

makedoc—Preprocessing documentation by T_EX*

Uwe Lück[†]

January 26, 2011

Abstract

makedoc provides commands for generating L^AT_EX input from a package file in order to typeset the latter’s documentation (somewhat similar and opposite to `docstrip`)—with v0.3 *a single one usually suffices*. Certain comment marks are removed, listing commands are inserted, and some (configurable) typographical `txt`→T_EX corrections are applied.—This continues the policy of `niceverb` to minimize documentation markup in package files. `makedoc` extends and exemplifies the parsing package `ffinddo`. After an edit (and test) of your package, you get the new documentation in one run (or the usual number of runs) of the documentation driver file.—The present approach is meant to be an *alternative* to the standard `doc` package and its `\DocInput`. It provides *less* than `doc` does, rather deliberately. It may be helpful at least for the development of small packages, or at least at early stages.

Contents

1	Introduction	2
2	Prior work and what is new	2
3	Styles supported (parsers provided)	3
3.1	Telling code from comments	4
3.2	Markup in comments	4
4	Requirements	5
5	Using <code>makedoc</code> the simplest way	5
6	Steps of advanced usage	6
6.1	Different main parsers (second mandatory argument)	6
6.2	Different extensions (optional arguments)	6
6.3	Commands modifying <code>\MakeInputJobDoc</code> ’s behaviour	7

*This document describes version [v0.41](#) of `makedoc.sty` as of 2011/01/25.

[†]<http://contact-ednotes.sty.de.vu>

1	INTRODUCTION	2
6.4	Separating preprocessing from typesetting	7
6.5	Other <code>makedoc</code> (and <code>fifindo</code>) script commands	8
6.5.1	Choosing parameter values for next preprocessing run	8
6.5.2	“Manual” insertions to the output file	9
6.5.3	Processing input and closing output	9
7	Examples	10
7.1	<code>nicetext</code> , especially <code>mdoccorr.cfg</code>	10
7.2	Packages from other authors	14
8	Implementation	15
8.1	Preliminaries	15
8.2	<code>\MakeDocCorrectHook</code> (“ <code>txt2TeX</code> ”)	16
8.3	Distinguishing package code from comments	16
8.4	Choice of package code environment	18
8.5	Dealing with comments	19
8.6	Sectioning	19
8.7	Commented code	21
8.8	Dealing with empty input lines	22
8.9	Bundling typical things: script commands	22
8.9.1	Output file and <code>\filelist</code> entry	22
8.9.2	Choose input file and <code>run!</code>	22
8.9.3	Preamble vs. main part of input file	23
8.9.4	Screen messages	24
8.9.5	Bundling-bundling Standalones	24
8.10	Leave the package	26
8.11	VERSION HISTORY	26

1 Introduction

The abstract will not be repeated in this section. Let me add instead that I was in dire need of such a package, I got stuck with my packages because I lost orientation in them, and I was unhappy with the forms of documentations of my other packages, and documenting them with the standard \LaTeX `doc` system was not attractive for me (neither considered helpful). I also worked on *Windows* until September 2008, and I find a system like the present one still more attractive then using (learning!) other filtering utilities (see below on `awk`). And I may work on *Windows* once again and don’t want to depend on installing some . . . there—I really would like to have powerful tools for everything depending on nothing but $TeX/LaTeX$!

2 Prior work and what is new

It is, of course, not a new idea to get around comment marks `%` to typeset the documentation. `doc`’s `\DocInput` does this by making `%` an “ignored” character.

This way you cannot use % for commenting comments (so `doc` offers a “new comment mark” `^^A`). You also cannot use % for commenting out code (that you are pondering—or using for debugging—only).

Moreover, `doc` requires enclosing package code explicitly by environment commands (behind comment marks). Stephan I. Böttcher with his `lineno.sty` and Grzegorz Murzynowski in `gmdoc` aimed at doing away with this requirement. `lineno.sty` contains `awk` scripts to remove starting comment marks and to insert listing commands. A file `lineno.tex` is generated that typesets the documentation. By the way, `lineno.sty` is full of discussions, but it is not `docstripped`—the maintainers never have received a complaint that inputting `lineno.sty` were too slow.

`gmdoc` seems to get around comment marks and insert listing commands *while typesetting* by a refined version of `\DocInput`, through some careful detecting and analysing comment marks, the approach resembles detection of lists in `wiki.sty`.¹ And this is a matter of principles—comparing the approaches of *preprocessing* (`lineno.sty`) and *“smart typesetting”* (`gmdoc`, `wiki`). Sometimes preprocessing seems to be simpler, sometimes detecting while typesetting. (Another example is the preprocessor `easylatex` of which `wiki.sty` is a much reduced “while typesetting” variant.) “While typesetting” may be easier when single characters or sequences of two or three encode markup information—but such detection can badly interfere with other packages etc. “Preprocessing” may be easier when entire “strings” of characters decide, which may be anywhere in a file line.

`makedoc` chooses *preprocessing*, as `lineno.sty`, but by `TEX`. There is a general discussion of this choice in the documentation of `fifinddo`. Preprocessing here can be done in the same `LATEX` run as typesetting, though you can avoid incompatibilities with packages needed for typesetting (by inputting them only *after* preprocessing).

`lineno.sty` exemplifies why preprocessing with `TEX` may be preferable to preprocessing with other utilities: When I took over maintenance of `lineno.sty`, I needed hard work to get the `awk` script running. The *Munich* `awk` seemed *not* to behave as the *Kiel* `awk` (I chose a Munich `nawk` and reworked the script a little). `TEX` seems to have better fixed functionality than other utilities!

A different alternative to `LATEX`’s `doc` system is Paul Isambert’s `CodeDoc` where the code environments extract package code in typesetting the documentation.

3 Styles supported (parsers provided)

We find different styles of documenting `LATEX` packages. As the main aspects I consider (i) *telling code from comments* and (ii) *markup in comments*. (You may find more details on the next matters in the “implementation” section.)

¹See `gmdoc.pdf` on `\DocInput`. You can learn a lot from this 220 pages document! I also find `pauldoc` and `xdoc` inspiring.

3.1 Telling code from comments

Comment marks (usually ‘%’ in the case of $\text{T}_{\text{E}}\text{X}$) probably were named so to mark “*comments*” as opposed to code ... great, but actually, in “daily practice,” they are so handy—and used—for “commenting out” *code*, i.e., *managing code versions* in a simple way: one does not actually want to *delete* code, one might want to use it another time, maybe for debugging ... or to remind of earlier attempts that should not be tried again ...

This is a problem for *high-quality typesetting* of documentation. *Code* should be typeset about as you see it on the *screen*—*monospaced*, this allows structuring by indenting, it is common practice to use a typewriter typeface for this. Real *comments* should be typeset in *high quality* as usual with $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. Little dilemmas therefore occur with “*hidden code*” (“commented-out”). A comment mark starts the line, but obviously it is not really a comment and rather should be typeset like code (and otherwise they may break). Another problem are comments at the *end* of a *code* line. Sometimes they are “real comments” (`gmdoc` supports this style). But sometimes this is only another version of “version management,” code “commented-out.”

I like the style of writing packages described before and use it all the time. I mark “real comments” with *two* adjacent comment marks and an ensuing space to distinguish them clearly from code commented out. *This style is presently the one supported by makedoc as default.* This way only a line starting with `%%` is considered a “real” comment line. The first three characters are removed, and the rest is typeset in high quality. Any other lines are typeset verbatim. The `makedoc parser` doing this has an “identifier” `PPScomment` (“percent, percent, space”). Another identifier `comment` is a placeholder for the comment parser to be used, by default it is an alias for `PPScomment`. Lines just containing `%%` (without the space) may be used to suppress empty code lines preceding section titles and for keeping some visual, relieving space between code and comment lines.

