The gmdoc bundle — a new tool for documenting (LA)TEX sources

Grzegorz Murzynowski
Sulejówek
natror at o2 dot pl

Abstract

There is a new package and a document class written by myself for documenting (LA)TEX packages and classes. 'Documenting' means that the comments are typeset as ordinary text and the code verbatim. All the control sequences are automatically indexed.

I think that the gmdoc package is superior to the doc package in two respects. First, the index entries, the table of contents and cross-references are made hyperlinks by default (with use of the hyperref package). Second, the gmdoc package allows you to typeset plain .sty and .cls files with the comments marked only with % (no special environments required).

The gmdoc bundle allows you to typeset the 'traditional' .dtx files, including LATEX 2ε Source and doc.dtx. The gmdoc bundle is available on CTAN.

gmdoc breaks free from macrocode

After I had written a couple of LATEX packages and even a class, I realised it would be nice to document them and make them available for everybody by putting them on CTAN. So I asked my TEX Guru, how can I document the code? I had already heard of the ideas of literate programming and self-documenting files. That idea is to write the code and the commentary on it simultaneously and mixed in one file, from which a respective program would extract the pure working code and another program would typeset a pretty narrated book or article about the code in question. Even before asking my TEX Guru, I had always added much commentary to my code, TEXnical or otherwise.

And my TEX Guru told me a fascinating tale of the doc package, and the .dtx files that make possible literate programming of (LA)TEX sources. The main idea, of changing the catcode of % depending on the mode of reading a file, or, from another side, of allowing the same file to be an executable (loadable) package or document class or a comprehensive documentation of that package or class depending on the catcode of %, enlightened my mind. But the rest of the tale, although equally fascinating, suggested that I do something I wouldn't like: mark up every piece of code with

```
%␣␣␣␣␣\begin{macrocode}
...
%␣␣␣␣␣\end{macrocode}
```

where the Percent and Four Spaces at the end are obligatory (see fig. 1). That would mean rewriting all of my .sty and .cls files.

Instead of such half-mechanical editorial work I chose to write my own documenting package such that just the percents would be sufficient as the markup, as in figure 2. Don't you think that three lines of commentary instead of seven do make a difference and are more readable?

So, the task was set: not to mark up the code. The most natural[1] solution to that was the active line end which could check whether the next line begins with a comment sign or not.

The fundamental idea of gmdoc is to consider the input file as consisting of two threads: the commentary, marked with the comment signs, and the code, which is the rest of the file.

Therefore the first thing done by the main input command is setting the catcode of the declared comment sign (% by default) to 'other' (12) and the catcode of ^^M (the line end char) to 'active' (13) and define the newly-active line end to check whether the next line begins with the comment sign.

To be precise, that active line end memorizes the number of leading spaces of the next line and *then* checks whether the first non-space character is the comment sign. (Later, if we discover that it's code, those spaces will be typeset as a respective indent.)

If the first non-space character of a line is not the comment sign, then the active line end opens a group for typesetting the code, within which the typewriter font is set, the catcodes of special characters are changed to 12 ('other') or 13 ('active') and

[1] I realize that what seems 'most natural' to me, may seem 'Against Nature' to some others ;-) .

```
%␣␣␣␣\begin{macrocode}
\def\macrocode{\macro@code
%␣␣␣␣\end{macrocode}
%␣␣␣␣Then␣we␣take␣care␣that␣all␣spaces␣have␣the␣same␣width,␣and␣that
%␣␣␣␣they␣are␣not␣discarded.
%␣␣␣␣\begin{macrocode}
␣␣␣\frenchspacing␣\@vobeyspaces
%␣␣␣␣\end{macrocode}
%␣␣␣␣Before␣closing,␣we␣need␣to␣call␣|\xmacro@code|.␣␣It␣is␣this
```

Figure 1: An excerpt from a .dtx file

```
\def\macrocode{\macro@code
␣␣%␣Then␣we␣take␣care␣that␣all␣spaces␣have␣the␣same␣width,␣and␣that
␣␣%␣they␣are␣not␣discarded.
␣␣␣\frenchspacing␣\@vobeyspaces%␣maybe␣an␣inline␣comment:
␣␣␣%␣Before␣closing,␣we␣need␣to␣call␣|\xmacro@code|.␣␣It␣is␣this
```

Figure 2: An example of the desired markup

the characters redefined in the latter case. Then an iterating macro is launched that eats the code character by character and typesets it until it finds the comment sign.

