The DocStrip program *

Frank Mittelbach Denys Duchier Johannes Braams
Marcin Woliński Mark Wooding

Printed October 27, 2010

Abstract

This document describes the implementation of the DocStrip program. The original version of this program was developed by Frank Mittelbach to accompany his `doc.sty` which enables literate programming in LaTeX. Denys Duchier rewrote it to run either with TeX or with LaTeX, and to allow full boolean expressions in conditional guards instead of just comma-separated lists. Johannes Braams re-united the two implementations, documented and debugged the code.

In September 1995 Marcin Woliński changed many parts of the program to make use of TeX's ability to write to multiple files at the same time to avoid re-reading sources. The performance improvement of version 2.3 came at a price of compatibility with some more obscure operating systems which limit the number of files a process can keep open. This was corrected in September 1996 by Mark Wooding and his changes were "creatively merged" by Marcin Woliński who made at the same time changes in batch files processing, handling of preambles and introduced "verbatim mode". After all that, David Carlisle merged the new version into the LaTeX sources, and made a few other changes, principally making DocStrip work under initex, and removing the need for batch files to say `\def\batchfile{...}`.

1 Introduction

1.1 Why the DocStrip program?

When Frank Mittelbach created the `doc` package, he invented a way to combine TeX code and its documentation. From then on it was more or less possible to do literate programming in TeX.

This way of writing TeX programs obviously has great advantages, especially when the program becomes larger than a couple of macros. There is one drawback however, and that is that such programs may take longer than expected to run because TeX is an interpreter and has to decide for each line of the program file what it has to do with it. Therefore, TeX programs may be speeded up by removing all comments from them.

By removing the comments from a TeX program a new problem is introduced. We now have two versions of the program and both of them *have* to be maintained. Therefore it would be nice to have a possibility to remove the comments automatically, instead of doing it by hand. So we need a program to remove comments from TeX programs. This could be programmed in any high level language, but maybe not everybody has the right compiler to compile the program. Everybody who wants to remove comments from TeX programs has TeX. Therefore the DocStrip program is implemented entirely in TeX.

*This file has version number 2.5d, last revised 2005/07/29, documentation dated 1999/03/31.

1.2 Functions of the DocStrip program

Having created the DocStrip program to remove comment lines from TEX programs[1] it became feasible to do more than just strip comments.
Wouldn't it be nice to have a way to include parts of the code only when some condition is set true? Wouldn't it be as nice to have the possibility to split the source of a TEX program into several smaller files and combine them later into one 'executable'?
Both these wishes have been implemented in the DocStrip program.

2 How to use the DocStrip program

A number of ways exist to use the DocStrip program:

1. The usual way to use DocStrip is to write a *batch file* in such a way that it can be directly processed by TEX. The batch file should contain the commands described below for controlling the DocStrip program. This allows you to set up a distribution where you can instruct the user to simply run

   `TEX` ⟨*batch file*⟩

   to generate the executable versions of your files from the distribution sources. Most of the LATEX distribution is packaged this way. To produce such a batch file include a statement in your 'batch file' that instructs TEX to read `docstrip.tex`. The beginning of such a file would look like:

   ```
   \input docstrip
   ...
   ```

   By convention the batch file should have extension `.ins`. But these days DocStrip in fact work with any extension.

2. Alternatively you can instruct TEX to read the file `docstrip.tex` and to see what happens. TEX will ask you a few questions about the file you would like to be processed. When you have answered these questions it does its job and strips the comments from your TEX code.

3 Configuring DocStrip

3.1 Selecting output directories

Inspired by a desire to simplify reinstallations of LATEX 2ε and to support operating systems which have an upper limit on the number of files allowed in a directory, DocStrip now allows installation scripts to specify output directories for files it creates. We suggest using TDS (TEX directory structure) names of directories relative to `texmf` here. However these names should be thought of as a labels rather than actual names of directories. They get translated to actual system-dependent pathnames according to commands contained in a configuration file named `docstrip.cfg`.

The configuration file is read by DocStrip just before it starts to process any batch file commands.

If this file is not present DocStrip uses some default settings which ensure that files are only written to the current directory. However by use of this configuration file, a site maintainer can 'enable' features of DocStrip that allow files to be written to alternative directories.

`\usedir` Using this macro package author can tell where a file should be installed.

[1] Note that only comment lines, that is lines that start with a single `%` character, are removed; all other comments stay in the code.

All \files generated in the scope of that declaration are written to a directory specified by its one argument. For example in LaTeX 2ε installation following declarations are used:

```
\usedir{tex/latex/base}
\usedir{makeindex}
```

And standard packages use

```
\usedir{tex/latex/tools}
\usedir{tex/latex/babel}
```

etc.

\showdirectory Used to display directory names in messages. If some label is not defined it expands to `UNDEFINED (label is ...)` otherwise to a directory name. It is probably a good idea for every installation script to display at startup list of all directories that would be used and asking user to confirm that.

The above macros are used by package/installation script author. The following macros are used in a configuration file, `docstrip.cfg`, by a system administrator to describe her/his local directory structure.

\BaseDirectory This macro is administrator's way of saying "yes, I want to use that directories support of yours". DocStrip will write only to current directory unless your config has a call to this macro. (This means DocStrip won't write to random directories unless you tell it to, which is nice.) Using this macro you can specify a base directory for TeX-related stuff. E.g., for many Unix systems that would be

```
\BaseDirectory{/usr/local/lib/texmf}
```

and for standard emTeX installation

```
\BaseDirectory{c:/emtex}
```

\DeclareDir Having specified the base directory you should tell DocStrip how to interpret labels used in \usedir commands. This is done with \DeclareDir with two arguments. The first is the label and the second is actual name of directory relative to base directory. For example to teach DocStrip using standard emTeX directories one would say:

```
\BaseDirectory{c:/emtex}
\DeclareDir{tex/latex/base}{texinput/latex2e}
\DeclareDir{tex/latex/tools}{texinput/tools}
\DeclareDir{makeindex}{idxstyle}
```

This will cause base latex files and font descriptions to be written to directory `c:\emtex\texinput\latex2e`, files of `tools` package to `c:\emtex\texinput\tools` and makeindex files to `c:\emtex\idxstyle`.

Sometimes it is desirable to put some files outside of the base directory. For that reason \DeclareDir has a star form specifying absolute pathname. For example one could say

```
\DeclareDir*{makeindex}{d:/tools/texindex/styles}
```

\UseTDS Users of systems conforming to TDS may well ask here "do I really need to put a dozen of lines like

```
\DeclareDir{tex/latex/base}{tex/latex/base}
```

in my config file". The answer is \UseTDS. This macro causes DocStrip to use labels themselves for any directory you haven't overriden with \DeclareDir. The default behaviour is to raise an error on undefined labels because some users may want to know exactly where files go and not to allow DocStrip to write to random places. However I (MW) think this is pretty cool and my config says just (I'm running teTeX under Linux)

```
\BaseDirectory{/usr/local/teTeX/texmf}
\UseTDS
```

The important thing to note here is that it is impossible to create a new directory from inside TEX. So however you configure DocStrip, you need to create all needed directories before running the installation. Authors may want to begin every installation script by displaying a list of directories that will be used and asking user if he's sure all of them exist.

Since file name syntax is OS specific DocStrip tries to guess it from the current directory syntax. It should succeed for Unix, MSDOS, Macintosh and VMS. However DocStrip will only initially know the current directory syntax if it is used with LATEX. If used with plainTEX or initex it will not have this information[2]. If you often use DocStrip with formats other than LATEX you should *start* the file `docstrip.cfg` with a definition of `\WriteToDir`. E.g., `\def\WriteToDir{./}` on MSDOS/Unix, `\def\WriteToDir{:}` on Macintosh, `\def\WriteToDir{[]}` on VMS.

If your system requires something completely different you can define in `docstrip.cfg` macros `\dirsep` and `\makepathname`. Check for their definition in the implementation part. If you want some substantially different scheme of translating `\usedir` labels into directory names try redefining macro `\usedir`.

3.2 Setting maximum numbers of streams

`\maxfiles` In support of some of the more obscure operating systems, there's a limit on the number of files a program can have open. This can be expressed to DocStrip through the `\maxfiles` macro. If the number of streams DocStrip is allowed to open is n, your configuration file can say `\maxfiles{`n`}`, and DocStrip won't try to open more files than this. Note that this limit won't include files which are already open. There'll usually be two of these: the installation script which you started, and the file `docstrip.tex` which it included; you must bear these in mind yourself. DocStrip assumes that it can open at least four files before it hits some kind of maximum: if this isn't the case, you have real problems.