The style I described previously may be considered “unprofessional.” The many $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ packages documented using the `doc/.dtx` system don’t use comment marks for “*commenting-out*”. Or one may mark code commented out by putting no space between the percent mark and the code. With v0.4 of `makedoc`, this style is supported as `PScomment`. You can directly call this as `<main-parser>` as described below, or you can switch to it by

```
\CopyFDconditionFromTo{PScomment}{comment}
```

3.2 Markup in comments

Packages using the `doc/.dtx` system as well as alternative highly developed systems mentioned above use (enhanced) usual $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ syntax for markup of comments. Other packages just use an *ASCII* style *without* any markup. My idea was to support the latter style by some `txt`→ $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ functionality. `makedoc` does this using a file `mdoccorr.cfg` which is very small right now.

I also thought of introducing another sort of “decent” markup not needing much more space than the “ASCII kernel” of the comments. This is to some extent implemented in `niceverb.sty`. I thought of the syntax of editing *Wikipedia* pages; this is partially implemented in `wiki.sty` which unfortunately is not yet compatible with `niceverb`.

But `makedoc` implements one *Wikipedia* feature in a different way than `wiki.sty` (cf. `wikicheat.pdf`) that looks about as follows:

```
%%_==_Section_==
%%_===_Subsection_===
%%_====_Subsubsection_====
```

i.e., you type `==<title>_==` in place of `\section{<title>}` etc. The parser must replace `====<title3>====` before `===<title2>===` and the latter before `==<title1>==`. In fact, `makedoc` provides three parsers for these situations:

`\SectionLevelThreeParseInput` is the most general parser offered. If it does not find two strings ‘====’ enclosing *something*, it passes to

`\SectionLevelTwoParseInput` which unless finding two strings ‘===’ enclosing something passes to

`\SectionLevelOneParseInput` ... passes to the comment detector `comment`.

4 Requirements

`makedoc` requires $\text{\LaTeX} 2_\epsilon$ (supporting star forms of `\newcommand` etc.) as \TeX -format, the package `fifinddo.sty` from the same directory (on CTAN etc.) as where `makedoc.sty` is, and the \LaTeX -package `moreverb` by Robin Fairbairns (after others)—it should be installed anyway, or you can get its latest version (v2.3, 2008/06/03?) from CTAN.

`makedoc`’s `.txt`→ \TeX functionality moreover needs a file `mdoccorr.cfg` that should have come along with `makedoc.sty` and `fifinddo.sty`. You may need to have a modified copy of it in the directory of your main `.tex` file `<jobname>.tex` fitting special needs of your project.

5 Using `makedoc` the simplest way

In the most simple case, you are preparing documentation for a package file `<jobname>.sty` only, and you prepare a file `<jobname>.tex` containing

```
\title{\textsf{<jobname>}---a\LaTeX\Package_for_<whatever>}
```

and `\maketitle` etc. about your package.² The documentation will be produced by running `<jobname>.tex` with \LaTeX (e.g., `latex <jobname>.tex`).

²With `niceverb` and `\title` after `\begin{document}`, you may replace ‘`\textsf{<jobname>}`’ by ‘`<jobname>`’.

First, $\langle jobname \rangle.tex$ must have `\usepackage{makedoc}` in its preamble. There are no package options.

Second, to typeset the commented implementation from $\langle jobname \rangle.sty$, include in $\langle jobname \rangle.tex$'s `document` environment a line

```
\MakeInputJobDoc{\langle header-lines \rangle}{\SectionLevelThreeParseInput}
```

$\langle header-lines \rangle$ refers to a non-negative integer as follows: We think the most simple and useful way of typesetting the first lines of a package file including license and copyrights is “depicting them as image,” i.e., *verbatim*. We could try to determine the number of these lines by parsing, but we won't do so soon. Please just count them and enter the number as $\langle header-lines \rangle$ —and change it until you can accept the outcome.

6 Steps of advanced usage

6.1 Different main parsers (second mandatory argument)

`\MakeInputJobDoc`'s mandatory syntax actually is

```
\MakeInputJobDoc{\langle header-lines \rangle}{\langle main-parser \rangle}
```

$\langle main-parser \rangle$ refers to the parsing macro that is applied to each input line whose number is greater than $\langle header-lines \rangle$. Examples for $\langle main-parser \rangle$ are named in section 3 above. `\SectionLevelThreeParseInput` is just the most general one. For *efficiency* (!? or also to avoid problems?) you may replace `Three` by `Two` or by `One`, if the `====` or the `===` feature is not used in $\langle jobname \rangle.sty$. If the “*Wikipedia* sectioning” feature is not used at all, use

```
\MakeInputJobDoc{\langle header-lines \rangle}{\ProcessInputWith{comment}}
```

—provided you want to adopt the ‘%_␣’ style of marking comments, cf. section 3. For the ‘%_␣’ style instead, use

```
\MakeInputJobDoc{\langle header-lines \rangle}{\ProcessInputWith{PScomment}}
```

6.2 Different extensions (optional arguments)

If your package to be documented is a *class* $\langle jobname \rangle.cls$, a local configuration file $\langle jobname \rangle.cfg$ or something else— $\langle jobname \rangle.\langle ext-in \rangle$, e.g., $\langle ext-in \rangle=cls$ or $\langle ext-in \rangle=cfg$, use

```
\MakeInputJobDoc[\langle ext-in \rangle]{\langle header \rangle}{\langle parser \rangle}
```

Moreover, `\MakeInputJobDoc` writes an intermediate file $\langle jobname \rangle.doc$ and then `\inputs` it. If you do not like `doc` as extension for the written file name

(maybe you use `<jobname>.doc` for something different already), preferring extension `<ext-out>`, use

```
\MakeInputJobDoc[<ext-in>][<ext-out>]{<header>}{<parser>}
```

Yes, you must state `<ext-in>` then as well, I can't help ...

If even `<jobname>` is wrong in your view, see next step ...

6.3 Commands modifying `\MakeInputJobDoc`'s behaviour

Already `<jobname>` may not be what you want. E.g., you may want to collect documentations of some other files `<job-1>`, `<job-2>`, ... in a single `<jobname>`. Then precede `\MakeInputJobDoc` with

```
\renewcommand*{\mdJobName}{<job-1>}
```

etc. (please reason yourself about additional requirements ...) As a matter of fact, `\MakeInputJobDoc` reads

```
\mdJobName.<ext-in> and writes \mdJobName.<ext-out>
```

Stated another way, `<jobname>` above referred to `\mdJobName`.

`\MakeInputJobDoc` moreover (by default) produces one dot per input line processed on screen to show progress. The reason is that `makedoc` issues the command `\ProcessLineMessage{\message{.}}`. Already this trivial thing seems to slow down processing considerably (nowadays). `\MakeInputJobDoc` will run faster if preceded by

```
\ProcessLineMessage{}
```

which will suppress any message about processing. However, the message may be helpful in trouble-shooting.

