

The l3doc class – experimental*

The L^AT_EX3 Project[†]

Released 2023-05-15

Contents

1	Introduction	1
2	Features of other packages	2
2.1	The hypdoc package	2
2.2	The docmfp package	2
2.3	The xdoc2 package	2
2.4	The gmdoc package	3
3	Problems & Todo	3
4	Documentation	3
4.1	Configuration	3
4.2	Partitioning documentation and implementation	4
4.3	General text markup	4
4.4	Describing functions in the documentation	6
4.5	Describing functions in the implementation	7
4.6	Keeping things consistent	8
4.7	Documenting templates	8
	Index	9

1 Introduction

Code and documentation for this class have been written prior to the change of `doc` from version 2 to version 3, which already shows how far behind this class currently is. So take the following warning seriously please:

**It is much less stable than the main `expl3` packages.
Use at own risk!**

*On popular request we now distribute the document for this experimental class. However, please note that it is by no means in final state and is *likely* to undergo modifications, even *incompatible ones*! Thus, using it might therefore require you to do updates, if the class changes.

[†]<https://www.latex-project.org/latex3/>

This is an ad-hoc class for documenting the `expl3` bundle, a collection of modules or packages that make up L^AT_EX3's programming environment. Eventually it will replace the `ltxdoc` class for L^AT_EX3, but not before the good ideas in `hypdoc`, `xdoc2`, `docmfp`, and `gmdoc` are incorporated.

It is written as a “self-contained” docstrip file: executing `latex l3doc.dtx` generates the `l3doc.cls` file and typesets this documentation; execute `tex l3doc.dtx` to only generate `l3doc.cls`.

2 Features of other packages

This class builds on the `ltxdoc` class and the `doc` package, but in the time since they were originally written some improvements and replacements have appeared that we would like to use as inspiration.

These packages or classes are `hypdoc`, `docmfp`, `gmdoc`, and `xdoc`. I have summarised them below in order to work out what sort of features we should aim at a minimum for `l3doc`.

2.1 The `hypdoc` package

This package provides hyperlink support for the `doc` package. I have included it in this list to remind me that cross-referencing between documentation and implementation of methods is not very good. (*E.g.*, it would be nice to be able to automatically hyperlink the documentation for a function from its implementation and vice-versa.)

2.2 The `docmfp` package

- Provides `\DescribeRoutine` and the `routine` environment (*etc.*) for MetaFont and MetaPost code.
- Provides `\DescribeVariable` and the `variable` environment (*etc.*) for more general code.
- Provides `\Describe` and the `Code` environment (*etc.*) as a generalisation of the above two instantiations.
- Small tweaks to the DocStrip system to aid non-L^AT_EX use.

2.3 The `xdoc2` package

- Two-sided printing.
- `\NewMacroEnvironment`, `\NewDescribeEnvironment`; similar idea to `docmfp` but more comprehensive.
- Tons of small improvements.

2.4 The **gmdoc** package

Radical re-implementation of **doc** as a package or class.

- Requires no `\begin{macrocode}` blocks!
- Automatically inserts `\begin{macro}` blocks!
- And a whole bunch of other little things.

3 Problems & Todo

Problems at the moment: (1) not flexible in the types of things that can be documented; (2) no obvious link between the `\begin{function}` environment for documenting things to the `\begin{macro}` function that's used analogously in the implementation.

The `macro` should probably be renamed to `function` when it is used within an implementation section. But they should have the same syntax before that happens!

Furthermore, we need another “layer” of documentation commands to account for “user-macro” as opposed to “code-functions”; the `expl3` functions should be documented differently, probably, to the `ltxcmd` user macros (at least in terms of indexing).

In no particular order, a list of things to do:

- Rename `function/macro` environments to better describe their use.
- Generalise `function/macro` for documenting “other things”, such as environment names, package options, even keyval options.
- New function like `\part` but for files (remove awkward “File” as `\partname`).
- Something better to replace `\StopEventually`; I'm thinking two environments `documentation` and `implementation` that can conditionally typeset/ignore their material. (This has been implemented but needs further consideration.)
- Hyperlink documentation and implementation of macros (see the DTX file of `svn-multi v2` as an example). This is partially done, now, but should be improved.

4 Documentation

4.1 Configuration

Before class options are processed, `l3doc` loads a configuration file `l3doc.cfg` if it exists, allowing you to customise the behaviour of the class without having to change the documentation source files.