`\maxoutfiles` Maybe instead of having a limit on the number of files TEX can have open, there's a limit on the number of files it can write to (e.g., TEX itself imposes a limit of 16 files being written at a time). This can be expressed by saying `\maxoutfiles{`m`}` in a configuration file. You must be able to have at least one output file open at a time; otherwise DocStrip can't do anything at all.

Both these options would typically be put in the `docstrip.cfg` file.

4 The user interface

4.1 The main program

`\processbatchFile` The 'main program' starts with trying to process a batch file, this is accomplished by calling the macro `\processbatchFile`. It counts the number of batch files it processes, so that when the number of files processed is still zero after the call to `\processbatchFile` appropriate action can be taken.

`\interactive` When no batch files have been processed the macro `\interactive` is called. It prompts the user for information. First the extensions of the input and output files is determined. Then a question about optional code is asked and finally the user can give a list of files that have to be processed.

`\ReportTotals` When the `stats` option is included in the DocStrip-program it keeps a record of the number of files and lines that are processed. Also the number of comments removed and passed as well as the number of code lines that were passed to the output are accounted. The macro `\ReportTotals` shows a summary of this information.

[2] Except when processing the main `unpack.ins` batch file for the LATEX distribution, which takes special measures so that initex can learn the directory syntax.

4.2 Batchfile commands

The commands described in this section are available to build a batch file for TEX.

\input All DocStrip batch files should start with the line: `\input docstrip`

Do not use the LATEX syntax `\input{docstrip}` as batch files may be used with plain TEX or iniTEX. You may that old batch files always have a line `\def\batchfile{`⟨*filename*⟩`}` just before the input. Such usage is still supported but is now discouraged, as it causes TEX to re-input the same file, using up one of its limited number of input streams.

\endbatchfile All batch files should end with this command. Any lines after this in the file are ignored. In old files that start `\def\batchfile{...` this command is optional, but is a good idea anyway. If this command is omitted from a batchfile then normally TEX will go to its interactive `*` prompt, so you may stop DocStrip by typing `\endbatchfile` to this prompt.

\generate \file \from The main reason for constructing a DocStrip command file is to describe what files should be generated, from what sources and what optional ('guarded') pieces of code should be included. The macro `\generate` is used to give TEX this information. Its syntax is:

`\generate{[\file{`⟨*output*⟩`}{[\from{`⟨*input*⟩`}{`⟨*optionlist*⟩`}]*}]*}`

The ⟨*output*⟩ and ⟨*input*⟩ are normal file specifications as are appropriate for your computer system. The ⟨*optionlist*⟩ is a comma separated list of 'options' that specify which optional code fragments in ⟨*input*⟩ should be included in ⟨*output*⟩. Argument to `\generate` may contain some local declarations (e.g., the `\use...` commands described below) that will apply to all `\file`s after them. Argument to `\generate` is executed inside a group, so all local declarations are undone when `\generate` concludes.

It is possible to specify multiple input files, each with its own ⟨*optionlist*⟩. This is indicated by the notation [...]*. Moreover there can be many `\file` specifications in one `\generate` clause. This means that all these ⟨*output*⟩ files should be generated while reading each of ⟨*input*⟩ files once. Input files are read in order of first appearance in this clause. E.g.

```
\generate{\file{p1.sty}{\from{s1.dtx}{foo,bar}}
          \file{p2.sty}{\from{s2.dtx}{baz}
                        \from{s3.dtx}{baz}}
          \file{p3.sty}{\from{s1.dtx}{zip}
                        \from{s2.dtx}{zip}}
         }
```

will cause DocStrip to read files `s1.dtx`, `s2.dtx`, `s3.dtx` (in that order) and produce files `p1.sty`, `p2.sty`, `p3.sty`.

The restriction to at most 16 output streams open in a while does not mean that you can produce at most 16 files with one `\generate`. In the example above only 2 streams are needed, since while `s1.dtx` is processed only `p1.sty` and `p3.sty` are being generated; while reading `s2.dtx` only `p2.sty` and `p3.sty`; and while reading `s3.dtx` file `p2.sty` . However example below needs 3 streams:

```
\generate{\file{p1.sty}{\from{s1.dtx}{foo,bar}}
          \file{p2.sty}{\from{s2.dtx}{baz}
                        \from{s3.dtx}{baz}}
          \file{p3.sty}{\from{s1.dtx}{zip}
                        \from{s3.dtx}{zip}}
         }
```

Although while reading `s2.dtx` file `p3.sty` is not written it must remain open since some parts of `s3.dtx` will go to it later.

Sometimes it is not possible to create a file by reading all sources once. Consider the following example:

```
\generate{\file{p1.sty}{\from{s1.dtx}{head}
```

```
                \from{s2.dtx}{foo}
                \from{s1.dtx}{tail}}
 \file{s1.drv}{\from{s1.dtx}{driver}}
}
```

To generate `p1.sty` file `s1.dtx` must be read twice: first time with option `head`, then file `s2.dtx` is read and then `s1.dtx` again this time with option `tail`. DocStrip handles this case correctly: if inside one `\file` declaration there are multiple `\from`es with the same input file this file *is* read multiple times.

If the order of `\from`s specified in one of your `\file` specifications does not match the order of input files established by previous `\file`s, DocStrip will raise an error and abort. Then you may either read one of next sections or give up and put that file in separate `\generate` (but then sources will be read again just for that file).

For impatient. Try following algorithm: Find file that is generated from largest number of sources, start writing `\generate` clause with this file and its sources in proper order. Take other files that are to be generated and add them checking if they don't contradict order of sources for the first one. If this doesn't work read next sections.

For mathematicians. Relation "file A must be read before file B" is a partial order on the set of all your source files. Each `\from` clause adds a chain to this order. What you have to do is to perform a topological sort i.e. to extend partial order to linear one. When you have done it just list your source files in `\generate` in such a way that order of their first appearance in the clause matches linear order. If this cannot be achieved read next paragraph. (Maybe future versions of DocStrip will perform this sort automatically, so all these troubles will disappear.)

For that who must know that all. There is a diverse case when it's not possible to achieve proper order of reading source files. Suppose you have to generate two files, first from `s1.dtx` and `s3.dtx` (in that order) and second from `s2.dtx` and `s3.dtx`. Whatever way you specify this the files will be read in either as `s1 s3 s2` or `s2 s3 s1`. The key to solution is magical macro `\needed` that marks a file as needed to be input but not directing any output from it to current `\file`. In our example proper specification is:

```
\generate{\file{p1.sty}{\from{s1.dtx}{foo}
                        \needed{s2.dtx}
                        \from{s3.dtx}{bar}}
          \file{p2.sty}{\from{s2.dtx}{zip}
                        \from{s3.dtx}{zap}}
}
```

`\askforoverwritetrue` `\askforoverwritefalse`

These macros specify what should happen if a file that is to be generated already exists. If `\askforoverwritetrue` is active (the default) the user is asked whether the file should be overwritten. If however `\askforoverwritefalse` was issued existing files will be overwritten silently. These switches are local and can be issued in any place in the file even inside `\generate` clause (between `\file`s however).

`\askonceonly`

You might not want to set `\askforoverwritefalse` in a batch file as that says that it us always all right to overwrite other people's files. However for large installations, such as the base LaTeX distribution, being asked individually about hundreds of files is not very helpful either. A batchfile may therefore specify `\askonceonly`. This means that after the first time the batchfile asks the user a question, the user is given an option of to change the behaviour so that 'yes' will be automatically assumed for all future questions. This applies to any use of the DocStrip commamnd `\Ask` including, but not restricted to, the file overwrite questions controlled by `\askforoverwritetrue`.

\preamble
\endpreamble
\postamble
\endpostamble

It is possible to add a number of lines to the output of the DocStrip program. The information you want to add to the start of the output file should be listed between the `\preamble` and `\endpreamble` commands; the lines you want to add to the end of the output file should be listed between the `\postamble` and `\endpostamble` commands. Everything that DocStrip finds for both the pre- and postamble it writes to the output file, but preceded with value of `\MetaPrefix` (default is two %-characters). If you include a `^^J` character in one of these lines, everything that follows it on the same line is written to a new line in the output file. This 'feature' can be used to add a `\typeout` or `\message` to the the stripped file.

\declarepreamble
\declarepostamble
\usepreamble
\usepostamble
\nopreamble
\nopostamble

Sometimes it is desirable to have different preambles for different files of a larger package (e.g., because some of them are customisable configuration files and they should be marked as such). In such a case one can say `\declarepreamble\somename`, then type in his/her preamble, end it with `\endpreamble`, and later on `\usepreamble\somename` to switch to this preamble. If no preamble should be used you can deploy the `\nopreamble` command. This command is equivalent to saying `\usepreamble\empty`. The same mechanism works for postambles, `\use...` declarations are local and can appear inside `\generate`.