6.4 Separating preprocessing from typesetting

To some surprise, I observe that `\MakeInputJobDoc` *works*. This is quite a new discovery of mine (2010/03/13); before I thought that, for safety, preprocessing should happen inside a local group *preceding* `\documentclass`. `\MakeJobDoc` works like `\MakeInputJobDoc` described above, yet it just *preprocesses* the package to be documented, waiting for an

```
\input{<jobname>.<ext-out>}
```

in the `document` environment to *typeset* the documentation. So `makedoc.tex` once had in its preamble

```
{\RequirePackage{makedoc}  
_ \ProcessLineMessage{}
```

```
\MakeJobDoc{22}{\ProcessInputWith{comment}}
\documentclass{article}
```

I did experience some truth in my earlier safety policy: With niceverb “running,” `\MakeJobDoc` cannot (easily) be used in the `document` environment. `\MakeInputJobDoc` in fact does some niceverb switching (provided niceverb has been loaded) when making use of `\MakeJobDoc`.

Thinking of this “safety” approach, especially grouping (`{\code}`), I had not much cared for compatibility with other packages in choosing `makedoc` macro names.

6.5 Other `makedoc` (and `fifindo`) script commands

The next script commands may be considered of a lower level than `\MakeJobDoc` and `\MakeInputJobDoc`, they underlie the latter commands. We also list commands from `fifinddo.sty` that may be useful or, indeed, are needed for preparing package documentations. This may result in ideas on how do use the script commands for different purposes than for preparing package documentations—e.g., apply `txt`→`TEX` preprocessing to arbitrary text files.

6.5.1 Choosing parameter values for next preprocessing run

This actually continues section 6.3.

`\ResultFile{<output>}` (from `fifinddo`) determines (and opens through the `TEX` primitive `\openout`) the file `<output>` which will contain the result of preprocessing the package file.

`\LaTeXresultFile{<output>}` —see next section.

`\Headerlines{<number>}` determines the `<number>` of lines starting the input file to be copied *verbatim* (the first mandatory argument of `\MakeJobDoc`).

`\MainDocParser{<parser>}` determines the `<parser>` as in the *second* mandatory argument of `\MakeJobDoc`.

We are now describing some parameters which rather must be switched “manually” instead of being modifiable by comfortable `makedoc` script commands.

With the “*Wikipedia sectioning*” feature, you may change the outcome regarding levels. Assume, e.g., the package file has titles along the scheme `==<title>==` only, but these should become *subsections* of the “implementation” section of the corresponding `.tex` file. Then

```
\renewcommand*{\mdSectionLevelOne}{\string\subsection}
```

– see the implementation of the sectioning feature for details.

There is a command

`\NoEmptyInputLines` and a parameter macro `\OnEmptyInputLine`

which is modified by the former. However, I cannot say much about them right now, I think they just were a difficult matter that I soon decided no longer to think about for a while (cf. the implementation). About the same holds for the hook `\EveryComment`.

The `txt`→`TeX` functionality is implemented through a hook

`\MakeDocCorrectHook{<characters>}`

`makedoc` initializes it as an alias of L^AT_EX’s `\@firstofone`, i.e., it won’t change `<characters>`. `mdocorr.cfg` should redefine it so it really “corrects” `<characters>`. You might try other definitions of `\MakeDocCorrectHook` for different “correcting” functions. It should be *noted* that (currently) `\MakeDocCorrectHook` must be *expandable*, `fifinddo.sty` provides setup for (expandable) chains of expandable replacements. The “Wikipedia” sectioning feature moreover uses expandable trimming (single) surrounding spaces, this might be provided in a more general way.³

6.5.2 “Manual” insertions to the output file

`\WriteResult{<balanced>}` (from `fifinddo`) writes `<balanced>` to `<output>` according to the earlier command `\ResultFile{<output>}`.

`\WriteProvides` (from `fifindo`) writes a `\ProvidesFile` line into `<output>` that declares the file to be generated by `fifindo`.

`\LaTeXresultFile{<output>}` issues `\ResultFile{<output>}` and then writes a `\ProvidesFile` line into `<output>` that declares the file to be generated by `makedoc`.

6.5.3 Processing input and closing output

`\MakeDoc{<input>}` reads `mdocorr.cfg` (for `\MakeDocCorrectHook`, see above), copies `<number>` according to `\HeaderLines` (see above) from `<input>` into `<output>` (according to `\ResultFile`), then processes the remaining lines of `<input>` according to `\MainDocParser` (writing several things to `<output>`). `\MakeDoc` invokes

`\ProcessFileWith{<input>}{<loop-body>}` (from `fifindo`) reads `<input>` line by line—each one stored as macro `\fdInputLine` and applies `<loop-body>` to it. T_EX’s “special” character codes (of characters listed in macro `\dospecials`) are replaced by 12 (“other”) by default—see the `fifinddo` documentation.

³The `trimspaces` package has been a *model* for this feature here. It cannot be used directly here because it reads blank spaces as T_EX characters with category code 10 while `makedoc` reads blank spaces as “other” characters (category code 12) in order to *keep* all blank spaces.

`\CloseResultFile` (from `fifinddo`) closes $\langle output \rangle$ (using \TeX 's primitive `\closeout`).

`\MakeCloseDoc{\langle input \rangle}` issues `\MakeDoc{\langle input \rangle}\CloseResultFile`.

Using `\MakeDoc` instead of `\MakeCloseDoc` allows processing additional $\langle input \rangle$ files writing into the same $\langle output \rangle$. Or maybe you want to add some additional lines manually to $\langle output \rangle$ using `\WriteResult`.

7 Examples

7.1 nicetext, especially mdoccorr.cfg

The documentations of `fifinddo`, `makedoc`, and `niceverb` themselves are typeset using `makedoc`. `fifinddo.pdf` documents `fifinddo.sty`, typeset from `fifinddo.tex`, likewise `makedoc.pdf` and `niceverb.pdf`. The Wikipedia syntax feature

```
%%_===_subsection_===
```

is used in `fifinddo.sty` and `niceverb.sty` only.

Along with `makedoc` should come files `makedoc.tpl`—a template `makedoc` script, and a file `fdtxttex.tex` that should start a dialogue on trying `\MakeDocCorrectHook` if you can manage to run it (WinShell?). With other definitions of `\MakeDocCorrectHook`—see below—you can use this dialogue for arbitrary replacement jobs (as an application of `fifinddo` rather than `makedoc`).