For example, to produce documentation on letter-sized paper instead of the default A4 size, create `l3doc.cfg` and include the line

```
\PassOptionsToClass{letterpaper}{l3doc}
```

By default, `l3doc` selects the `T1` font encoding and loads the Latin Modern fonts. To prevent this, use the class option `cm-default`.

4.2 Partitioning documentation and implementation

doc uses the `\OnlyDocumentation/\AlsoImplementation` macros to guide the use of `\StopEventually{}`, which is intended to be placed to partition the documentation and implementation within a single `.dtx` file.

This isn't very flexible, since it assumes that we *always* want to print the documentation. For the `expl3` sources, I wanted to be able to input `.dtx` files in two modes: only displaying the documentation, and only displaying the implementation. For example:

```
\DisableImplementation
\DocInput{l3basics,l3prg,...}
\EnableImplementation
\DisableDocumentation
\DocInputAgain
```

The idea being that the entire `expl3` bundle can be documented, with the implementation included at the back. Now, this isn't perfect, but it's a start.

Use `\begin{documentation}...\end{documentation}` around the documentation, and `\begin{implementation}...\end{implementation}` around the implementation. The `\EnableDocumentation/\EnableImplementation` causes them to be typeset when the `.dtx` file is `\DocInput`; use `\DisableDocumentation/\DisableImplementation` to omit the contents of those environments.

Note that `\DocInput` now takes comma-separated arguments, and `\DocInputAgain` can be used to re-input all `.dtx` files previously input in this way.

4.3 General text markup

Many of the commands in this section come from `ltxdoc` with some improvements.

<code>\cmd</code>	<code>\cmd</code>	<code>[<options>]</code>	<code><control sequence></code>
<code>\cs</code>	<code>\cs</code>	<code>[<options>]</code>	<code>{<csname>}</code>

These commands are provided to typeset control sequences. `\cmd\foo` produces “`\foo`” and `\cs{foo}` produces the same. In general, `\cs` is more robust since it doesn’t rely on catcodes being “correct” and is therefore recommended.

These commands are aware of the `@@ l3docstrip` syntax and replace such instances correctly in the typeset documentation. This only happens after a `%<@@=<module>` declaration.

Additionally, commands can be used in the argument of `\cs`. For instance, `\cs{\meta{name}:\meta{signature}}` produces `\<name>:\<signature>`.

The `<options>` are a key–value list which can contain the following keys:

- `index=<name>`: the `<csname>` is indexed as if one had written `\cs{<name>}`.
- `no-index`: the `<csname>` is not indexed.
- `module=<module>`: the `<csname>` is indexed in the list of commands from the `<module>`; the `<module>` can in particular be `TeX` for “`TeX` and `LATeX 2ε`” commands, or empty for commands which should be placed in the main index. By default, the `<module>` is deduced automatically from the command name.
- `replace` is a boolean key (`true` by default) which indicates whether to replace `@@` as `l3docstrip` does.

These commands allow hyphenation of control sequences after (most) underscores. By default, a hyphen is used to mark the hyphenation, but this can be changed with the `cs-break-nohyphen` class option. To disable hyphenation of control sequences entirely, use `cs-break-off`.

<code>\tn</code>	<code>\tn</code>	<code>[<options>]</code>	<code>{<csname>}</code>
------------------	------------------	--------------------------------	-------------------------------

Analoguous to `\cs` but intended for “traditional” `TeX` or `LATeX 2ε` commands; they are indexed accordingly. This is in fact equivalent to `\cs [module=TeX, replace=false, <options>] {<csname>}`.

<code>\meta</code>	<code>\meta</code>	<code>{<name>}</code>
--------------------	--------------------	-----------------------------

`\meta` typesets the `<name>` italicised in *<angle brackets>*. Within a `function` environment or similar, angle brackets `<...>` are set up to be a shorthand for `\meta{...}`.

This function has additional functionality over its `ltxdoc` versions; underscores can be used to subscript material as in math mode. For example, `\meta{arg_{xy}}` produces “`argxy`”.

<code>\Arg</code>	<code>\Arg</code>	<code>{<name>}</code>
-------------------	-------------------	-----------------------------

<code>\marg</code>	Typesets the <code><name></code> as for <code>\meta</code> and wraps it in braces.
<code>\oarg</code>	
<code>\parg</code>	

The `\marg/\oarg/\parg` versions follow from `ltxdoc` in being used for “mandatory” or “optional” or “picture” brackets as per `LATeX 2ε` syntax.