Commands `\preamble` and `\postamble` define and activate pre(post)ambles named `\defaultpreamble` and `\defaultpostamble`.

\batchinput

The batch file commands can be put into several batch files which are then executed from a master batch file. This is, for example, useful if a distribution consists of several distinct parts. You can then write individual batch files for every part and in addition a master file that simply calls the batch files for the parts. For this, call the individual batch files from the master file with the command `\batchinput{`⟨*file*⟩`}`. Don't use `\input` for this purpose, this command should be used only for calling the DocStrip program as explained above and is ignored when used for any other purpose.

\ifToplevel

When batch files are nested you may want to suppress certain commands in the lower-level batch files such as terminal messages. For this purpose you can use the `\ifToplevel` command which executes its argument only if the current batch file is the outermost one. Make sure that you put the opening brace of the argument into the same line as the command itself, otherwise the DocStrip program will get confused.

\showprogress
\keepsilent

When the option `stats` is included in DocStrip it can write message to the terminal as each line of the input file(s) is processed. This message consists of a single character, indicating kind of that particular line. We use the following characters:

- `%` Whenever an input line is a comment %-character is written to the terminal.
- `.` Whenever a code line is encountered a .-character is written on the terminal.
- `/` When a number of empty lines appear in a row in the input file, at most one of them is retained. The DocStrip program signals the removal of an empty line with the /-character.
- `<` When a 'guard line' is found in the input and it starts a block of optionally included code, this is signalled on the terminal by showing the <-character, together with the boolean expression of the guard.
- `>` The end of a conditionally included block of code is indicated by showing the >-character.

This feature is turned on by default when the option `stats` is included, otherwise it is turned off. The feature can be toggled with the commands `\showprogress` and `\keepsilent`.

4.2.1 Supporting old interface

\generateFile Here is the old syntax for specifing what files are to be generated. It allows specification of just one output file.

\generateFile{⟨*output*⟩}{⟨*ask*⟩}{[\from{⟨*input*⟩}{⟨*optionlist*⟩}]*}

The meaning of ⟨*output*⟩, ⟨*input*⟩ and ⟨*optionslist*⟩ is just as for \generate. With ⟨*ask*⟩ you can instruct TEX to either silently overwrite a previously existing file (f) or to issue a warning and ask you if it should overwrite the existing file (t) (it overrides the \askforoverwrite setting).

\include \processFile The earlier version of the DocStrip program supported a different kind of command to tell TEX what to do. This command is less powerful than \generateFile; it can be used when ⟨*output*⟩ is created from one ⟨*input*⟩. The syntax is:

\include{⟨*optionlist*⟩}

\processFile{⟨*name*⟩}{⟨*inext*⟩}{⟨*outext*⟩}{⟨*ask*⟩}

This command is based on environments where filenames are constructed of two parts, the name and the extension, separated with a dot. The syntax of this command assumes that the ⟨*input*⟩ and ⟨*output*⟩ share the same name and only differ in their extension. This command is retained to be backwards compatible with the older version of DocStrip, but its use is not encouraged.

5 Conditional inclusion of code

When you use the DocStrip program to strip comments out of TEX macro files you have the possibility to make more than one stripped macro file from one documented file. This is achieved by the support for optional code. The optional code is marked in the documented file with a 'guard'.

A guard is a boolean expression that is enclosed in < and >. It also *has* to follow the % at the beginning of the line. For example:

```
...
%<bool>\TeX code
...
```

In this example the line of code will be included in ⟨*output*⟩ if the option bool is present in the ⟨*optionlist*⟩ of the \generateFile command.

The syntax for the boolean expressions is:

⟨*Expression*⟩ ::= ⟨*Secondary*⟩ [{|, ,} ⟨*Secondary*⟩]*
⟨*Secondary*⟩ ::= ⟨*Primary*⟩ [& ⟨*Primary*⟩]*
⟨*Primary*⟩ ::= ⟨*Terminal*⟩ | !⟨*Primary*⟩ | (⟨*Expression*⟩)

The | stands for disjunction, the & stands for conjunction and the ! stands for negation. The ⟨*Terminal*⟩ is any sequence of letters and evaluates to ⟨*true*⟩ iff[3] it occurs in the list of options that have to be included.

Two kinds of optional code are supported: one can either have optional code that 'fits' on one line of text, like the example above, or one can have blocks of optional code.

To distinguish both kinds of optional code the 'guard modifier' has been introduced. The 'guard modifier' is one character that immediately follows the < of the guard. It can be either * for the beginning of a block of code, or / for the end of a block of code[4]. The beginning and ending guards for a block of code have to be on a line by themselves.

When a block of code is *not* included, any guards that occur within that block are *not* evaluated.

[3] iff stands for 'if and only if'

[4] To be compatible with the earlier version of DocStrip also + and - are supported as 'guard modifiers'. However, there is an incompatibility with the earlier version since a line with a +-modified guard is not included inside a block with a guard that evaluates to false, in contrast to the previous behaviour.

6 Those other languages

Since TEX is an open system some of TEX packages include non-TEX files. Some authors use DocStrip to generate PostScript headers, shell scripts or programs in other languages. For them the comments-stripping activity of DocStrip may cause some trouble. This section describes how to produce non-TEX files with DocStrip effectively.

6.1 Stuff DocStrip puts in every file

First problem when producing files in "other" languages is that DocStrip adds some bits to the begining and end of every generated file that may not fit with the syntax of the language in question. So we'll study carefully what exactly goes where.

The whole text put on begining of file is kept in a macro defined by `\declarepreamble`. Every line of input presented to `\declarepreamble` is prepended with current value of `\MetaPrefix`. Standard DocStrip header is inserted before your text, and macros `\inFileName`, `\outFileName` and `\ReferenceLines` are used as placeholders for information which will be filled in later (specifically for each output file). Don't try to redefine these macros. After

```
\declarepreamble\foo
----------------------------
Package FOO for use with TeX
\endpreamble
```

macro `\foo` is defined as

```
%%^^J
%% This is file `\outFileName ',^^J
%% generated with the docstrip utility.^^J
\ReferenceLines^^J
%% ----------------------------^^J
%% Package FOO for use with TeX.
```

You can play with it freely or even define it from scratch. To embed the preamble in Adobe structured comments just use `\edef`:

```
\edef\foo{\perCent!PS-Adobe-3.0^^J%
          \DoubleperCent\space Title: \outFileName^^J%
          \foo^^J%
          \DoubleperCent\space EndComments}
```

After that use `\usepreamble\foo` to select your new preamble. Everything above works as well for postambles.

You may also prevent DocStrip from adding anything to your file, and put any language specific invocations directly in your code:

```
\generate{\usepreamble\empty
          \usepostamble\empty
          \file{foo.ps}{\from{mypackage.dtx}{ps}}}
```

or alternatively `\nopreamble` and `\nopostamble`.

6.2 Meta comments

You can change the prefix used for putting meta comments to output files by redefining `\MetaPrefix`. Its default value is `\DoubleperCent`. The preamble uses value of `\MetaPrefix` current at time of `\declarepreamble` while meta comments in the source file use value current at time of `\generate`. Note that this means that you cannot produce concurrently two files using different `\MetaPrefix`es.

6.3 Verbatim mode

If your programming language uses some construct that can interferes badly with DocStrip (e.g., percent in column one) you may need a way for preventing it from being stripped off. For that purpose DocStrip features 'verbatim mode'.

A 'Guard expression' of the form `%<<`⟨*END-TAG*⟩ marks the start of a section that will be copied verbatim upto a line containing only a percent in column 1 followed by ⟨*END-TAG*⟩. You can select any ⟨*END-TAG*⟩ you want, but note that spaces count here. Example:

```
%<*myblock>
some stupid()
   #computer<program>
%<<COMMENT
% These two lines are copied verbatim (including percents
%% even if \MetaPrefix is something different than %%).
%COMMENT
   using*strange@programming<language>
%</myblock>
```

And the output is (when stripped with `myblock` defined):

```
some stupid()
   #computer<program>
% These two lines are copied verbatim (including percents
%% even if \MetaPrefix is something different than %%).
   using*strange@programming<language>
```

7 Producing the documentation

We provide a short driver file that can be extracted by the DocStrip program using the the conditional 'driver'. To allow the use of `docstrip.dtx` as a program at IniTEX time (e.g., to strip off its own comments) we need to add a bit of primitive code. With this extra checking it is still possible to process this file with LaTeX 2ε to typeset the documentation.

```
1 ⟨*driver⟩
```

If `\documentclass` is undefined, e.g., if IniTEX or plain TEX is used for formatting, we bypass the driver file.