`fifinddo.pdf`, `makedoc.pdf`, and `niceverb.pdf` were typeset with the following typographical corrections in `mdoccorr.cfg` that defines `\MakeDocCorrectHook`:

```
1  \ProvidesFile{mdoccorr.cfg}[2011/01/12
      local typographical corrections
      with makedoc.sty]
    %% ... also demonstrates 'niceverb.sty'---see the typeset
5  %% documentation of the present file in 'makedoc.pdf'.
    %%
    %% |\SetPatternCodes{<commands>}| redefines
    %% '\PatternCodes' to be used in parsing and replacing
    %% (some 'sanitizing'). %% improved line breaks 2010/03/29
10 \SetPatternCodes{\MakeOther\\MakeOther\ }
    %% |\StartPrependingChain| initializes setup of a replacement
    %% chain:
    \StartPrependingChain
    %% |\PrependExpandableAllReplacer{<id>}{<find>}{<subst>}|:
15 \PrependExpandableAllReplacer{etc}{etc. }{etc.\ }
    %% ... you can keep inter-sentence space after 'etc.'
    %% by a code line break.
    \PrependExpandableAllReplacer{Cf}{Cf. }{Cf.\ } %% 2011/01/12
    \PrependExpandableAllReplacer{cf}{cf. }{cf.\ } %% corr. 2010/03/23
```

```

20 %% ... but think of ‘cf.~’. Don’t leave ‘cf.’ at code line end!
   \PrependExpandableAllReplacer{rarr}{->}{$\to$}      %% 2010/11/06
   \PrependExpandableAllReplacer{larr}{<-}{$\gets$}    %% 2010/11/06
   % \PrependExpandableAllReplacer{dots}{...}{$\dots$}
   %% ... 2010/11/24 ‘symmetric’ variant of ‘\textellipsis’:
25 %% allow extra space at line end.
   \newcommand*{\PXAR}{\PrependExpandableAllReplacer}
   \PXAR{dots}{...}{.\kern\fontdimen3\font.\kern\fontdimen3\font.}
   \PXAR{dots }{... }{.\kern\fontdimen3\font.\kern\fontdimen3\font.\ }
   %% ... chain starts here, and here |\MakeDocCorrectHook| (silently)
30 %% enters through a default variant |\SetCorrectHookJobLast|
   %% of |\SetCorrectHookJob{<id>}|:
   \SetCorrectHookJobLast
   %%
   %% |\ResetPatternCodes| resets ‘\PatternCodes’ to their default
35 %% value (which is ‘\fdPatternCodes’):
   \ResetPatternCodes
   %% ... restores ‘fifinddo’ default.
   \endinput

40 HISTORY
   2009/04/05 with makedoc v0.2
   2010/03/11 broke some too long code lines
   2010/03/16 rendered ‘mdoccorr.cfg’
   2010/03/22 try \Prepend...
45 2010/03/23 corrected ‘cf’
   2010/03/29 use \SetPatternCodes etc.
   2010/11/06 <- and ->
   2010/11/24 ‘...’ ‘symmetric variant’ of ‘\textellipsis’;
               \StartPrependingChain, no more \MakeExpandableAllreplacer

```

This code also exemplifies the syntax `niceverb` provides for writing about \LaTeX macros. It is typeset here with `makedoc.sty` and then looks thus:

```

1 \ProvidesFile{mdoccorr.cfg}[2011/01/12
2     local typographical corrections
3     with makedoc.sty]

```

... also demonstrates `niceverb.sty`—see the typeset documentation of the present file in `makedoc.pdf`.

`\SetPatternCodes{<commands>}` redefines `\PatternCodes` to be used in parsing and replacing (some “sanitizing”).

```

4 \SetPatternCodes{\MakeOther\\MakeOther\ }

```

`\StartPrependingChain` initializes setup of a replacement chain:

```

5 \StartPrependingChain

```

```
\PrependExpandableAllReplacer{<id>}{<find>}{<subst>}:

```

```
6 \PrependExpandableAllReplacer{etc}{etc. }{etc.\ }

```

... you can keep inter-sentence space after etc. by a code line break.

```
7 \PrependExpandableAllReplacer{Cf}{Cf. }{Cf.\ } %% 2011/01/12

```

```
8 \PrependExpandableAllReplacer{cf}{cf. }{cf.\ } %% corr. 2010/03/23

```

... but think of cf.~. Don't leave cf. at code line end!

```
9 \PrependExpandableAllReplacer{rarr}{->}{$\to$} %% 2010/11/06

```

```
10 \PrependExpandableAllReplacer{larr}{<-}{$\gets$} %% 2010/11/06

```

```
11 % \PrependExpandableAllReplacer{dots}{...}{$\dots$}

```

... 2010/11/24 “symmetric” variant of `\textellipsis`: allow extra space at line end.

```
12 \newcommand*{\PXAR}{\PrependExpandableAllReplacer}

```

```
13 \PXAR{dots}{...}{.\kern\fontdimen3\font.\kern\fontdimen3\font.}

```

```
14 \PXAR{dots }{... }{.\kern\fontdimen3\font.\kern\fontdimen3\font.\ }

```

... chain starts here, and here `\MakeDocCorrectHook` (silently) enters through a default variant `\SetCorrectHookJobLast` of `\SetCorrectHookJob{<id>}`:

```
15 \SetCorrectHookJobLast

```

`\ResetPatternCodes` resets `\PatternCodes` to their default value (which is `\fdPatternCodes`):

```
16 \ResetPatternCodes

```

... restores `fifinddo` default.

```
17 \endinput

```

```
18

```

```
19 HISTORY

```

```
20 2009/04/05 with makedoc v0.2

```

```
21 2010/03/11 broke some too long code lines

```

```
22 2010/03/16 rendered 'mdoccorr.cfg'

```

```
23 2010/03/22 try \Prepend...

```

```
24 2010/03/23 corrected 'cf'

```

```
25 2010/03/29 use \SetPatternCodes etc.

```

```
26 2010/11/06 <- and ->

```

```
27 2010/11/24 '...' 'symmetric variant' of '\textellipsis';

```

```
28 \StartPrependingChain, no more \MakeExpandableAllreplacer

```

```
29

```

And this is the content of the intermediate generated file:

```
1 \ProvidesFile{mdoccorr.doc}[2011/01/26 automatically generated with makedoc.sty]

```

```
\begin{mdPackageCode}

```

```
\ProvidesFile{mdoccorr.cfg}[2011/01/12

```

```
local typographical corrections

```

```

5          with makedoc.sty]
\end{mdPackageCode}
.\kern\fontdimen3\font.\kern\fontdimen3\font.\ also demonstrates 'niceverb.sty'---see the
documentation of the present file in 'makedoc.pdf'.

10 |\SetPatternCodes{<commands>}| redefines
'\PatternCodes' to be used in parsing and replacing
(some "sanitizing"). %% improved line breaks 2010/03/29
\begin{mdPackageCode}
\SetPatternCodes{\MakeOther\\\MakeOther\ }
15 \end{mdPackageCode}
|\StartPrependingChain| initializes setup of a replacement
chain:
\begin{mdPackageCode}
\StartPrependingChain
20 \end{mdPackageCode}
|\PrependExpandableAllReplacer{<id>}{<find>}{<subst>}|:
\begin{mdPackageCode}
\PrependExpandableAllReplacer{etc}{etc. }{etc.\ }
\end{mdPackageCode}
25 .\kern\fontdimen3\font.\kern\fontdimen3\font.\ you can keep inter-sentence space after 'e
by a code line break.
\begin{mdPackageCode}
\PrependExpandableAllReplacer{Cf}{Cf. }{Cf.\ } %% 2011/01/12
\PrependExpandableAllReplacer{cf}{cf. }{cf.\ } %% corr. 2010/03/23
30 \end{mdPackageCode}
.\kern\fontdimen3\font.\kern\fontdimen3\font.\ but think of 'cf.~'. Don't leave 'cf.' at
\begin{mdPackageCode}
\PrependExpandableAllReplacer{rarr}{->}{$\to$} %% 2010/11/06
\PrependExpandableAllReplacer{larr}{<-}{$\gets$} %% 2010/11/06
35 % \PrependExpandableAllReplacer{dots}{...}{$\dots$}
\end{mdPackageCode}
.\kern\fontdimen3\font.\kern\fontdimen3\font.\ 2010/11/24 "symmetric" variant of '\text
allow extra space at line end.
\begin{mdPackageCode}
40 \newcommand*{\PXAR}{\PrependExpandableAllReplacer}
\PXAR{dots}{...}{.\kern\fontdimen3\font.\kern\fontdimen3\font.}
\PXAR{dots }{... }{.\kern\fontdimen3\font.\kern\fontdimen3\font.\ }
\end{mdPackageCode}
.\kern\fontdimen3\font.\kern\fontdimen3\font.\ chain starts here, and here |\MakeDocCorre
45 enters through a default variant |\SetCorrectHookJobLast|
of |\SetCorrectHookJob{<id>}|:
\begin{mdPackageCode}
\SetCorrectHookJobLast
\end{mdPackageCode}
50 |\ResetPatternCodes| resets '\PatternCodes' to their default

