# The LaTeX3 kernel: style guide for code authors

The LaTeX Project*

Released 2026-11-16

## Contents

**1 Introduction** **1**

**2 Documentation style** **1**

**3 Format of the code itself** **2**

**4 Code conventions** **3**

**5 Private and internal functions** **3**
5.1 Access from other modules . . . . . 4
5.2 Access to primitives . . . . . 4

**6 Auxiliary functions** **4**

**7 Functions with 'weird' arguments** **5**

## 1 Introduction

This document is intended as a style guide for authors of code and documentation for the LaTeX3 kernel. It covers both aspects of coding style and the formatting of the sources. The aim of providing these guidelines is help ensure consistency of the code and sources from different authors. Experience suggests that in the long-term this helps with maintenance. There will of course be places where there are exceptions to these guidelines: common sense should always be applied!

## 2 Documentation style

LaTeX3 source and documentation should be written using the document class `l3doc` in `dtx` format. This class provides a number of logical mark up elements, which should be used where possible. In the main, this is standard LaTeX practice, but there are a few points to highlight:

- Where possible, use `\cs` to mark up control sequences rather than using a verbatim environment.

*E-mail: latex-team@latex-project.org

- Arguments which are given in braces should be marked using `\Arg` when code-level functions are discussed, but using `\marg` for document functions.
- The names TEX, LATEX, *etc.* use the normal logical mark up followed by an empty group (`{}`), with the exception of `\LaTeX3`, where the number should follow directly.
- Where in line verbatim text is used, it should be marked up using the `|...|` construct (i.e., vertical bars delimit the verbatim text).
- In line quotes should be marked up using the `\enquote` function.
- Where numbers in the source have a mathematical meaning, they should be included in math mode. Such in-line math mode material should be marked up using `$...$` and *not* `\(...\)`.

Line length in the source files should be under 80 characters where possible, as this helps keep everything on the screen when editing files. In the `dtx` format, documentation lines start with a `%`, which is usually followed by a space to leave a "comment margin" at the start of each line.

As with code indenting (see later), nested environments and arguments should be indented by (at least) two spaces to make the nature of the nesting clear. Thus for example a typical arrangement for the `function` environment might be

```
\begin{function}{\seq_gclear:N,␣\seq_gclear:c}
␣␣\begin{syntax}
␣␣␣␣␣\cs{seq_gclear:N}␣\meta{sequence}
␣␣\end{syntax}
␣␣Clears␣all␣entries␣from␣the␣\meta{sequence}␣globally.
\end{function}
```

The "outer" `%␣\begin{function}` should have the customary space after the `%` character at the start of the line.

In general, a single `function` or `macro` environment should be used for a group of closely-related functions, for example argument specification variants. In such cases, a comma-separated list should be used, as shown in the preceding example.

# 3 Format of the code itself

The requirement for fewer than 80 characters per line applies to the code itself as well as the surrounding documentation. A number of the general style principles for LATEX3 code apply: these are described in the following paragraph and an example is then given.

With the exception of simple runs of parameter (`{#1}`, `#1#2`, etc.), everything should be divided up using spaces to make the code more readable. In general, these will be single spaces, but in some places it makes more sense to align parts of the code to emphasize similarity. (Tabs should not be used for introducing white space.)

Each conceptually-separate step in a function should be on a separate line, to make the meaning clearer. Hence the `false` branch in the example uses two lines for the two auxiliary function uses.

Within the definition, a two-space indent should be used to show each "level" of code. Thus in the example `\tl_if_empty:nTF` is indented by two spaces, but the two branches are indented by four spaces. Within the `false` branch, the need for multiple

lines means that an additional two-space indent should be used to show that these lines are all part of the brace group.

The result of these lay-out conventions is code which in general looks like the example:

```
\cs_new:Npn␣\module_foo:nn␣#1#2
␣␣{
␣␣␣␣\tl_if_empty:nTF␣{#1}
␣␣␣␣␣␣␣{␣\module_foo_aux:n␣{␣X␣#2␣}␣}
␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣\module_foo_aux:nn␣{#1}␣{#2}
␣␣␣␣␣␣␣␣␣\module_foo_aux:n␣{␣#1␣#2␣}
␣␣␣␣␣␣␣}
␣␣}
```

# 4 Code conventions

All code-level functions should be "long" if they accept any arguments, even if it seems "very unlikely" that a `\par` token will be passed. Thus `\cs_new_nopar:Npn` and so forth should only be used to create interfaces at the document level (where trapping `\par` tokens may be appropriate) or where comparison to other code known not to be "long" is required (*e.g.* when working with mixed LaTeX $2_\varepsilon$/expl3 situations).