We use some trickery to avoid issuing `\end{document}` when the `\ifx` construction is unfinished. If condition below is true a `\fi` is constructed on the fly, the `\ifx` is completed, and the real `\fi` will never be seen as it comes after `\end{document}`. On the other hand if condition is false TEX skips over `\csname fi\endcsname` having no idea that this could stand for `\fi`, driver is skipped and only then the condition completed.

Additional guard `gobble` prevents DocStrip from extracting these tricks to real driver file.

```
2 ⟨*gobble⟩
3 \ifx\jobname\relax\let\documentclass\undefined\fi
4 \ifx\documentclass\undefined
5 \else \csname fi\endcsname
6 ⟨/gobble⟩
```

Otherwise we process the following lines which will result in formatting the documentation.

```
7  \documentclass{ltxdoc}
8     \EnableCrossrefs
9  % \DisableCrossrefs
10 % use \DisableCrossrefs if the
11 % index is ready
12 \RecordChanges
13 % \OnlyDescription
```

```
\typeout{Expect some Under- and overfull boxes}
\begin{document}
  \DocInput{docstrip.dtx}
\end{document}
⟨*gobble⟩
\fi
⟨/gobble⟩
⟨/driver⟩
```

8 The implementation

8.1 Initex initializations

Allow this program to run with `initex`. The `Z` trickery saves the need to worry about `\outer` stuff in plain TEX.

```
⟨*initex⟩
\catcode`\Z=\catcode`\%
\ifnum13=\catcode`\~{\egroup\else
  \catcode`\Z=9
Z
Z  \catcode`\{=1  \catcode`\}=2
Z  \catcode`\#=6  \catcode`\^=7
Z  \catcode`\@=11 \catcode`\^^L=13
Z  \let\bgroup={  \let\egroup=}
Z
Z  \dimendef\z@=10 \z@=0pt \chardef\@ne=1 \countdef\m@ne=22 \m@ne=-1
Z  \countdef\count@=255
Z
Z  \def\wlog{\immediate\write\m@ne} \def\space{ }
Z
Z  \count10=22 % allocates \count registers 23, 24, ...
Z  \count15=9 % allocates \toks registers 10, 11, ...
Z  \count16=-1 % allocates input streams 0, 1, ...
Z  \count17=-1 % allocates output streams 0, 1, ...
Z
Z  \def\alloc@#1#2#3{\advance\count1#1\@ne#2#3\count1#1\relax}
Z
Z  \def\newcount{\alloc@0\countdef} \def\newtoks{\alloc@5\toksdef}
Z  \def\newread{\alloc@6\chardef}   \def\newwrite{\alloc@7\chardef}
Z
Z \def\newif#1{%
Z   \count@\escapechar \escapechar\m@ne
Z     \let#1\iffalse
Z     \@if#1\iftrue
Z     \@if#1\iffalse
Z   \escapechar\count@}
Z \def\@if#1#2{%
Z   \expandafter\def\csname\expandafter\@gobbletwo\string#1%
Z                       \expandafter\@gobbletwo\string#2\endcsname
Z                          {\let#1#2}}
Z
Z  \def\@gobbletwo#1#2{}
Z  \def\@gobblethree#1#2#3{}
Z
Z  \def\loop#1\repeat{\def\body{#1}\iterate}
Z  \def\iterate{\body \let\next\iterate \else\let\next\relax\fi \next}
Z  \let\repeat\fi
Z
Z  \def\empty{}
Z
Z  \def\tracingall{\tracingcommands2 \tracingstats2
```

```
Z    \tracingpages1 \tracingoutput1 \tracinglostchars1
Z    \tracingmacros2 \tracingparagraphs1 \tracingrestores1
Z    \showboxbreadth 10000 \showboxdepth 10000 \errorstopmode
Z    \errorcontextlines 10000 \tracingonline1 }
Z
\bgroup}\fi\catcode`\Z=11
\let\bgroup={  \let\egroup=}
⟨/initex⟩
```

8.2 Declarations and initializations

In order to be able to include the `@`-sign in control sequences its category code is changed to ⟨*letter*⟩. The 'program' guard here allows most of the code to be excluded when extracting the driver file.

```
⟨*program⟩
\catcode`\@=11
```

When we want to write multiple lines to the terminal with one statement, we need a character that tells TEX to break the lines. We use `^^J` for this purpose.

```
\newlinechar=`\^^J
```

8.2.1 Switches

`\ifGenerate` The program will check if a file of the same name as the file it would be creating already exists. The switch `\ifGenerate` is used to indicate if the stripped file has to be generated.

```
\newif\ifGenerate
```

`\ifContinue` The switch `\ifContinue` is used in various places in the program to indicate if a `\loop` has to end.

```
\newif\ifContinue
```

`\ifForlist` The program contains an implementation of a for-loop, based on plain TEX's `\loop` macros. The implementation needs a switch to terminate the loop.

```
\newif\ifForlist
```

`\ifDefault` The switch `\ifDefault` is used to indicate whether the default batch file has to be used.

```
\newif\ifDefault
```

`\ifMoreFiles` The switch `\ifMoreFiles` is used to decide if the user wants more files to be processed. It is used only in interactive mode; initially it evaluates to ⟨*true*⟩.

```
\newif\ifMoreFiles \MoreFilestrue
```

`\ifaskforoverwrite` The switch `\askforoverwrite` is used to decide if the user should be asked when a file is to be overwritten.

```
\newif\ifaskforoverwrite \askforoverwritetrue
```

8.2.2 Count registers

`\blockLevel` Optionally included blocks of code can be nested. The counter `\blockLevel` will be used to keep track of the level of nesting. Its initial value is zero.

```
\newcount\blockLevel \blockLevel\z@
```

`\emptyLines` The count register `\emptyLines` is used to count the number of consecutive empty input lines. Only the first will be copied to the output file.

```
\newcount\emptyLines \emptyLines \z@
```

\processedLines
\commentsRemoved
\commentsPassed
\codeLinesPassed

To be able to provide the user with some statistics about the stripping process four counters are allocated if the statistics have been included when this program was DocStripped. The number of lines processed is stored in the counter \processedLines. The number of lines containing comments that are not written on the output file is stored in the counter \commentsRemoved; the number of comments copied to the output file is stored in the counter \commentsPassed. The number of lines containing macro code that are copied to the output file is stored in the counter \codeLinesPassed.

```
⟨*stats⟩
\newcount\processedLines   \processedLines  \z@
\newcount\commentsRemoved  \commentsRemoved \z@
\newcount\commentsPassed   \commentsPassed  \z@
\newcount\codeLinesPassed  \codeLinesPassed \z@
```

\TotalprocessedLines
\TotalcommentsRemoved
\TotalcommentsPassed
\TotalcodeLinesPassed

When more than one file is processed and when statistics have been included we provide the user also with information about the total amount of lines processed. For this purpose four more count registers are allocated here.

```
\newcount\TotalprocessedLines   \TotalprocessedLines  \z@
\newcount\TotalcommentsRemoved  \TotalcommentsRemoved \z@
\newcount\TotalcommentsPassed   \TotalcommentsPassed  \z@
\newcount\TotalcodeLinesPassed  \TotalcodeLinesPassed \z@
⟨/stats⟩
```

\NumberOfFiles

When more than one file is processed, the number of files is stored in the count register \NumberOfFiles.

```
\newcount\NumberOfFiles \NumberOfFiles\z@
```

8.2.3 I/O streams

\inFile

For reading the file with documented TEX-code, an input stream \inFile is allocated.

```
\newread\inFile
```

\ttyin
\ttyout

Communication with the user goes through (nonexistent) stream 16.

```
\chardef\ttyin16
\chardef\ttyout16
```

\inputcheck

This stream is only used for checking for existence of files.

```
\newread\inputcheck
```

\ifToplevel

Execute the argument if current batch file is the outermost one. Otherwise suppress it.

```
\newif\iftopbatchfile \topbatchfiletrue
\def\ifToplevel{\relax\iftopbatchfile
   \expandafter\iden \else \expandafter\@gobble\fi}
```

\batchinput

When the file docstrip.tex is read because of an \input statement in a batch file we have to prevent an endless loop (well, limited by TEX's stack). Therefore we save the original primitive \input and define a new macro with an argument delimited by ␣ (i.e. a space) that just gobbles the argument. Since the end-of-line character is converted by TEX to a space. This means that \input is not available as a command within batch files.