```

```

value (which is '\fdPatternCodes'):
\begin{mdPackageCode}
\ResetPatternCodes
\end{mdPackageCode}
55 .\kern\fontdimen3\font.\kern\fontdimen3\font.\ restores 'fifinddo' default.
\begin{mdPackageCode}
\endinput

HISTORY
60 2009/04/05 with makedoc v0.2
2010/03/11 broke some too long code lines
2010/03/16 rendered 'mdoccorr.cfg'
2010/03/22 try \Prepend...
2010/03/23 corrected 'cf'
65 2010/03/29 use \SetPatternCodes etc.
2010/11/06 <- and ->
2010/11/24 '...' 'symmetric variant" of '\textellipsis';
\StartPrependingChain, no more \MakeExpandableAllreplacer

70 \end{mdPackageCode}

```

7.2 Packages from other authors

`substr.tex` should typeset a nicely formatted documentation of Harald Harder's `substr.sty`, see my own result `substr.pdf`. `substr.sty` is a rare case of the '%_␣' commenting style that `nicetext` has used itself.

`arseneau.tex` should typeset nicely formatted documentations of a few packages by Donald Arseneau, see my own result `arseneau.pdf`. This demonstrates the usual '%_␣' commenting style that `makedoc` supports with v0.4.

8 Implementation

8.1 Preliminaries

Head of file (Legalese):

```

1  %% Macro package ‘makedoc.sty’ for LaTeX2e,
2  %% copyright (C) 2009–2011 Uwe L"uck,
3  %%   http://www.contact-ednotes.sty.de.vu
4  %% -- author-maintained in the sense of LPPL below --
5  %% for preparing documentations from packages.
6
7  \def\fileversion{0.41} \def\filedate{2011/01/25}
8
9  %% This file can be redistributed and/or modified under
10 %% the terms of the LaTeX Project Public License; either
11 %% version 1.3a of the License, or any later version.
12 %% The latest version of this license is in
13 %%   http://www.latex-project.org/lppl.txt
14 %% We did our best to help you, but there is NO WARRANTY.
15 %%
16 %% Please report bugs, problems, and suggestions via
17 %%
18 %%   http://www.contact-ednotes.sty.de.vu
19 %%
20 \NeedsTeXFormat{LaTeX2e}[1994/12/01]
21 % 1994/12/01: \newcommand* etc.
22 \ProvidesPackage{makedoc}[\filedate\space v\fileversion\space
23   TeX input from *.sty (UL)]

```

`\PackageCodeTrue` and `\PackageCodeFalse` set `\ifPackageCode` globally, so redefinition of `\sim` (playing a key role in `fifinddo`) may be kept local. Note the capital T and F!

```

24 \newcommand*{\PackageCodeTrue} {\global\let\ifPackageCode\iftrue}
25 \newcommand*{\PackageCodeFalse}{\global\let\ifPackageCode\iffalse}

```

`\ifPackageCode` is used to determine whether a listing environment must be `\begun` or `\ended`. You may also want to suppress empty code lines, while empty lines should issue a `\par` break in “comment” mode.

Since `\newif` is not used, `\ifPackageCode` must be declared explicitly. Declaration of new `\ifs` must be early in case they occur in code that is skipped by another `\if...` [TODO!? cf. others 2010/03/15]

```

26 \PackageCodeFalse

```

`makedoc` is an extension of `fifinddo` on which it depends.

```

27 \RequirePackage{fifinddo}[2009/04/13]

```

Both `fifinddo` and `makedoc` use the “underscore” `_` as “private letter” and restore its standard “subscript” function (*TEXbook Chap. 7*) at their end. Push/pop

functionality as with @ and \RequirePackage is missing here. So after loading fifinddo, we need to declare our private letter (again).

```
28 \catcode'\_ =11 %% underscore used in control words
```

8.2 \MakeDocCorrectHook (“txt2TeX”)

`\MakeDocCorrectHook` is predefined to leave its argument without the enclosing braces, otherwise unchanged:

```
29 \let\MakeDocCorrectHook\@firstofone
```

Less efficiently, the same could have been set up as

```
30 % \newcommand*\MakeDocCorrectHook}[1]{\ProcessStringWith{#1}{LEAVE}}
```

according to fifinddo.

It may be *redefined* in a *configuration* file like `makedoc.cfg` or the `makedoc` script file applying to a single package file in order to, e.g., converting plain text to T_EX input conforming to typographical conventions, making `$$dots$` from ‘...’, e.g. Replace LEAVE in the previous suggestion by an identifier whose job you have defined before, and use `\renewcommand` in place of `\newcommand`. See an example in `makedoc.cfg`.

You can *test* your own `\MakeDocCorrectHook` by

```
\typeout{\MakeDocCorrectHook{\test-string}}
```

... provided (sometimes) `\MakeOther\` ... You can even change it using `\IfInputLine` from fifinddo in the midst of preprocessing a package documentation.

8.3 Distinguishing package code from comments

Use of comment marks is a matter of personal style. Only lines starting with the sequence `[%\]` are typeset in T_EX quality under the present release. Lines just containing `[%]` (without the space) are used to suppress empty code lines preceding section titles (while keeping some visual space in the package file). There is a preferable way to do this, however not in the present release ...

The parsing macros must be set up reading % and `\` as “other” characters. Using the optional arguments for this creates difficulties that can be somewhat avoided by redefining `\PatternCodes`.

```
31 \SetPatternCodes{\MakeOther\% \MakeOther\ }% 2010/03/30
```

The next line sets the “sandbox” for the detecting macro, as it is coined in the documentation of fifinddo, with “identifier” `PPScomment`.

```
32 \MakeSetupSubstringCondition{PPScomment}{% }{#1}
```

The last argument stores the expanded input line for reference by macros called. The next line is a test whether the setup works.


```
33 % \expandafter \show \csname \setup_substr_cond PPScomment\endcsname
```

Here comes the definition of the corresponding testing macro. #3 is the expanded input line from above. The `\If...` commands, `\fdInputLine`, `\fdInputLine`, and `\RemoveDummyPatternArg` are from `fifinddo`.

```
34 \MakeSubstringConditional{PPScomment}{%% }#3{%% #3 entire test string
35 \DecideCommentCode{#1}{#2}{#3}\PPstring}
```

`\DecideCommentCode{#1}{#2}{#3}{#4}` is shared with the parser for the ‘%_␣’ commenting style.

```
36 \newcommand*{\DecideCommentCode}[4]{%
37 \IfFDinputEmpty{\OnEmptyInputLine}{%
```

The empty line test comes early to offer control with `\OnEmptyInputLine` both in code and comment mode. Maybe it should always?

```
38 \IfDempty{#1}%
39 \TreatAsComment{%
40 \RemoveDummyPatternArg\MakeDocCorrectHook{#2}}}%
41 {\ifx\fdInputLine#4%
42 \ifPackageCode\else \WriteResult{}\fi%
```

... allows paragraphs in comments.

```
43 \else \TreatAsCode{#3}\fi}}
```

Job `PPScomment` deals with the ‘%_␣’ commenting style:

```
44 \MakeSetupSubstringCondition{PPScomment}{% }{#1}}
45 \MakeSubstringConditional{PPScomment}{% }#3{%
46 \DecideCommentCode{#1}{#2}{#3}\PercentChar}
```

`\PercentChar` is from `fifinddo`.—Return to `\fdPatternCodes`:

```
47 \ResetPatternCodes %% 2010/03/30
```

`\PPstring` stores the line suppressing empty code lines with the ‘%_␣’ style:

```
48 \newcommand*{\PPstring}{\xdef\PPstring{\PercentChar\PercentChar}}
```

`comment` will be a “generic” identifier to call a comment line detector. It might be predefined to issue an “undefined” error; however this release predefines it to behave like `PPScomment`:

```
49 \CopyFDconditionFromTo{PPScomment}{comment}
```

This means that the ‘%_␣’ commenting style is assumed by default.

```
\CopyFDconditionFromTo{PPScomment}{comment}
```

switches to the ‘%_␣’ style (with the *Wikipedia* sectioning parser). Or just choose

```
\ProcessInputWith{PPScomment}
```

as `\main-parser` (without the *Wikipedia* feature).

