The LaTeX3 kernel: style guide for code authors*

The LaTeX3 Project†

Released 2011/09/07

Contents

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.

- Arguments which are given in braces should be marked using `\Arg` when code-level functions are discussed, but using `\marg` for document functions.

---

*This file describes v2772, last revised 2011/09/07.

†E-mail: latex-team@latex-project.org

- 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 to 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 less than 80 characters per line applies to the code itself as well as the surrounding documentation. A number of the general style principals 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 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 emphasise 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 will in general look 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 functions which accept no arguments at all.

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 are 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.