\@@input

We therefore keep a copy of the original under the name \@@input for internal use. If DocStrip runs under LATEX this command is already defined, so we make a quick test.

```
\ifx\undefined\@@input \let\@@input\input\fi
```

To allow the nesting of batch files the `\batchinput` command is provided it takes one argument, the name of the batch file to switch to.

```
\def\batchinput#1{%
```

We start a new group and locally redefine `\batchFile` to hold the new batch file name. We toggle the `\iftopbatchfile` switch since this definitely is not top batch file.

```
  \begingroup
    \def\batchfile{#1}%
    \topbatchfilefalse
    \Defaultfalse
    \usepreamble\org@preamble
    \usepostamble\org@postamble
    \let\destdir\WriteToDir
```

After this we can simply call `\processbatchFile` which will open the new batch file and read it until it is exhausted. Note that if the batch file is not available, or misspelled this routine will produce a warning and return.

```
    \processbatchFile
```

The value of `\batchfile` as well as local definitions of preambles, directories etc. will be restored at this closing `\endgroup`, so that further processing continues in the calling batch file.

```
  \endgroup
}
```

`\skip@input` And here is the promised redefinition of `\input`:

```
\def\skip@input#1 {}
\let\input\skip@input
```

8.2.4 Empty macros and macros that expand to a string

`\guardStack` Because blocks of code that will conditionally be included in the output can be nested, a stack is maintained to keep track of these blocks. The main reason for this is that we want to be able to check if the blocks are properly nested. The stack itself is stored in `\guardStack`.

```
\def\guardStack{}
```

`\blockHead` The macro `\blockHead` is used for storing and retrieving the boolean expression that starts a block.

```
\def\blockHead{}
```

`\yes` `\y` When the user is asked a question that he has to answer with either ⟨*yes*⟩ or ⟨*no*⟩, his response has to be evaluated. For this reason the macros `\yes` and `\y` are defined.

```
\def\yes{yes}
\def\y{y}
```

`\n` We also define `\n` for use in DocStrip command files.

```
\def\n{n}
```

`\Defaultbatchile` When the DocStrip program has to process a batch file it can look for a batch file with a default name. This name is stored in `\DefaultbatchFile`.

```
\def\DefaultbatchFile{docstrip.cmd}
```

`\perCent` `\DoubleperCent` `\MetaPrefix` To be able to display percent-signs on the terminal, a `%` with category code 12 is stored in `\perCent` and `\DoubleperCent`. The macro `\MetaPrefix` is put on begining of every meta-comment line. It is defined indirect way since some applications need redefining it.

```
{\catcode`\%=12
 \gdef\perCent{%}
```

```
\gdef\DoubleperCent{%%}
}
\let\MetaPrefix\DoubleperCent
```

In order to allow formfeeds in the input we define a one-character control sequence `^^L`.

```
\def^^L{ }
```

The only result of using `\Name` is slowing down execution since its typical use (e.g., `\Name\def{foo bar}...`) has exactly the same number of tokens as its expansion. However I think that it's easier to read. The meaning of `\Name` as a black box is: "construct a name from second parameter and then pass it to your first parameter as a parameter".

`\@stripstring` is used to get tokens building name of a macro without leading backslash.

```
\def\Name#1#2{\expandafter#1\csname#2\endcsname}
\def\@stripstring{\expandafter\@gobble\string}
```

8.2.5 Miscellaneous variables

`\sourceFileName` The macro `\sourceFileName` is used to store the name of the current input file.

`\batchfile` The macro `\batchfile` is used to store the name of the batch file.

`\inLine` The macro `\inLine` is used to store the lines, read from the input file, before further processing.

`\answer` When some interaction with the user is needed the macro `\answer` is used to store his response.

`\tmp` Sometimes something has to be temporarily stored in a control sequence. For these purposes the control sequence `\tmp` is used.

8.3 Support macros

8.3.1 The stack mechanism

It is possible to have 'nested guards'. This means that within a block of optionally included code a subgroup is only included when an additional option is specified. To keep track of the nesting of the guards the currently 'open' guard can be pushed on the stack `\guardStack` and later popped off the stack again. The macros that implement this stack mechanism are loosely based on code that is developed in the context of the LaTeX3 project.

To be able to implement a stack mechanism we need a couple of support macros.

`\eltStart` `\eltEnd` The macros `\eltStart` and `\eltEnd` are used to delimit a stack element. They are both empty.

```
\def\eltStart{}
\def\eltEnd{}
```

`\qStop` The macro `\qStop` is a so-called 'quark', a macro that expands to itself[5].

```
\def\qStop{\qStop}
```

`\pop` The macro `\pop`⟨*stack*⟩⟨*cs*⟩ 'pops' the top element from the stack. It assigns the value of the top element to ⟨*cs*⟩ and removes it from ⟨*stack*⟩. When ⟨*stack*⟩ is empty a warning is issued and ⟨*cs*⟩ is assigned an empty value.

```
\def\pop#1#2{%
  \ifx#1\empty
```

[5] The concept of 'quarks' is developed for the LaTeX3 project.

```
    \Msg{Warning: Found end guard without matching begin}%
    \let#2\empty
  \else
```

To be able to 'peel' off the first guard we use an extra macro `\popX` that receives both the expanded and the unexpanded stack in its arguments. The expanded stack is delimited with the quark `\qStop`.

```
    \def\tmp{\expandafter\popX #1\qStop #1#2}%
    \expandafter\tmp\fi}
```

`\popX` When the stack is expanded the elements are surrounded with `\eltStart` and `\eltEnd`. The first element of the stack is assigned to `#4`.

```
\def\popX\eltStart #1\eltEnd #2\qStop #3#4{\def#3{#2}\def#4{#1}}
```

`\push` Guards can be pushed on the stack using the macro `\push`⟨*stack*⟩⟨*guard*⟩. Again we need a secondary macro (`\pushX`) that has both the expanded and the unexpanded stack as arguments.

```
\def\push#1#2{\expandafter\pushX #1\qStop #1{\eltStart #2\eltEnd}}
```

`\pushX` The macro `\pushX` picks up the complete expansion of the stack as its first argument and places the guard in `#3` on the 'top'.

```
\def\pushX #1\qStop #2#3{\def #2{#3#1}}
```

8.3.2 Programming structures

`\forlist` When the program is used in interactive mode the user can supply a list of files that have to be processed. In order to process this list a for-loop is needed. This implementation of such a programming construct is based on the use of the `\loop{`⟨*body*⟩`}\repeat` macro that is defined in plain TEX. The syntax for this loop is:

`\for`⟨*control sequence*⟩ `:=` ⟨*list*⟩ `\do`
⟨*body*⟩
`\od`

The ⟨*list*⟩ should be a comma separated list.

The first actions that have to be taken are to set the switch `\ifForlist` to ⟨*true*⟩ and to store the loop condition in the macro `\ListCondition`. This is done using an `\edef` to allow for a control sequence that contains a ⟨*list*⟩.

```
\def\forlist#1:=#2\do#3\od{%
    \edef\ListCondition{#2}%
    \Forlisttrue
```

Then we start the loop. We store the first element from the `\ListCondition` in the macro that was supplied as the first argument to `\forlist`. This element is then removed from the `\ListCondition`.

```
    \loop
      \edef#1{\expandafter\FirstElt\ListCondition,\empty.}%
      \edef\ListCondition{\expandafter\OtherElts\ListCondition,\empty.}%
```

When the first element from the ⟨*list*⟩ is empty, we are done processing, so we switch `\ifForlist` to ⟨*false*⟩. When it is not empty we execute the third argument that should contain TEX commands to execute.

```
      \ifx#1\empty \Forlistfalse \else#3\fi
```

Finally we test the switch `\ifForlist` to decide whether the loop has to be continued.

```
      \ifForlist
    \repeat}
```

`\FirstElt` The macro `\FirstElt` is used to get the first element from a comma-separated list.

```
\def\FirstElt#1,#2.{#1}
```

\OtherElts The macro \OtherElts is used to get all elements *but* the first element from a comma-separated list.

```
\def\OtherElts#1,#2.{#2}
```

\whileswitch When the program is used in interactive mode the user might want to process several files with different options or extensions. This goal could be reached by running the program several times, but it is more user-friendly to ask if he would like to process more files when we are done processing his last request. To accomplish this we need the implementation of a `while`-loop. Again plain TEX's `\loop{`⟨*body*⟩`}\repeat` is used to implement this programming structure.

The syntax for this loop is:

`\whileswitch`⟨*switch*⟩ `\fi` ⟨*list*⟩ `{`⟨*body*⟩`}`

The first argument to this macro has to be a switch, defined using `\newif`; the second argument contains the statements to execute while the switch evaluates to ⟨*true*⟩.

```
\def\whileswitch#1\fi#2{#1\loop#2#1\repeat\fi}
```

8.3.3 Output streams allocator

For each of sixteen output streams available we have a macro named `\s@0` through `\s@15` saying if the stream is assigned to a file (1) or not (0). Initially all streams are not assigned.