Alternative still to be considered:

```

50 % \@namedef{\setup_substr_cond comment}{%
51 %   \PackageError{makedoc}{Job 'comment' not defined}%
52 %   {Use \string\COPYFDconditionFromTo{comment}}}
```

8.4 Choice of package code environment

With v0.3, we adopt the solution for typesetting package code that was implemented in the former `makedoc.cfg`. So we rely on the `listing` and `listingcont` environments of the `moreverb` package.

The earlier idea was that `makedoc.sty` uses an undefined L^AT_EX environment `packagecode` that will be defined in `makedoc.cfg`. An accompanying idea was that `makedoc` works before the `\documentclass` line inside a group, while `makedoc.cfg` is read *after* the `\documentclass` line.

We now want to simplify things. We replace

`packagecode` by `mdPackageCode`

and define the new environment globally here. `moreverb` and our choice for `\listinglabel` are called at `\begin{document}`—outside the possible group.

```

53 \gdef\mdPackageCode{%
54 %   \small
```

2011/01/19, v0.41: `\small` has affected the `\baselineskip` above the code. So a `\par` must be executed before `\small`. But we don't want to have the extra `\partopsep`—forced ...

```

55   \mdStartPackageCode           %% 2011/01/19 v0.41
```

From the next occurrence of the environment onwards, `listing` must be replaced by `listingcont`.

```

56   \gdef\mdPackageCode{\mdStartPackageCode  %% 2011/01/19 v0.41
57   \listingcont}%
58   \listing{1}}
59   \gdef\mdStartPackageCode{%           %% 2011/01/19 v0.41
```

Smart `\small`—we might once allow `\partopsep` in `vmode`—not this time:

```

60   \par \small \partopsep\z@skip
```

Get rid of `niceverb` stuff:

```

61   % \MakeOther\'\MakeOther\'%% probably OK with moreverb
62   \MakeOther\<\MakeOther\|}%
63   }
64   \gdef\endmdPackageCode{%
65   \endlisting
66   \global\let\endmdPackageCode\endlistingcont}
67   \AtBeginDocument{%
68   \RequirePackage{moreverb}%
69   \renewcommand*\listinglabel}[1]{%
70   \llap{\scriptsize\rmfamily\the#1}\hskip\listingoffset\relax}%
71   }
```

`\ResetCodeLineNumbers` may be needed to *reset* `moreverb`'s (global!) code line number settings (*first* line number + *modulo* for displaying) when combining documentations of *more* than one package with the present solution for implementing `mdPackageCode`.

```
72 \ifdefinable\ResetCodeLineNumbers{% global as above, v0.41
73     \gdef\ResetCodeLineNumbers{%
74         \global\listing@line\@ne \gdef\listing@step{\@ne}}
```

8.5 Dealing with comments

`\TreatAsComment{⟨text⟩}` writes `⟨text⟩` to the documentation file. If we had “package code” (were in “code mode”) so far, the listing environment is ended first.

```
75 \newcommand*\TreatAsComment}[1]{%
76     \ifPackageCode
77         \WriteResult{\string\end{mdPackageCode}\@empty}}%
```

The `\@empty` here is a lazy trick to save self-documentation fighting `verbatim`'s “highlight” of finding ends of listings (to be improved).

We always use `\string` to prevent macro expansion in `\write` in place of \LaTeX 's `\protect`, as long as `ffinddo` simply uses the primitive `\write` in place of \LaTeX 's `\protect\write` ...

```
78     \PackageCodeFalse
79     \EveryComment
80 %     \_empty_code_lines_false
81     \fi
82     \WriteResult{#1}}
```

Here, `\EveryComment` is a macro hook for inserting material that should not appear in a listing environment, I had tried this successfully:

```
\gdef\EveryComment{%
    \global\let\EveryComment\relax
    \WriteResult{\string\AutoCmdVerbSyntax}}
```

Initialized:

```
83 \global\let\EveryComment\relax %% should be changed globally.
```

8.6 Sectioning

We provide a facility from `wiki.sty` that imitates the sectioning syntax used in editing *Wikipedia* pages, in a different implementation (better compatibility) and in a more general way. On *Wikipedia*, `==_Definition_==` works similar as `\section{Definition}` does with \LaTeX . With the present implementation, you can type, e.g.,

```
%%_Definition_==_Definition_==_Definition_==_Definition_==_Definition_==_
```

to get a similar result. The number of % characters doesn't matter, and there can be other stuff, however: additional = symbols may harm. Three sectioning levels are supported, through `==<text>==`, `===<text>===`, and `====<text>====` (deepest).

There are three detector macros made for programmers. The most general one is in the following definitions, there is a single tilde to prevent = symbols being gobbled by the test (realized by accident).

`\SectionLevelThreeParseInput`:

```
84 \newcommand*{\SectionLevelThreeParseInput}{%
85   \expandafter \test_sec_level_iii \fdInputLine ~=====&}
```

`\SectionLevelTwoParseInput`

```
86 \newcommand*{\SectionLevelTwoParseInput}{%
87   \expandafter \test_sec_level_ii \fdInputLine ~=====&}
```

and `\SectionLevelOneParseInput`

```
88 \newcommand*{\SectionLevelOneParseInput}{%
89   \expandafter \test_sec_level_i \fdInputLine ~====&}
```

allow skipping deeper levels for efficiency.

In the terminology of the `fifinddo` documentation, the previous three commands are “sandbox builders.” The following three commands are the corresponding “substring conditionals.” However, `fifinddo` so far only deals with *single* substrings, while here we are dealing with *pairs* of substrings. We are not using general setup macros, but define the parsing macros “manually,” as it is typical in many other macros in `latex.ltx` and other \LaTeX packages. You can fool our macros easily, there is no syntax check.

```
90 \def\test_sec_level_iii#1====#2====#3&{%
91   \IfDempty{#2}%
92     {\test_sec_level_ii #1=====&}%
93     {\WriteSection\mdSectionLevelThree{#2}}}
94 \def\test_sec_level_ii#1====#2====#3&{%
95   \IfDempty{#2}%
96     {\test_sec_level_i #1====&}%
97     {\WriteSection\mdSectionLevelTwo{#2}}}
98 \def\test_sec_level_i#1==#2==#3&{%
99   \IfDempty{#2}%
100    {\RemoveTildeArg \ProcessStringWith{#1}{comment}}%
101    {\WriteSection\mdSectionLevelOne{#2}}}
```

`\ProcessStringWith` here passes the expanded `\fdInputLine` to the general comment detector.

