\\named~parameter~delimiter~`#1'.~
    Delimiters~for~named~parameters~can't~be~control~sequence~nor~
    active~characters.
  }
\msg_new:nnn { namedef } { group-delim }
  {
    Invalid~\iow_char:N\\named~parameter~delimiter~`\iow_char:N\\#1'.~
    Delimiters~for~named~parameters~can't~be~
      begin-/end-group~character~tokens.
  }
\msg_new:nnn { namedef } { blank-delim }
  {
    Invalid~\iow_char:N\\named~parameter~delimiter.~
    Delimiters~for~named~parameters~can't~be~empty~nor~space~tokens.
  }
\msg_new:nnn { namedef } { param-delim }
  {
    Invalid~\iow_char:N\\named~parameter~delimiter.~
    Delimiters~for~named~parameters~can't~be~parameter~tokens.
  }
\msg_new:nnn { namedef } { missing-end }
  {
    Missing~\__namedef_end_name_token:\iow_char:N\ inserted~in~
    definition~of~\l__namedef_macro_tl.
  }
```

Now execute the end package hook (in LaTeX it is `\prg_do_nothing:`, but in plain TeX it does `\ExplSyntaxOff`).

```
\__namedef_end_package_hook:
```

Index

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

Symbols

`\#` 333, 338
`\\` 117, 343, 348, 354, 360, 365
`\␣` 370

B

`\begingroup` 3
bool commands:
  `\bool_lazy_or:nnTF` 310

C

cs commands:
  `\cs:w` 72, 152
  `\cs_end:` 72, 145, 152, 171
  `\cs_gset:Npn` 288
  `\cs_if_exist:NTF` 10, *13*, 161
  `\cs_if_exist_use:NTF` 24
  `\cs_new:Npn` *2*, 37, 38, 79, 91, 96, 143, 159, 175, 182, 193, 195, 198, 205, 211, 222, 224, 227, 229, 236, 244, 250, 252, 263, 277, 279, 281, 283, 284
  `\cs_new_eq:NN` 12, 17, 39, 139
  `\cs_new_protected:Npn` 40, 41, 62, 109, 114, 121, 129, 134, 140, 285, 287, 289, 298
  `\cs_set:Npn` 111, 203, 286
  `\cs_set_eq:NN` 118, 136
  `\cs_set_nopar:Npn` 64
  `\cs_set_protected:Npn` 20, 87

  \cs_to_str:N 315, 321, 327
\csname 4

D

\def *1*, *2*, *2*, *3*, *7*, *8*

E

\edef *3*, *11*
\else 6
else commands:
  \else: 58, 100, 147
\endcsname 4
\endgroup 3
exp commands:
  \exp:w 178
  \exp_after:wN *11*, 69, 70, 71, 112, 118, 146, 148, 152, 177
  \exp_args:Ne 169
  \exp_args:Nf 155, 163
  \exp_args:No 238, 271
  \exp_end: 194
  \exp_last_unbraced:NNo 226
  \exp_not:N *11*, 67, 79, 82, 83, 84, 93, 99, 100, 101, 102, 112, 118
  \exp_not:n 73
  \exp_stop_f: 56
\expandafter 3, 4
\ExplSyntaxOff 13, 17, 19, *19*
\ExplSyntaxOn 9

F

\fi 8
fi commands:
  \fi: 51, 52, 53, 54, 55, 60, 102, 149
flag internal commands:
  \flag␣__namedef_parm_count *13*, 31
flag commands:
  \flag_height:n 156
  \flag_new:n 31
  \flag_raise:n 153

G

\global *2*, *10*
\globalNamedDelim *2*, *17*, 285
group commands:
  \group_begin: 18, 116
  \group_end: 22, 131, 209

H

\hello *1*

I

\if *2*
if commands:
  \if_cs_exist:w 145
  \if_int_compare:w 50
  \if_meaning:w 51, 52, 53, 54, 55, 99
\ifx 4
\input 5
int commands:
  \int_set:Nn 117
iow commands:
  \iow_char:N 333, 338, 343, 348, 354, 360, 365, 370
  \iow_log:n 24

L

\let *13*
\long *1*, *2*, *8*, 107

M

\meaning *5*
msg commands:
  \msg_error:nn 302
  \msg_error:nnn 307, 314, 320, 326
  \msg_expandable_error:nn 267
  \msg_expandable_error:nnn 150, 168
  \msg_new:nnn 331, 336, 341, 346, 352, 358, 363, 368

N

\named *1*, *2*, *3–8*, *10*, *17*, *17*, 40, 289, 298
\NamedDelim *1*, *2*, *17*, *17*, 285
namedef internal commands:
  __namedef_abort_definition:w 175
  __namedef_append_output:nnNwnn *14*, 155, 163, 169, 175
  __namedef_begin_name_token: *15*, 170, 239, 283, 294
  __namedef_check_delimiter:n 285
  __namedef_define:nnnN *12*, 109
  __namedef_detect_prefixes:Nn 45, 62
  __namedef_end_name_token: *16*, *17*, 170, 272, 283, 295, 370
  __namedef_end_package_hook: 3, 373
  __namedef_extract_long:n *10*, 62
  __namedef_extract_long_aux:w 62
  __namedef_extract_outer:n *10*, 62
  __namedef_extract_outer_aux:w 62
  __namedef_extract_prefixes:w *10*, 62
  __namedef_extract_protected:n *10*, 62
  __namedef_extract_protected_aux:w 62
  __namedef_flush:nw 175, 218, 225, 228, 233, 247, 248
  __namedef_flush_group:nnw 211
  __namedef_grab_parm:Nw *15*, *15*, 217, 229
  __namedef_grab_parm_aux:NNw *15*, 229

__namedef_grab_parm_end:nw 229
__namedef_grab_parm_loop:nw 229
__namedef_grab_parm_noop:NNw 229
__namedef_grab_parm_scan:NNw *15*, 229
__namedef_grab_prefix:nN 40
__namedef_handle_parameter:nN 109, 282
__namedef_if_prefix:NTF 43, 48
__namedef_if_prefix_p:N 48
__namedef_in_parameter:nN *12*, *13*, 124, 143
__namedef_in_replacement:nN *12*, *13*, 125, 143
__namedef_kill_outer:nN 45, 109
\l__namedef_macro_tl 33, 123, 334, 339, 371
__namedef_named_delim_set:Nnn 285
__namedef_parameter_output:nnw 109
__namedef_parm_get_group:nn 229
__namedef_parm_get_normal:nN 229
__namedef_parm_get_space:nw 229
\c__namedef_prefix_tl *13*, *14*, 32, 145, 152, 161, 164, 203
__namedef_replace_end:wnn 175, 214
__namedef_replace_group:n 188, 211
__namedef_replace_loop:w 175
__namedef_replace_named:nNnn *10*, *11*, *12*, 109
__namedef_replace_normal:N 185, 211
__namedef_replace_parameter:Nn *12*, 109
__namedef_replace_parm:Nn *12*, *14*, *15*, 137, 175, 223
__namedef_replace_space:w 189, 211
__namedef_skip_to_scan_mark:nw 37
__namedef_skip_to_scan_mark:w *14*, 37, 209, 303, 308, 316, 322, 328
__namedef_start:nNp *11*, 109
__namedef_tmp:w *10*, 20, 27, 39, 64, 87, 106, 107, 108, 111, 112
\namedefDate 29
\namedefVersion 29
\newcommand *2*, *2*, *6*, *7*
\NewDocumentCommand *2*, *3*
\noexpand *11*

O

\outer *2*, *10*, *11*, 108

P

prg commands:
  \prg_do_nothing: 12, *19*
  \prg_new_conditional:Npnn 48
  \prg_return_false: 59
  \prg_return_true: 57
\protected *2*, 106
\ProvidesExplPackage 10, 14

Q

quark commands:
  \quark_new:N 34, 35
quark internal commands:
  \s__namedef *9*, 36, 37, 38, 72, 80, 94, 97, 127, 296
  \q__namedef_mark *15*, *16*, *16*, *16*, 34, 180, 213, 246, 265
  \q__namedef_stop *16*, 34, 180, 182, 191, 193, 196, 197, 199, 201, 206, 208, 229, 234, 250, 251, 252, 261, 281, 282

R

\relax 4, *11*, *13*, *13*
\RequirePackage 7, *8*

S

scan commands:
  \scan_new:N 36
  \scan_stop: 55, 94, 99, 118
seq commands:
  \seq_use:Nn *8*
str commands:
  \c_backslash_str 327
\string *3*, *4*

T

tex commands:
  \tex_escapechar:D 117
  \tex_global:D 52
  \tex_long:D 54
  \tex_outer:D 53
  \tex_protected:D 51
tl commands:
  \c_space_tl 227
  \tl_const:Nn 32
  \tl_gset:Nn 23
  \tl_if_empty:nTF 300
  \tl_if_head_is_group:nTF 187, 257
  \tl_if_head_is_N_type:nTF 184, 231, 254
  \tl_if_single_token:nTF 305
  \tl_new:N 33
  \tl_set:Nn 123
  \tl_to_str:n 80
  \tl_trim_spaces_apply:nN 292, 293
tl internal commands:
  __tl_act:NNNnn *14*
token commands:
  \token_if_cs:NTF 324

\token_if_eq_charcode:NNTF . 238, 271
\token_if_eq_meaning:NNTF 213, 246, 265
\token_if_group_begin_p:N 311
\token_if_group_end_p:N 312
\token_if_parameter:NTF 216, 318
\token_to_meaning:N 71
\token_to_str:N *8*, 94, 97, 123

U

use commands:
\use:N 164
\use:n 65, 77, 89
\use_i:nn 146
\use_ii:nn 148
\use_none:n 152
\use_none:nn 171

W

\wlog 24

X

\xdef *3*