We also declare 16 counters which will be needed by the conditional code inclusion algorithm.

```
\ifx\@tempcnta\undefined \newcount\@tempcnta \fi
\@tempcnta=0
\loop
\Name\chardef{s@\number\@tempcnta}=0
\csname newcount\expandafter\endcsname%
  \csname off@\number\@tempcnta\endcsname
\advance\@tempcnta1
\ifnum\@tempcnta<16\repeat
```

We will use *The TEXbook* style list to search through streams.

```
\let\s@do\relax
\edef\@outputstreams{%
  \s@do\Name\noexpand{s@0}\s@do\Name\noexpand{s@1}%
  \s@do\Name\noexpand{s@2}\s@do\Name\noexpand{s@3}%
  \s@do\Name\noexpand{s@4}\s@do\Name\noexpand{s@5}%
  \s@do\Name\noexpand{s@6}\s@do\Name\noexpand{s@7}%
  \s@do\Name\noexpand{s@8}\s@do\Name\noexpand{s@9}%
  \s@do\Name\noexpand{s@10}\s@do\Name\noexpand{s@11}%
  \s@do\Name\noexpand{s@12}\s@do\Name\noexpand{s@13}%
  \s@do\Name\noexpand{s@14}\s@do\Name\noexpand{s@15}%
  \noexpand\@nostreamerror
  }
```

\@nostreamerror \@streamfound When `\@outputstreams` is executed `\s@do` is defined to do something on condition of some test. If condition always fails macro `\@nostreamerror` on the end of the list causes an error. When condition succeeds `\@streamfound` is called, which gobbles rest of the list including the ending `\@nostreamerror`. It also gobbles `\fi` ending the condition, so the `\fi` is reinserted.

```
\def\@nostreamerror{\errmessage{No more output streams!}}
\def\@streamfound#1\@nostreamerror{\fi}
```

`\@stripstr` is auxiliary macro eating characters `\s@` (backslash,s,@). It is defined in somewhat strange way since `\s@` must have all category code 12 (other). This macro is used to extract stream numbers from stream names.

```
\bgroup\edef\x{\egroup
 \def\noexpand\@stripstr\string\s@{}}
\x
```

\StreamOpen
\StreamPut
\StreamClose

Here is stream opening operator. Its parameter should be a macro named the same as the external file being opened. E.g., to write to file `foo.tex` use `\StreamOpen\foo`, then `\StreamPut\foo` and `\StreamClose\foo`.

```
\chardef\stream@closed=16
\def\StreamOpen#1{%
  \chardef#1=\stream@closed
  \def\s@do##1{\ifnum##1=0
    \chardef#1=\expandafter\@stripstr\string##1 %
    \global\chardef##1=1 %
    \immediate\openout#1=\csname pth@\@stripstring#1\endcsname %
    \@streamfound
    \fi}
  \@outputstreams
  }
\def\StreamClose#1{%
  \immediate\closeout#1%
  \def\s@do##1{\ifnum#1=\expandafter\@stripstr\string##1 %
    \global\chardef##1=0 %
    \@streamfound
    \fi}
  \@outputstreams
  \chardef#1=\stream@closed
  }
\def\StreamPut{\immediate\write}
```

8.3.4 Input and Output

\maybeMsg
\showprogress
\keepsilent

When this program is used it can optionally show its progress on the terminal. In that case it will write a special character to the terminal (and the transcript file) for each input line. This option is on by default when statistics are included in `docstrip.tex`. It is off when statistics are excluded. The commands `\showprogress` and `\keepsilent` can be used to choose otherwise.

```
\def\showprogress{\let\maybeMsg\message}
\def\keepsilent{\let\maybeMsg\@gobble}
⟨*stats⟩
\showprogress
⟨/stats⟩
⟨-stats⟩\keepsilent
```

\Msg

For displaying messages on the terminal the macro `\Msg` is defined to write *immediately* to `\ttyout`.

```
\def\Msg{\immediate\write\ttyout}
```

\Ask

The macro `\Ask{`⟨*cs*⟩`}{`⟨*string*⟩`}` is a slightly modified copy of the LaTeX macro `\typein`. It is used to ask the user a question. The ⟨*string*⟩ will be displayed on his terminal and the response will be stored in the ⟨*cs*⟩. The trailing space left over from the carriage return is stripped off by the macro `\strip`. If the user just types a carriage return, the result will be an empty macro.

```
\def\iden#1{#1}
\def\strip#1#2 \@gobble{\def #1{#2}}
\def\@defpar{\par}
\def\Ask#1#2{%
    \message{#2}\read\ttyin to #1\ifx#1\@defpar\def#1{}\else
    \iden{\expandafter\strip
       \expandafter#1#1\@gobble\@gobble} \@gobble\fi}
```

\OriginalAsk

```
\let\OriginalAsk=\Ask
```

\askonceonly

```
\def\askonceonly{%
  \def\Ask##1##2{%
    \OriginalAsk{##1}{##2}%
    \global\let\Ask\OriginalAsk
    \Ask\noprompt{%
      By default you will be asked this question for every file.^^J%
      If you enter `y' now,^^J%
      I will asssume `y' for all future questions^^J%
      without prompting.}%
    \ifx\y\noprompt\let\noprompt\yes\fi
    \ifx\yes\noprompt\gdef\Ask####1####2{\def####1{y}}\fi}}
```

8.3.5 Miscellaneous

\@gobble A macro that has an argument and puts it in the bitbucket.

```
\def\@gobble#1{}
```

\Endinput When a doc file contains a \endinput on a line by itself this normally means that anything following in this file should be ignored. Therefore we need a macro containing \endinput as its replacement text to check this against \inLine (the current line from the current input file). Of course the backslash has to have the correct \catcode. One way of doing this is feeding \\ to the \string operation and afterwards removing one of the \ characters.

```
\edef\Endinput{\expandafter\@gobble\string\\endinput}
```

\makeOther During the process of reading a file with TEX code the category code of all special characters has to be changed to ⟨*other*⟩. The macro \makeOther serves this purpose.

```
\def\makeOther#1{\catcode`#1=12\relax}
```

\end For now we want the DocStrip program to be compatible with both plain TEX and LATEX. LATEX hides plain TEX's \end command and calls it \@@end. We unhide it here.

```
\ifx\undefined\@@end\else\let\end\@@end\fi
```

\@addto A macro extending macro's definition. The trick with \csname is necessary to get around \newtoks being outer in plain TEX and LATEX version 2.09.

```
\ifx\@temptokena\undefined \csname newtoks\endcsname\@temptokena\fi

\def\@addto#1#2{%
  \@temptokena\expandafter{#1}%
  \edef#1{\the\@temptokena#2}}
```

\@ifpresent This macro checks if its first argument is present on a list passed as the second argument. Depending on the result it executes either its third or fourth argument.

```
\def\@ifpresent#1#2#3#4{%
  \def\tmp##1#1##2\qStop{\ifx!##2!}%
  \expandafter\tmp#2#1\qStop #4\else #3\fi
  }
```

\tospaces This macro converts its argument delimited with \secapsot to appropriate number of spaces. We need this for smart displaying messages on the screen.

\@spaces are used when we need many spaces in a row.

```
\def\tospaces#1{%
  \ifx#1\secapsot\secapsot\fi\space\tospaces}
\def\secapsot\fi\space\tospaces{\fi}
\def\@spaces{\space\space\space\space\space}
```

`\uptospace` This macro extracts from its argument delimited with `\qStop` part up to first occurence of space.

```
\def\uptospace#1 #2\qStop{#1}
```

`\afterfi` This macro can be used in conditionals to perform some actions (its first parameter) after the condition is completed (i.e. after reading the matching `\fi`. Second parameter is used to gobble the rest of `\if ... \fi` construction (some `\else` maybe). Note that this won't work in nested `\if`s!

```
\def\afterfi#1#2\fi{\fi#1}
```

`\@ifnextchar` This is one of LaTeX's macros not defined by plain. My devious definition differs from the standard one but functionality is the same.

```
\def\@ifnextchar#1#2#3{\bgroup
  \def\reserved@a{\ifx\reserved@c #1 \aftergroup\@firstoftwo
    \else \aftergroup\@secondoftwo\fi\egroup
    {#2}{#3}}%
  \futurelet\reserved@c\@ifnch
  }
\def\@ifnch{\ifx \reserved@c \@sptoken \expandafter\@xifnch
      \else \expandafter\reserved@a
      \fi}
\def\@firstoftwo#1#2{#1}
\def\@secondoftwo#1#2{#2}
\iden{\let\@sptoken= } %
\iden{\def\@xifnch} {\futurelet\reserved@c\@ifnch}
```