`\WriteSection{<command>}{<text>}` replaces an input line with a line

`<command>{<text>}`

in the documentation file and switches into “comment mode.” One possible space between = and the beginning of `<text>` and one possible space between the

end of $\langle text \rangle$ and = are removed. The method of dealing with surrounding blank spaces is new with v0.3, moreover we now rely on a new method in niceverb.sty v0.3 to support its single right quote feature in section titles.⁴

```
102 \newcommand*\WriteSection}[2]{%
103   \TreatAsComment{^^J#1{\trim_correct{#2}}^^J}}
```

Trimming “other” spaces is a little more clumsy than what the trimspaces package does whose code is by Morten Høgholm. It still has inspired the following.

```
104 \begingroup \MakeOther\ %% CARE! we must not indent ...
105 \long\gdef\trim_correct#1{\trim_fosp#1$ $}
106 \long\gdef\trim_fosp#1$ {%
107   \IfFDempty{#1}{\trim_losp$}{\trim_losp#1$ }}
```

So we have a string ‘ $\backslash\text{trim_losp}\langle text \rangle_\square$ ’.

```
108 \long\gdef\trim_losp#1 $ {\tidy_sp_trim#1$}
```

So we have a string ‘ $\backslash\text{tidy_sp_trim}\langle text \rangle_\square$ ’ or ‘ $\backslash\text{tidy_sp_trim}\langle text \rangle_\square$ ’.

```
109 \long\gdef\tidy_sp_trim#1#2$\MakeDocCorrectHook{#1}}
110 \endgroup
```

We insert $\backslash\text{section}$ using $\backslash\text{mdSectionLevelOne}$ etc. which the programmer can redefine, e.g., when the documentation is part of a $\backslash\text{section}$ (or even deeper) according to the “documentation driver” file.

```
111 \newcommand*\mdSectionLevelOne {\string\section}
112 \newcommand*\mdSectionLevelTwo {\string\subsection}
113 \newcommand*\mdSectionLevelThree{\string\subsubsection}
```

This sectioning feature is not used in (the documentation) of makedoc.sty—*definitions* of the parsing macros fool the same macros ...

8.7 Commented code

$\backslash\text{TreatAsCode}\langle text \rangle$ is the opposite to $\backslash\text{TreatAsComment}\langle text \rangle$:

```
114 \newcommand*\TreatAsCode}[1]{%
115   \ifPackageCode
116   %   \empty_code_lines_true
117   \else
118     \WriteResult{\string\begin{mdPackageCode}}%
119     \PackageCodeTrue
120   \fi
121   \WriteResult{#1}%
122   %   \WriteResult{\maybe_result_empty_line #1}%
123   %   \let\maybe_result_empty_line\empty
124 }
```

⁴ $\backslash\text{ignorespaces}$ and $\backslash\text{unskip}$ used previously do not work in PDF bookmarks.

8.8 Dealing with empty input lines

`\OnEmptyInputLine` is a default setting (or hook) for what to do with empty lines in the input file. The default is to insert an empty line in the output file:

```
125 \newcommand*\OnEmptyInputLine{\WriteResult{}}
```

`\NoEmptyCodeLines` changes the setting to suppressing empty code lines, while in “comment mode” an empty input line *does* insert an empty line, for starting a new paragraph:

```
126 \newcommand*\NoEmptyCodeLines{%% suppress empty code lines
127 \renewcommand*\OnEmptyInputLine{%
128 \ifPackageCode \else \WriteResult{}\fi}}
```

There is a better policy—didn’t work so far ...

8.9 Bundling typical things: script commands

Practical experience suggested the following shorthands, combining commands from `makedoc` and `fifinddo`.

8.9.1 Output file and `\filelist` entry

`\LaTeXresultFile{<output>}` chooses `<output>` as name for the output file and saves you the extra line for inserting the `\ProvidesFile` line as with `fifinddo`’s `\WriteProvides`—however, it differs, actually it is `makedoc` that wants to be mentioned with `\ProvidesFile` ...

```
129 \newcommand*\LaTeXresultFile}[1]{%
130 \ResultFile{#1}%% \WriteProvides}
131 \WriteResult{%
132 \string\ProvidesFile{\result_file_name}%
133 [\the\year/\two@digits\month/\two@digits\day\space
134 automatically generated with makedoc.sty]}}%
```

8.9.2 Choose input file and run!

`\MakeDoc{<input>}` preprocesses `<input>` to render input for \LaTeX , considering what is typical for a \LaTeX package as the `<input>` one here:

```
135 \newcommand*\MakeDoc}[1]{%
```

In case of a “header” (see below) we change into “code mode”:

```
136 \ifnum\header_lines>\z@
137 \WriteResult{\string\begin{mdPackageCode}}%
138 \PackageCodeTrue %% TODO both lines makedoc command!?
139 %% 2009/04/08
140 \else \PackageCodeFalse \fi
```

The loop follows. There is a placeholder `\makd_doc_line_body` that is predefined below and can be changed while processing the $\langle input \rangle$ file.

```

141   \ProcessFileWith{#1}{%
142     \CountInputLines %% stepping line counter is standard
143     \make_doc_line_body
144     \process_line_message}%

```

Currently the “VERSION HISTORY” or, more generally, a final part of the $\langle input \rangle$ file is typeset verbatim (for “tabbing” in the version history), so we must leave “code mode” finally:

```

145   \ifPackageCode
146     \WriteResult{\string\end{mdPackageCode\@empty}}}% self-doc-trick
147     \PackageCodeFalse %% TODO both lines makedoc command!? 2009/04/08
148   \fi

```

When the $\langle input \rangle$ file has been processed, certain default settings might be restored—in case another $\langle input \rangle$ file is processed for the same documentation document:

```

149 %   \HeaderLines{0}%
150 %   \MainDocParser{\SectionLevelThreeParseInput}%% TODO!? 2009/04/08
151 }

```

`\MakeCloseDoc{ $\langle input \rangle$ }` just is a shorthand for

$$\backslash\text{MakeClose}\{\langle input \rangle\}\backslash\text{CloseResultFile}$$

where `\CloseResultFile` is from `fifinddo`.

```

152 \newcommand*{\MakeCloseDoc}[1]{\MakeDoc{#1}\CloseResultFile}

```

`\MakeDoc` and `\MakeCloseDoc` actually *process* the $\langle input \rangle$ file, depending on certain *parameters* some of which are set by the commands described next.