The expandability of each function should be well-defined. Functions which cannot be fully expanded must be `protected`. This means that expandable functions must themselves only contain expandable material. Functions which use any non-expandable material must be defined using `\cs_new_protected:Npn` or similar.

When using `\cs_generate_variant:Nn`, group related variants together to make the pattern clearer. A common example is variants of a function which has an `N`-type first argument:

```
\cs_generate_variant:Nn \foo:Nn {     NV , No }
\cs_generate_variant:Nn \foo:Nn { c , cV , co }
```

There may be cases where omitting braces from `o`-type arguments is desirable for performance reasons. This should only be done if the argument is a single token, thus for example

```
\tl_set:No \l_some_tl \l_some_other_tl
```

remains clear and can be used where appropriate.

# 5 Private and internal functions

Private functions (those starting `\__`) should not be used between modules. The only exception is where a "family" of modules share some "internal" methods: this happens most obviously in the kernel itself. Any internal functions or variables *must* be documented in the same way as public ones.

The DocStrip method should be used for internal functions in a module. This requires a line

```
%<@@=⟨module⟩>
```

at the start of the source (`.dtx`) file, with internal functions then written in the form

```
\cs_new_protected:Npn \@@_function:nn #1#2
  ...
```

### 5.1 Access from other modules

There may be cases where it is useful to use an internal function from a third-party module (this includes cases where you are the author of both but they are not part of the same "family"). In these cases, you should *copy* the definition of the internal function to your code: this avoids relying on non-documented interfaces. At the same time, it is strongly encouraged that you discuss your requirements with the author of the code you need to access. The best long-term solution to these cases is for new documented interfaces to be added to the parent module.

### 5.2 Access to primitives

As expl3 is still a developing system, there are places where direct access to engine primitives is required. These are all marked as "do not use" in the code and so require special handling. Where a programmer is sure that they need to use a primitive (for example where the team have not yet covered access to an area) then a local copy of the primitive should be made, for example

```
\cs_new_eq:NN \__module_message:w \tex_message:D
% ...
\cs_new_protected:Npn \__module_fancy_msg:n #1
  { \__module_message:w { *** #1 *** } }
```

This approach makes it possible for the team and others to find such usage (by searching for the `:D` argument type) but avoids multiple uses in general code.

At the same time, the team ask that these use cases are raised by emailing us. The team are keen to collect use cases for areas that have not yet been addressed and to provide new code where the required interfaces become clear.

Programmers using primitives should be ready to make updates to their code as the team develop additional interfaces.

## 6 Auxiliary functions

In general, the team encourages the use of descriptive names in LaTeX3 code. Thus many helper functions would have names which describe briefly what they do, rather than simply indicating that they are auxiliary to some higher-level function. However, there are places where one or more `aux` functions are required. Where possible, these should be differentiated by signature

```
\cs_new_protected:Npn \@@_function:nn #1#2
  {
    ...
  }
\cs_new_protected:Npn \@@_function_aux:nn #1#2
  {
    ...
```

```
  }
\cs_new_protected:Npn \@@_function_aux:w #1#2 \q_stop
  {
    ...
  }
```

Where more than one auxiliary shares the same signature, the recommended naming scheme is `auxi`, `auxii` and so on.

```
\cs_new_protected:Npn \@@_function_auxi:nn #1#2
  {
    ...
  }
\cs_new_protected:Npn \@@_function_auxii:nn #1#2
  {
    ...
  }
```

The use of `aux_i`, `aux_ii`, etc. is discouraged as this conflicts with the convention used by `\use_i:nn` and related functions.

# 7 Functions with 'weird' arguments

When defining commands that do not follow the usual convention of accepting arguments as single-tokens or braced-text, the `w` argument specifier is used to denote that the function signature cannot fully describe the syntax. Two examples from the LaTeX3 kernel are:

`\use_none_delimit_by_q_stop:w` ⟨ *text* ⟩ `\q_stop`
`\use_i_delimit_by_q_stop:nw` {⟨ *arg* ⟩} ⟨ *text* ⟩ `\q_stop`

More complex definitions are possible if commands are to parse tokens, such as the internal kernel command

```
\cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
  { \tl_set:Nn #3 {#1} }
```

When the `w` specifier is being used, it is encouraged not to try and complicate the rest of the signature too much—for example, it would be considered poor style to have a function with a signature like `\foo_bar:wnw` unless there were very clear reasons of code clarity. A signature such as `:ww` would certainly be discouraged.

Examining the examples above, it can be seen that there are scenarios in which it may make logical sense for having a signature such as `:wN` or `:nw`, but when in doubt the recommended approach is to simply use `:w` as a catch-all.