`\kernel@ifnextchar` The 2003/12/01 release of LaTeX incorporated this macro to avoid problems with `amsmath` but this also means that we have to perform the same trick here when people use LaTeX on a installation file containing `\ProvidesFile`.

```
\let\kernel@ifnextchar\@ifnextchar
```

8.4 The evaluation of boolean expressions

For clarity we repeat here the syntax for the boolean expressions in a somewhat changed but equivalent way:

$$\begin{aligned}
\langle \textit{Expression} \rangle &::= \langle \textit{Secondary} \rangle \mid \langle \textit{Secondary} \rangle \ \{\texttt{|}, \texttt{,}\} \ \langle \textit{Expression} \rangle \\
\langle \textit{Secondary} \rangle &::= \langle \textit{Primary} \rangle \mid \langle \textit{Primary} \rangle \ \texttt{\&} \ \langle \textit{Secondary} \rangle \\
\langle \textit{Primary} \rangle &::= \langle \textit{Terminal} \rangle \mid \texttt{!}\langle \textit{Primary} \rangle \mid \texttt{(}\langle \textit{Expression} \rangle\texttt{)}
\end{aligned}$$

The `|` stands for disjunction, the `&` stands for conjunction and the `!` stands for negation. The $\langle \textit{Terminal} \rangle$ is any sequence of letters and evaluates to $\langle \textit{true} \rangle$ iff it occurs in the list of options that have to be included.

Since we can generate multiple output files from one input, same guard expressions can be computed several times with different options. For that reason we first "compile" the expression to the form of one parameter macro `\Expr` expanding to nested `\if`s that when given current list of options produces 1 or 0 as a result. The idea is to say `\if1\Expr{`$\langle$*current set of options*$\rangle$`}...\fi` for all output files.

Here is a table recursively defining translations for right sides of the grammar. $\tau(X)$ denotes translation of X.

$$\begin{aligned}
\tau(\langle \textit{Terminal} \rangle) &= \texttt{\textbackslash t@<Terminal>,\#1,<Terminal>,\textbackslash qStop} \\
\tau(\texttt{!}\langle \textit{Primary} \rangle) &= \texttt{\textbackslash if1}\ \tau(\langle \textit{Primary} \rangle)\ \texttt{0\textbackslash else1\textbackslash fi} \\
\tau(\texttt{(}\langle \textit{Expression} \rangle\texttt{)}) &= \tau(\langle \textit{Expression} \rangle) \\
\tau(\langle \textit{Primary} \rangle\texttt{\&}\langle \textit{Secondary} \rangle) &= \texttt{\textbackslash if0}\ \tau(\langle \textit{Primary} \rangle)\ \texttt{0\textbackslash else}\ \tau(\langle \textit{Secondary} \rangle)\ \texttt{\textbackslash fi} \\
\tau(\langle \textit{Secondary} \rangle\texttt{|}\langle \textit{Expression} \rangle) &= \texttt{\textbackslash if1}\ \tau(\langle \textit{Secondary} \rangle)\ \texttt{1\textbackslash else}\ \tau(\langle \textit{Expression} \rangle)\ \texttt{\textbackslash fi}
\end{aligned}$$

`\t@<Terminal>` denotes macro with name constructed from `t@` with appended tokens of terminal. E.g., for terminal `foo` the translation would be

```
\t@foo,#1,foo,\qStop
```

This will end up in definition of `\Expr`, so `#1` here will be substituted by current list of options when `\Expr` is called. Macro `\t@foo` would be defined as

```
\def\t@foo#1,foo,#2\qStop{\ifx>#2>0\else1\fi}
```

When called as above this will expand to `1` if `foo` is present on current list of options and to `0` otherwise.

Macros below work in "almost expand-only" mode i.e. expression is analyzed only by expansion but we have to define some macros on the way (e.g., `\Expr` and `\t@foo`).

The first parameter of each of these macros is "continuation" (in the sense similar to the language SCHEME). Continuation is a function of at least one argument (parameter) being the value of previous steps of computation. For example macro `\Primary` constructs translation of ⟨*Primary*⟩ expression. When it decides that expression is completed it calls its continuation (its first argument) passing to it whole constructed translation. Continuation may expect more arguments if it wants to see what comes next on the input.

We will perform recursive descent parse, but definitions will be presented in bottom-up order.

`\Terminal` ⟨*Terminal*⟩s are recognized by macro `\Terminal`. The proper way of calling it is `\Terminal{`⟨*current continuation*⟩`}{}`. Parameters are: continuation, ⟨*Terminal*⟩ so far and next character from the input. Macro checks if `#3` is one of terminal-ending characters and then takes appropriate actions. Since there are 7 ending chars and probably one `\csname` costs less than 7 nested `\if`s we construct a name and check if it is defined.

We must expand `\ifx` completely before taking next actions so we use `\afterfi`.

```
\def\Terminal#1#2#3{%
  \expandafter\ifx\csname eT@#3\endcsname\relax
```

If condition is true `#3` belongs to current ⟨*Terminal*⟩ so we append it to ⟨*Terminal*⟩-so-far and call `\Terminal` again.

```
    \afterfi{\Terminal{#1}{#2#3}}\else
```

When condition is false it's time to end the ⟨*Terminal*⟩ so we call macro `\TerminalX`. Next character is reinserted to the input.

In both cases continuation is passed unchanged.

```
    \afterfi{\TerminalX{#1}{#2}#3}\fi
}
```

`\eT@` Here we define macros marking characters that cannot appear inside terminal. The value is not important as long as it is different from `\relax`.

```
\Name\let{eT@>}=1
\Name\let{eT@&}=1 \Name\let{eT@!}=1
\Name\let{eT@|}=1 \Name\let{eT@,}=1
\Name\let{eT@(}=1 \Name\let{eT@)}=1
```

`\TerminalX` This macro should end scanning of ⟨*Terminal*⟩. Parameters are continuation and gathered tokens of ⟨*Terminal*⟩.

Macro starts by issuing an error message if ⟨*Terminal*⟩ is empty.

```
\def\TerminalX#1#2{%
  \ifx>#2> \errmessage{Error in expression: empty terminal}\fi
```

Then a macro is constructed for checking presence of ⟨*Terminal*⟩ in options list.

```
  \Name\def{t@#2}##1,#2,##2\qStop{\ifx>##2>0\else1\fi}%
```

And then current continuation is called with translation of ⟨*Terminal*⟩ according to formula

$$\tau(\langle \textit{Terminal} \rangle) = \texttt{\t@<Terminal>,\#1,<Terminal>,\qStop}$$

```
  #1{\Name\noexpand{t@#2},##1,#2,\noexpand\qStop}%
  }
```

`\Primary` Parameters are continuation and next character from the input.

According to the syntax ⟨*Primari*⟩es can have three forms. This makes us use even more dirty tricks than usual. Note the `\space` after a series of `\ifx`s. This series produces an one digit number of case to be executed. The number is given to `\ifcase` and `\space` stops TEX scanning for a ⟨*number*⟩. Use of `\ifcase` gives possibility to have one of three actions selected without placing them in nested `\if`s and so to use `\afterfi`.

```
\def\Primary#1#2{%
  \ifcase \ifx!#20\else\ifx(#21\else2\fi\fi\space
```

First case is for `!` i.e. negated ⟨*Primary*⟩. In this case we call `\Primary` recursively but we create new continuation: macro `\NPrimary` that will negate result passed by `\Primary` and pass it to current continuation (`#1`).

```
  \afterfi{\Primary{\NPrimary{#1}}}\or
```

When next character is `(` we call `\Expression` giving it as continuation macro `\PExpression` which will just pass the result up but ensuring first that a `)` comes next.

```
  \afterfi{\Expression{\PExpression{#1}}}\or
```

Otherwise we start a ⟨*Terminal*⟩. `#2` is not passed as ⟨*Terminal*⟩-so-far but reinserted to input since we didn't check if it can appear in a ⟨*Terminal*⟩.

```
  \afterfi{\Terminal{#1}{}#2}\fi
  }
```

`\NPrimary` Parameters are continuation and previously computed ⟨*Primary*⟩.