In the last case, the macro checks whether it's a real beginning of a commentary and not just a concatenation of two lines of code, and if so (it's a commentary), it closes the verbatim group and lets the commentary be typeset.

In the comment or 'narration' layer, the comment char's catcode is set to 'ignored' (9), as with doc.

The solutions developed make gmdoc superior to the doc package in one and a half respects:

1. The `macrocode` environment is not compulsory anymore. It is available, however.

1.5. Inline comments are supported. That is, they are not typeset verbatim, but in a roman font, as the comments should be IMO.

By the way, don't you find the gmdocish version of a source (fig. 2) more readable (than in fig. 1)?

Usage

We haven't yet seen *how* the package should be used. The usage is very simple and analogous to the usage of doc: you write a usual LaTeX document with some input commands specific to gmdoc, usually `\DocInput{`⟨*file.sty*⟩`}` or `\DocInclude{`⟨*file*⟩`}`; cf. fig. 3. That LaTeX document file is called *the driver* (as in doc).

The text typeset in a roman font belongs to the narration layer, that is, it occurs after some % sign. (As you might guess, the lines are numbered automatically.)

gmdoc meets hyperref

Since I've been into TeX for only some three years, .pdf is a most natural output IMO and pdfε-TeX is the most natural TeX engine (though the marvellous XeTeX may become so soon). So, an obligatory and almost subconscious behaviour is to use the hyperref package.

The sophisticated features of doc, such as automatic indexing of the control sequences and marking them in the margin seemed to me so useful and clever that I implemented them in gmdoc. And the features that I consider as 'naturally hyperlinking' are indeed made hyperlinks: the index entries, the cross-references, the footnotes, and the table of contents entries.

That's the other thing that makes gmdoc superior to doc IMO.

The TOC entries, the footnotes and the cross-references are made hyperlinks by default whenever you use the hyperref package. Therefore these features of gmdoc needed no work of mine (except

`\RequirePackage{hyperref}`).

The fourth thing, hyperlinking of the index entries, did need some care. By default, hyperref wants to make a hyperindex and that's very nice in most cases. But the case of documenting a (La)TeX source is different: The index entries may be of three kinds, two of which are specially formatted, and may be preceded with a source file identifier (here I follow

```
\documentclass[fleqn]{ltugproc}
\def\fileversion{\relax}
\hfuzz4pt
\PrelimDraftfalse
\tolerance990
\pretolerance1450
\input ../lsetup.tex
\setcounter{page}{85}
\parskip0pt plus .4pt
\usepackage{gmdoc}
```

This is a comment written as a separate paragraph. (The code is an excerpt from the source of this document.)

(...)

```
\begin{document}
```

(...)

The output of fig. 2:

```
\def\macrocode{\macro@code
```

Then we take care that all spaces have the same width, and that they are not discarded.

```
   \frenchspacing\@vobeyspaces% maybe an inline comment: Before closing, we need to call
      \xmacro@code. It is this
```

(...)

```
  \DocInput{gmdocEBT.tex}
  And this is an example of a~very long code line. See how is it {broken {%
     at {left {brace {with {a~\% sign as `hyphen' and hang-indented.}}}}}}
```

(...)

Figure 3: An example of use and output at once

the rules set by doc and ltxdoc, which I consider to be a (high) standard).

The need to use special encapsulation commands is obvious and that conflicts with the default |hyperpage encapsulation inserted by hyperref. So the appropriate encapsulations were written and now I dare say the high standard of a three-way[2] indexing of the CSs set by doc, along with the high standard of preceding the entries with the source file identifier when the source consists of several files set by ltxdoc, are wed to hyperref in gmdoc and the marriage is consummated.