8.9.3 Preamble vs. main part of input file

A \LaTeX package typically has a “header” or “preamble” (automatically inserted by `docstrip`) with very scarce information on which file it is and what it provides, and with much more Legalese. Typesetting it in \TeX quality may be more misleading than typesetting it *verbatim*. So we typeset it *verbatim*. Now: where does the “header” end? `\NeedsTeXFormat` might be considered the border.— Yet it seems to be more simple and reliable just to act in terms of the *number of lines* that the header should be long. This length $\langle how-many-lines \rangle$ is declared by `\HeaderLines{ $\langle how-many-lines \rangle$ }`:

```

153 \newcommand*{\HeaderLines}{\def\header_lines}
154 \HeaderLines{0}

```

So the default is that there aren’t any header lines, unless another `\HeaderLines` is issued before some `\MakeDoc`. The way input is parsed *after* the “header” is set by `\MainDocParser{ $\langle parsing-command \rangle$ }`.

```
155 \newcommand*{\MainDocParser}{\def\main_doc_parser}
```

`\SectionLevelThreeParseInput` from section 8.6 is the default, two alternatives are defined there, another one is `\ProcessInputWith{comment}` from `fifinddo` and section 8.3 (general dividing into code and comments).

```
156 \MainDocParser{\SectionLevelThreeParseInput}
```

Here is how `\HeaderLines` and `\MainDocParser` act:

```
157 \def\make_doc_line_body{%
158   \IfInputLine{>\header_lines}%
159   {\let\make_doc_line_body\main_doc_parser
160    \make_doc_line_body}% %% switch to deciding
161   {\TreatAsCode{\fdInputLine}}} %% header verbatim
```

8.9.4 Screen messages

`\ProcessLineMessage{<command>}` is designed to choose a screen (or log) message `<command>`. `\ProcessLineMessage{\message{.}}` has a result like with `docstrip`. You just get one dot on screen per input line as a simple confirmation that the program is not hung up. However, the message may slow down a run considerably (if so, choose `\ProcessLineMessage{}` in the script). But it is better for beginner users of the package, so made default.

```
162 \newcommand*{\ProcessLineMessage}{\def\process_line_message}
163 % % \ProcessLineMessage{} %% no, still more efficient:
164 % \let\process_line_message\relax
165 \ProcessLineMessage{\message{.}}
```

8.9.5 Bundling-bundling Standalones

`\MakeInputJobDoc{<header-lines>}{<main-parser>}` by default produces a file

```
\jobname.doc from \jobname.sty
```

with some standard settings.⁵ `mdoccorr.cfg` (for `.txt`→`LATEX` functionality) is read, `\HeaderLines{<header-lines>}` and `\MainDocParser{<main-parser>}` and finally `\MakeCloseDoc{\jobname.sty}{\jobname.doc}` are executed. Here `\jobname` expands to the file name base of the `.tex` file you are running. It is assumed that you are preparing documentation for `\jobname.tex` for your `\jobname.sty`. In order to produce `<my-job>.doc` from `<my-job>.sty` instead,

```
\renewcommand{\mdJobName}{<my-job>}
```

If your input file has a different file name extension `<in-ext>` than ‘`sty`’, use an optional argument of `\MakeInputJobDoc`:

```
\MakeInputJobDoc[<in-ext>]{<header>}{<parser>}
```

⁵This command is new with v0.3.

If the output file should have a different extension $\langle out-ext \rangle$ than ‘doc’, you must use *two* optional arguments as follows:

```
\MakeInputJobDoc [ $\langle in-ext \rangle$ ] [ $\langle out-ext \rangle$ ] { $\langle header \rangle$ } { $\langle parser \rangle$ }
```

`\MakeInputJobDoc` does *not* execute `\ProcessLineMessage`, you can use the latter before so `\MakeInputJobDoc` respects it.

`\MakeJobDoc` does the same as `\MakeInputJobDoc` apart from typesetting the $\langle created \rangle$ file, so the latter needs an additional `\input{ $\langle created \rangle$ }`.

My original idea was that all preprocessing of package files to be documented should $\langle happen \rangle$ before `\documentclass`—loading `makedoc.sty` included—inside a group (`{ $\langle happen \rangle$ }`)—in order to avoid compatibility issues). However, it now appears to me that loading `makedoc` the usual way in the document *preamble* and processing the package file (that is to be documented) within the `document` environment works well enough and will be easier to comprehend.

This is the code for both `\MakeJobDoc` and `\MakeInputJobDoc`:

```
166 \ifdefinable{\mdJobName}{\let\mdJobName\jobname}
167 \newif\if_makedoc_input_
168 \newcommand*{\MakeInputJobDoc}{\_makedoc_input_true
169                               \make_job_doc_read}
170 \newcommand*{\MakeJobDoc}     {\_makedoc_input_false
171                               \make_job_doc_read}
172 \newcommand*{\make_job_doc_read}[1][sty]
173                               {\@testopt{\make_job_doc[#1]}\doc}}
```

Reading files as follows would fail with active `niceverb` settings, so we issue `\noNiceVerb` if it is defined. We do it inside a group in case `niceverb` settings are to be restored afterwards.

```
174 \def\make_job_doc[#1][#2]#3#4{%
175   \begingroup
176   \@ifundefined{noNiceVerb}{}%
177   {\let\MakeNormal\MakeNormalHere \noNiceVerb}%
178   \input{mdoccorr.cfg}%
179   %% <- TODO warning if not found!?
180   %%      or if one from TEXMF/ used inadvertently!?
181   %%      avoid reading twice!? 2010/03/11
182   %% <- TODO stack danger in group!? 2010/03/13
183   %% <- TODO or read it from 'makedoc' already! 2010/03/13
184   \LaTeXresultFile{\mdJobName.#2}%
185   \HeaderLines{#3}%
186   \MainDocParser{#4}%
187   \MakeCloseDoc{\mdJobName.#1}%
```

For typesetting the file just created, some `nicetext` features may be needed ... so restore the previous ones ...

```
188   \endgroup
189   \if_makedoc_input_\input{\mdJobName.#2}\fi
190 }
```

This feature may *change*.

8.10 Leave the package

```
191 \catcode'\_ =8 %% restores underscore use for subscripts
192 \endinput
```

8.11 VERSION HISTORY

```
193 v0.1 2009/04/03 very first version, tested on morgan.sty
194 v0.2 2009/04/05 \OnEmptyInputLine, \NoEmptyCodeLines
195 comment -> PPScomment, \IfFDinputEmpty,
196 \EveryComment, \PPstring may be par break
197 2009/04/08 \InputString -> \fdInputLine,
198 \section -> \subsection; documentation!
199 2009/04/08f. \MakeDoc
200 2009/04/12 ‘‘line too long’’ w/o redefining \PatternCodes;
201 \MakeDocCorrectHook
202 2009/04/13 tilde with sectioning
203 v0.3 2010/03/08 \WriteSection ‘trimspace’-like
204 2010/03/09 "fool" ("wiki" sectioning) nicer worded,
205 more use of ‘...’ in place of ‘\dots’;
206 different treatment of package code environment
207 (new separate subsection); clarification on
208 \ProcessInputWith{comment}
209 2010/03/10 supplied ‘\ref’
210 2010/03/11 \MakeCloseDoc; corrected "undifined";
211 \par\noindent in ‘Sectioning’; \MakeJobDoc
212 2010/03/12 &.&.&; updated copyright
213 2010/03/13 corr.: ‘_’ not ‘other’; tried to explain my
214 earlier reasoning about ‘\ifPackageCode’;
215 \MakeInputJobDoc
216 2010/03/14 \make_doc_job without \niceverb_aux_cat
217 2010/03/15 another remark to \ifPackageCode
218 2010/03/16 use box with comment line markers;
219 mdcorr -> mdoccorr
220 2010/03/18 report on using \EveryComment
221 2010/03/19 ’’ -> "
222 v0.4 2010/03/23 "Distinguishing"
223 2010/03/24 "both in"
224 2010/03/27 switch back to \fdPatternCodes
225 2010/03/28 include ‘% ’ commenting style
226 2010/03/29 \ResetCodeLineNumbers
227 2010/03/30 use \SetPatternCodes, \ResetPatternCodes
228 v0.41 2010/12/20 \ResetCodeLineNumbers defined globally
229 2010/12/21 ... rather presented as a bug-fix
230 2011/01/19 \mdStartPackageCode
231 2011/01/25 updated (C)
232
```

The previous empty code line is the one TeX insists to add at every end of a file it writes.