<code>\file</code>	<code>\pkg</code>	<code>{<name>}</code>
--------------------	-------------------	-----------------------------

<code>\env</code>	These all take one argument and are intended to be used as semantic commands for representing files, environments, package names, and class names, respectively.
<code>\pkg</code>	
<code>\cls</code>	

```

\NB \NB {\tag} {\comments}
\NOTE \begin{NOTE} {\tag}
      {\comments}
\end{NOTE}

```

Make notes in the source that are not typeset by default. When the `show-notes` class option is active, the comments are typeset in a detokenized and verbatim mode, respectively.

4.4 Describing functions in the documentation

function (*env.*) Two heavily-used environments are defined to describe `expl3` functions and variables. If **variable** (*env.*) describing a variable, use the latter environment; it behaves identically to the **function** environment. Both of the above environments are typically combined with the **syntax** (*env.*) environment, to describe their syntax.

```

\begin{function}{\package_function_one:N, \package_function_two:n}
  \begin{syntax}
    \cs{package_function_one:N} \meta{cs}
    \cs{package_function_two:n} \marg{Argument}
  \end{syntax}
  Descriptive text here ...
\end{function}

```

```

\package_function_one:N \package_function_one:N <cs>
\package_function_two:n \package_function_two:n {\Argument}

```

Descriptive text here ...

Function environments take an optional argument to indicate whether the function(s) it describes are expandable (use `EXP`) or restricted-expandable (use `rEXP`) or defined in conditional forms (use `TF`, `pTF`, or `noTF`). Note that `pTF` implies `EXP` since predicates must always be expandable, and that `noTF` means that the function without `TF` should be documented in addition to `TF`. For the conditional forms `TF` and `pTF`, the argument of the **function** environment is *not* in fact a command that exists: in the example below, `\tl_if_empty:N` does not exist, but its conditional forms `\tl_if_empty:NT`, `\tl_if_empty:NF`, `\tl_if_empty:NTF` and predicate form `\tl_if_empty_p:N` exist:

```

\begin{function}[pTF]{\tl_if_empty:N, \tl_if_empty:c}
  \begin{syntax}
    \cs{tl_if_empty_p:N} \meta{tl-var}
    \cs{tl_if_empty:NTF} \meta{tl-var} \Arg{true code} \Arg{false code}
  \end{syntax}
  Tests if the \meta{token list variable} is entirely empty
  (\emph{i.e.}~contains no tokens at all).
\end{function}

```

```

\tl_if_empty_p:N * \tl_if_empty_p:N <tl var>
\tl_if_empty_p:c * \tl_if_empty:NTF <tl var> {\true code}
\tl_if_empty:NTF * {\false code}
\hr/>
\hr/>
\tl_if_empty:cTF * Tests if the <token list variable> is entirely
                    empty (i.e. contains no tokens at all).

```

texnote (*env.*) This environment is used to call out sections within **function** and similar environments that are only of interest to seasoned T_EX developers.

4.5 Describing functions in the implementation

macro (*env.*) The well-used environment from L^AT_EX 2_ε for marking up the implementation of macros/functions remains the **macro** environment. Some changes in **l3doc**: it now accepts comma-separated lists of functions, to avoid a very large number of consecutive **\end{macro}** statements. Spaces and new lines are ignored (the option **[verb]** prevents this).

```

% \begin{macro}{\foo:N, \foo:c}
%   \begin{macrocode}
... code for \foo:N and \foo:c ...
%   \end{macrocode}
% \end{macro}

```

If you are documenting an auxiliary macro, it's generally not necessary to highlight it as much and you also don't need to check it for, say, having a test function and having a documentation chunk earlier in a **function** environment. **l3doc** will pick up these cases from the presence of **__** in the name, or you may force marking as internal by using **\begin{macro}[int]** to mark it as such. The margin call-out is then printed in grey for such cases.

For documenting **expl3**-type conditionals, you may also pass this environment a **TF** option (and omit it from the function name) to denote that the function is provided with **T**, **F**, and **TF** suffixes. A similar **PTF** option prints both **TF** and **_p** predicate forms. An option **noTF** prints both the **TF** forms and a form with neither **T** nor **F**, to document functions such as **\prop_get:NN** which also have conditional forms (**\prop_get:NNTF**).