Finishing touches

The preceding sections describe the two main ideas of gmdoc. The rest of the bundle I would call finishing touches. And they are many; I'll mention only few of them.

The gmdoc package provides hooks for the beginning and end of the input: `\AtBegInput{`⟨*initial stuff to be added*⟩`}` and `\AtEndInput{`⟨*finishing stuff to be added*⟩`}`. Both use the 'adding to a macro' trick so multiple instances are allowed and accumulate. Both act globally.

But I also needed a hook that would add something only once, to the next input file. Therefore I wrote `\AtBegInputOnce{`⟨*the stuff*⟩`}` hook that defines a macro of a unique name, thanks to

```
\csname...\the\some@count\endcsname
```

and the first thing the meaning of that macro consists of is `\let\this@macro\relax`, if you get what I mean, and then ⟨*the stuff*⟩, of course.

The `\IndexInput` command analogous to doc's homonym is crafted very simply: it consists mostly of the basic `\DocInput`, only the comment char, the code delimiter that is, is declared ⟨*char1*⟩. Since ⟨*char1*⟩ is declared 'invalid' in LaTeX, we don't expect one to be in a source file. Therefore the entire contents of a source file is considered to be the code, and typeset verbatim with its CSs automatically indexed.

[2] ;-) .

In this command there is clearly visible a detail not that clear in 'ordinary' `\DocInput`: we have to put our code delimiter at the end of the input to be sure there are none in the file itself. But we do the same in `\DocInput` since we don't want to require that a source file be ended with `%`.

There are a couple of commands for nicely typesetting CSs and their arguments. They are inspired by doc's analogs, but defined in my own way. For instance, my `\cs{cs}` typesets `\cs` as expected, but also allows an optional argument, `\` by default, that is typeset before its mandatory argument. Thus, you may get `!macro` by writing `\cs[!]{macro}`. Why not just `\verb` or a 'short verb'? Remember, that neither `\verb` nor 'short verb' can be used in an argument of a macro, nor can they be written to a file properly. And `\cs` is robust.

To get ⟨*a meta-symbol*⟩ I took the `\<...>` macro from *The TEXbook* (and mixed it with (ltx)doc's `\meta`). I mean, to get ⟨*a meta-symbol*⟩ you write `\<a~meta-symbol>`.

Moreover, for typesetting `{`⟨*arguments like this*⟩`}`, I defined the `\arg` command my way such that

code	typesets
`$\arg␣x=\pi$`	$\arg x = \pi$
`\arg{arg1}`	{⟨*arg1*⟩}
`\arg[optional]`	[⟨*optional*⟩]
`\arg(pictorial)`	(⟨*pictorial*⟩)

I also repeat a handful of logos provided in doc and add my 'drei Groschen':

`\AmSTeX`	$\mathcal{A}\mathcal{M}\mathcal{S}$-TEX
`\BibTeX`	BIBTEX
`\SliTeX`	SLITEX
`\PlainTeX`	PLAIN TEX
`\Web`	WEB
`\TeXbook`	*The TEXbook*
`\eTeX`	ε-TEX
`\pdfeTeX`	pdfε-TEX
`\pdfTeX`	pdfTEX
`\XeTeX`	XEETEX
`\LaTeXpar`	(LA)TEX
`\ds`	DocStrip

The first `E` in XEETEX is reversed if the graphics package is loaded. The (LA)TEX logo is defined in gmutils and therefore available independent of gmdoc).

I allow for a given source file to be typeset both standalone and as part of a multi-file document (The Great Anthology of My Œuvres for instance ;-) and therefore I provide 'relative' sectioning commands: `\division` and `\subdivision` are `\let` to `\section` and `\subsection` respectively but may be assigned another way in The Anthology.

Since my goal is for gmdoc to support both the standard classes and my favourite mwcls, in gmutils I cheat a bit about the sectioning commands to deal with their optional arguments in both the standard classes and mwcls.

Since I often use the Quasi-Fonts (now renamed and updated in TEX Gyre) in the QX encoding,[3] which doesn't have the ␣ sign and that sign is needed when I wish the spaces in a verbatim environment[4] to be 'visible', I added a hook to be executed (expanded) in every verbatim, after setting the catcodes and font. The contents of this hook, if you declare `\VerbT1`, is

```
\fontencoding{T1}\selectfont
```

so a visible space is typeset despite the general font encoding.

As in doc, you may declare some character(s) as 'short verbatim' and then write e.g. `|\verb␣|` instead of `\verb*+\verb␣+`. In fact, it's not gmdoc.sty which makes it possible but gmverb.sty, so you may use that feature independent of gmdoc.

I prefer shorter markup to longer so to display single lines of code,

```
such␣\as␣\THIS␣one,
```

I redefined `\[` to make it properly typeset a short verbatim and spaces. So, you may type

```
\[|such␣\as␣\THIS␣one,|\]
```

to get the above.

I also wrote a document class to typeset the code in a pretty way, gmdocc.cls. This class is strongly inspired by the ltxdoc class but, again, it's not a mere transcription.

In this article there's not room to discuss all the features of this class so let's look at a sample of output (see next page). Please notice the Latin Modern Typewriter Condensed on the margin (hope you like it as I do).

I could write many more words about what I consider the finishing touches. There are many options, declarations and commands to make documenting of sources as much comfortable as a princess could expect.

Approximately 87.31% of those touches were written to make the gmdoc bundle compatible with doc and ltxdoc, that is, to make gmdoc typeset the LATEX canon of scriptures. And that leads us to the last part of this article.

[3] Why do I use QX? I don't remember, to be honest.

[4] I mean all the verbatim-like commands: not only `verbatim`, but also the 'shortverbs' and the groups for the TEX code in gmdoc.

The \code@delim should be 12 so a space is not allowed as a code delimiter. I don't think it *really* to be a limitation.

And let's assume you do as we all do:

```
\CodeDelim\%
```

We'll play with \everypar, a bit, and if you use such things as the {itemize} environment, an error would occur if we didn't store the previous value of \everypar and didn't restore it at return to the narration. So let's assign a \toks list to store the original \everypar.

```
\newtoks\gmd@preverypar
\newcommand*\settexcodehangi{%
  \hangindent=\verbatimhangindent␣\hangafter=\@ne}%
\@ifdefinable\@@settexcodehangi{\let\@@settexcodehangi=%
      \settexcodehangi}
```

we'll use it in the inline comment case. \verbatimhangindent is provided by the gmverb package and = 3 em by default.

We'll play a bit with \leftskip, so let the user have a parameter instead. For normal text (i.e. the comment):

\TextIndent
```
\newlength\TextIndent
```

I assume it's originally equal to \leftskip, i.e. \z@. And for the TEX code:

\CodeIndent
```
\newlength\CodeIndent
\CodeIndent=1,5em\relax
```

And the vertical space to be inserted where there are blank lines in the source code:

```
\@ifundefined{stanzaskip}{\newlength\stanzaskip}{}
```

I use \stanzaskip in gmverse package and derivatives for typesetting poetry. A computer program code *is* poetry.

\stanzaskip
```
\stanzaskip=\medskipamount
\advance\stanzaskip␣by-.25\medskipamount%
```

to preserve the stretch- and shrinkability.

A vertical space between the commentary and the code seems to enhance readability so declare

```
\newskip\CodeTopsep
\newskip\MacroTopsep
```

And let's set them. For æsthetic minimality[7] let's unify them and the other most important vertical spaces used in gmdoc. I think a macro that gathers all these assignments may be handy.