\TestFiles **\TestFiles{<list of files>}** is used to indicate which test files are used for the current code; they are printed in the documentation.

\UnitTested Within a **macro** environment, it is a good idea to mark whether a unit test has been created for the commands it defines. This is indicated by writing **\UnitTested** anywhere within **\begin{macro} ... \end{macro}**.

If the class option **checktest** is enabled, then it is an *error* to have a **macro** environment without a call to **Testfiles**. This is intended for large packages such as **expl3** that should have absolutely comprehensive tests suites and whose authors may not always be as sharp at adding new tests with new code as they should be.

\TestMissing If a function is missing a test, this may be flagged by writing (as many times as needed) **\TestMissing {<explanation of test required>}**. These missing tests are summarised in the listing printed at the end of the compilation run.

variable (*env.*) When documenting variable definitions, use the **variable** environment instead. Here

it behaves identically to the `macro` environment, except that if the class option `checktest` is enabled, variables are not required to have a test file.

`arguments` (*env.*) Within a `macro` environment, you may use the `arguments` environment to describe the arguments taken by the function(s). It behaves like a modified `enumerate` environment.

```
% \begin{macro}{\foo:nn, \foo:VV}
% \begin{arguments}
%   \item Name of froozle to be frazzled
%   \item Name of muble to be jubled
% \end{arguments}
% \begin{macrocode}
... code for \foo:nn and \foo:VV ...
% \end{macrocode}
% \end{macro}
```

4.6 Keeping things consistent

Whenever a function is either documented or defined with `function` and `macro` respectively, its name is stored in a sequence for later processing.

At the end of the document (*i.e.*, after the `.dtx` file has finished processing), the list of names is analysed to check whether all defined functions have been documented and vice versa. The results are printed in the console output.

If you need to do more serious work with these lists of names, take a look at the implementation for the data structures and methods used to store and access them directly.

4.7 Documenting templates

The following macros are provided for documenting templates; might end up being something completely different but who knows.

```
\begin{TemplateInterfaceDescription} {<template type name>}
  \TemplateArgument{none}{---}
OR ONE OR MORE OF THESE:
  \TemplateArgument {<arg no>} {<meaning>}
AND
\TemplateSemantics
  <text describing the template type semantics>
\end{TemplateInterfaceDescription}

\begin{TemplateDescription} {<template type name>} {<name>}
ONE OR MORE OF THESE:
  \TemplateKey {<key name>} {<type of key>}
    {<textual description of meaning>}
    {<default value if any>}
AND
\TemplateSemantics
  <text describing special additional semantics of the template>
\end{TemplateDescription}
```



```

\begin{InstanceDescription} [<text to specify key column width (optional)>]
    {<template type name>}{<instance name>}{<template name>}
ONE OR MORE OF THESE:
    \InstanceKey {<key name>}{<value>}
AND
\InstanceSemantics
    <text describing the result of this instance>
\end{InstanceDescription}

```

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.

A		M	
\AlsoImplementation	4	macro (env.)	7
\Arg	5	\marg	5
arguments (env.)	8	\meta	5
C		N	
\cls	5	\NB	6
\cmd	5	\NewDescribeEnvironment	2
\cs	5	\NewMacroEnvironment	2
		\NOTE	6
D		O	
\Describe	2	\oarg	5
\DescribeRoutine	2	\OnlyDocumentation	4
\DescribeVariable	2		
\DisableDocumentation	4	P	
\DisableImplementation	4	package commands:	
\DocInput	4	\package_function_one:N	6
\DocInputAgain	4	\package_function_two:n	6
		\parg	5
		\pkg	5
E		S	
\EnableDocumentation	4	\StopEventually	3, 4
\EnableImplementation	4	syntax (env.)	6
\env	5		
environments:		T	
arguments	8	\TestFiles	7
function	6	\TestFiles	7
macro	7	\TestMissing	7
syntax	6	\TestMissing	7
texnote	7	T _E X and L ^A T _E X 2 _ε commands:	
variable	6, 7	\part	3
		\partname	3
F		texnote (env.)	7
\file	5	tl commands:	
\foo	5	\tl_if_empty:NTF	6, 7
function (env.)	6		

\tl_if_empty_p:N	6, 7	\UnitTested	7
\tn	5		
	U		V
\UnitTested	7	variable (env.)	6, 7