\UniformSkips \CodeTopsep \MacroTopsep
```
\def\UniformSkips{%
  \CodeTopsep=\stanzaskip
  \MacroTopsep=\stanzaskip
  \abovedisplayskip=\stanzaskip
```

%\abovedisplayshortskip remains untouched as it is 0.0 pt plus 3.0 pt by default.

```
  \belowdisplayskip=\stanzaskip
```

[7] The terms 'minimal' and 'minimalist' used in gmdoc are among others inspired by the *South Park* cartoon's episode *Mr. Hankey The Christmas (...)* in which 'Philip Glass, a Minimalist New York composer' appears in a 'non-denominational non-offensive Christmas play' ;-) . (Philip Glass composed the music to the *Qatsi* trilogy among others)

Figure 4: A sample of gmdocc output

Testing or Missa papae Marcelli

In the 16th century there was a controversy in the Roman Catholic Church about polyphony. There is a legend that Pope Marcellus II considered banning it since many composers were making the texture of their works so complex that the words were not recognizable. Then Giovanni Pierluigi da Palestrina wrote a beautiful and ingenious polyphonic Missa whose texture is extremely dense but the words are very clearly recognizable. That missa, dedicated to the pope, convinced him to allow polyphony in the church music.

Why do I write about this? Because I hope the gmdoc bundle at least generates a controversy whether to use the doc package and the ltxdoc class or *itself*. To be honest, my hope is the gmdoc bundle could replace doc and ltxdoc. In a sense, gmdoc is compatible with them: it typesets 'traditional' .dtx files including The LaTeX 2ε Source.

One has just to use `\OldDocInput` instead of `\DocInput` or declare `\olddocIncludes` before `\DocInclude` of a docish file.

The (working!) driver files for The Source and some other canonical files are my Missa papae Marcelli.

First, an homage to doc and ltxdoc, from which I took most of the ideas (although, as a rule, I didn't copy the macros but rather made mine do what they do): doc_gmdoc.tex.

My esteem for those packages and classes is so deep that I didn't report either of the two typos noticed during my typesetting nor did I change the original text, but wrote some 'diving hooks' to fix them.

Then, for their close relative, docstrip.dtx: docstrip_gmdoc.tex.

And, last and most thrilling, The LaTeX 2ε Source: source2e_gmdoc.tex.

Those drivers are available on CTAN as a part of the gmdoc bundle.

I hope this humble bundle will be useful for someone else and not only for me.

Brave new version 0.99g

While preparing this article for *TUGboat*, I revised the gmdoc bundle and made it work with XeTeX and automatically detect a couple of definitions.

'Works with XeTeX' means that you can specify the `sysfonts` option of the gmdocc document class; the basic three XeTeX-related packages (fontspec, xunicode and xltxtra) will be loaded, and then you can specify the system fonts with the fontspec package declarations.

'Automatically detects a couple of definitions' means that if you use gmdoc with its default settings, any occurrence (in the code layer) of the defining commands listed below causes marking of their argument (the thing being defined) as defined at that point: the control sequence, environment, counter or option being defined appears in a margin note and is indexed as a 'definition' entry.

The detected commands are:

- the (La)TeX standard definitions: `\def`, `\newcount`, `\newdimen`, `\newskip`, `\newif`, `\newtoks`, `\newbox`, `\newread`, `\newwrite`, `\newlength`, `\newcommand(*)`, `\renewcommand(*)`, `\providecommand(*)`, `\DeclareRobustCommand(*)`, `\DeclareTextCommand(*)`, `\DeclareTextCommandDefault(*)`, `\newenvironment(*)`, `\renewenvironment(*)`, `\DeclareOption(*)`, `\newcounter`;
- the definitions of the xkeyval package: `\define@key`, `\define@boolkey`, `\define@choicekey`, `\DeclareOptionX`;
- and the option definitions of the kvoptions package by Heiko Oberdiek: `\DeclareStringOption`, `\DeclareBoolOption`, `\DeclareComplementaryOption`, `\DeclareVoidOption`.

Moreover, if you have your own defining commands, they can now be detected with `\DeclareDefining`⟨*command*⟩. On the other hand, you can turn off the detection with `\HideDefining`⟨*command*⟩ for the ⟨*command*⟩ only or `\HideAllDefining` for all the definitions.

There are further commands that allow resuming detection after 'hiding' it and particular declarations for `\def` since it does not always define an important macro.

And you still have the `\Define` declaration and the `macro(*)` environment if the automatic detection doesn't fit your needs.

Concluding, the gmdoc bundle now makes possible typesetting of (La)TeX sources with almost no markup and with the advantages of hyperref and XeTeX.