This macro negates result of previous computations according to the rule

$$\tau(!\langle Primary\rangle) = \texttt{\if1}\,\tau(\langle Primary\rangle)\,\texttt{0\else1\fi}$$

```
\def\NPrimary#1#2{%
  #1{\noexpand\if1#20\noexpand\else1\noexpand\fi}%
  }
```

`\PExpression` Parameters: continuation, ⟨*Expression*⟩, next character from input. We are checking if character is `)` and then pass unchanged result to our continuation.

```
\def\PExpression#1#2#3{%
  \ifx)#3\else
    \errmessage{Error in expression: expected right parenthesis}\fi
  #1{#2}}
```

`\Secondary` Each ⟨*Secondary*⟩ expression starts with ⟨*Primary*⟩. Next checks will be performed by `\SecondaryX`.

```
\def\Secondary#1{%
  \Primary{\SecondaryX{#1}}}
```

`\SecondaryX` Parameters: continuation, translation of ⟨*Primary*⟩, next character. We start by checking if next character is `&`.

```
\bgroup\catcode`\&=12
\gdef\SecondaryX#1#2#3{%
  \ifx&#3%
```

If it is we should parse next ⟨*Secondary*⟩ and then combine it with results so far. Note that `\SecondaryXX` will have 3 parameters.

```
  \afterfi{\Secondary{\SecondaryXX{#1}{#2}}}\else
```

Otherwise ⟨*Secondary*⟩ turned out to be just ⟨*Primary*⟩. We call continuation passing to it translation of that ⟨*Primary*⟩ not forgetting to reinsert `#3` to the input as it does not belong here.

```
  \afterfi{#1{#2}#3}\fi
  }
\egroup
```

`\SecondaryXX` Parameters: continuation, translation of ⟨*Primary*⟩, translation of ⟨*Secondary*⟩. We construct translation of whole construction according to the rule:

$$\tau(\langle Primary\rangle\texttt{\&}\langle Secondary\rangle) = \texttt{\if0}\,\tau(\langle Primary\rangle)\,\texttt{0\else}\,\tau(\langle Secondary\rangle)\,\texttt{\fi}$$

and pass it to our continuation.

```
\def\SecondaryXX#1#2#3{%
  #1{\noexpand\if0#20\noexpand\else#3\noexpand\fi}}
```

`\Expression` Every ⟨*Expression*⟩ starts with ⟨*Secondary*⟩. We construct new continuation and pass it to `\Secondary`.

```
\def\Expression#1{%
  \Secondary{\ExpressionX{#1}}}
```

`\ExpressionX` Parameters: continuation, translation of ⟨*Secondary*⟩, next character. We perform check if character is `|` or `,`.

```
\def\ExpressionX#1#2#3{%
  \if0\ifx|#31\else\ifx,#31\fi\fi0
```

If it is not ⟨*Expression*⟩ is just a ⟨*Secondary*⟩. We pass its translation to continuation and reinsert `#3`.

```
  \afterfi{#1{#2}#3}\else
```

If we are dealing with complex ⟨*Expression*⟩ we should parse another `\Expression` now.

```
  \afterfi{\Expression{\ExpressionXX{#1}{#2}}}\fi
  }
```

`\ExpressionXX` Parameters: continuation, translation of ⟨*Secondary*⟩, translation of ⟨*Expression*⟩. We finish up translating of ⟨*Expression*⟩ according to the formula:

$$\tau(\langle Secondary\rangle\texttt{|}\langle Expression\rangle) = \texttt{\if1}\,\tau(\langle Secondary\rangle)\,\texttt{1\else}\,\tau(\langle Expression\rangle)\,\texttt{\fi}$$

```
\def\ExpressionXX#1#2#3{%
  #1{\noexpand\if1#21\noexpand\else#3\noexpand\fi}}
```

`\StopParse` Here is initial continuation for whole parse process. It will be used by `\Evaluate`. Note that we assume that expression has `>` on its end. This macro eventually defines `\Expr`. Parameters: translation of whole ⟨*Expression*⟩ and next character from input.

```
\def\StopParse#1#2{%
  \ifx>#2 \else\errmessage{Error in expression: spurious #2}\fi
  \edef\Expr##1{#1}}
```

`\Evaluate` This macro is used to start parsing. We call `\Expression` with continuation defined above. On end of expression we append a `>`.

```
\def\Evaluate#1{%
  \Expression\StopParse#1>}
```

8.5 Processing the input lines

`\normalLine` The macro `\normalLine` writes its argument (which has to be delimited with `\endLine`) on all active output files i.e. those with off-counters equal to zero. If statistics are included, the counter `\codeLinesPassed` is incremented by 1.

```
\def\normalLine#1\endLine{%
⟨*stats⟩
  \advance\codeLinesPassed\@ne
⟨/stats⟩
  \maybeMsg{.}%
  \def\inLine{#1}%
  \let\do\putline@do
  \activefiles
  }
```

\putline@do This is a value for \do when copying line to output files.

```
\def\putline@do#1#2#3{%
  \StreamPut#1{\inLine}}
```

\removeComment The macro \removeComment throws its argument (which has to be delimited with \endLine) away. When statistics are included in the program the removed comment is counted.

```
%
\def\removeComment#1\endLine{%
⟨*stats⟩
    \advance\commentsRemoved\@ne
⟨/stats⟩
    \maybeMsg{\perCent}}
```

\putMetaComment If a line starts with two consecutive percent signs, it is considered to be a *MetaComment*. Such a comment line is passed on to the output file unmodified.

```
\bgroup\catcode`\%=12
\iden{\egroup
\def\putMetaComment%}#1\endLine{%
```

If statistics are included the line is counted.

```
⟨*stats⟩
  \advance\commentsPassed\@ne
⟨/stats⟩
```

The macro \putMetaComment has one argument, delimited with \endLine. It brings the source line with %% stripped. We prepend to it \MetaPrefix (which can be different from %%) and send the line to all active files.

```
  \edef\inLine{\MetaPrefix#1}%
  \let\do\putline@do
  \activefiles
  }
```

\processLine Each line that is read from the input stream has to be processed to see if it has to be written on the output stream. This task is performed by calling the macro \processLine. In order to do that, it needs to check whether the line starts with a '%'. Therefore the macro is globally defined within a group. Within this group the category code of '%' is changed to 12 (other). Because a comment character is needed, the category code of '*' is changed to 14 (comment character).

The macro increments counter \processedLines by 1 if statistics are included. We cannot include this line with %<*stats> since the category of % is changed and the file must be loadable unstripped. So the whole definition is repeated embedded in guards.

The next token from the input stream is passed in #1. If it is a '%' further processing has to be done by \processLineX; otherwise this is normal (not commented out) line.

In either case the character read is reinserted to the input as it may have to be written out.

```
⟨*!stats⟩
\begingroup
\catcode`\%=12 \catcode`\*=14
\gdef\processLine#1{*
  \ifx%#1
    \expandafter\processLineX
  \else
    \expandafter\normalLine
  \fi
  #1}
\endgroup
⟨/!stats⟩
⟨*stats⟩
```

```
\begingroup
\catcode`\%=12 \catcode`\*=14
\gdef\processLine#1{*
  \advance\processedLines\@ne
  \ifx%#1
    \expandafter\processLineX
  \else
    \expandafter\normalLine
  \fi
  #1}
\endgroup
⟨/stats⟩
```

\processLineX This macro is also defined within a group, because it also has to check if the next token in the input stream is a '%' character.

If the second token in the current line happens to be a '%', a ⟨*MetaComment*⟩ has been found. This has to be copied in its entirety to the output. Another possible second character is '<', which introduces a guard expression. The processing of such an expression is started by calling \checkOption.

When the token was neither a '%' nor a '<', the line contains a normal comment that has to be removed.

We express conditions in such a way that all actions appear on first nesting level of \ifs. In such conditions just one expandafter pushes us outside whole construction. A thing to watch here is \relax. It stops search for numeric constant. If it wasn't here TEX would expand the first case of \ifcase before knowing the value.

```
\begingroup
\catcode`\%=12 \catcode`\*=14
\gdef\processLineX%#1{*
  \ifcase\ifx%#10\else
         \ifx<#11\else 2\fi\fi\relax
    \expandafter\putMetaComment\or
    \expandafter\checkOption\or
    \expandafter\removeComment\fi
  #1}
\endgroup
```

8.6 The handling of options

\checkOption When the macros that process a line have found that the line starts with '%<', a guard line has been encountered. The first character of a guard can be an asterisk (*), a slash (/) a plus (+), a minus (-), a less-than sign (<) starting verbatim mode or any other character that can be found in an option name. This means that we have to peek at the next token and decide what kind of guard we have.

We reinsert #1 as it may be needed by \doOption.

```
\def\checkOption<#1{%
  \ifcase
    \ifx*#10\else \ifx/#11\else
    \ifx+#12\else \ifx-#13\else
    \ifx<#14\else 5\fi\fi\fi\fi\fi\relax
  \expandafter\starOption\or
  \expandafter\slashOption\or
  \expandafter\plusOption\or
  \expandafter\minusOption\or
  \expandafter\verbOption\or
  \expandafter\doOption\fi
  #1}
```

\doOption When no guard modifier is found by \checkOptions, the macro \doOption is called. It evaluates a boolean expression. The result of this evaluation is stored

in \Expr. The guard only affects the current line, so \do is defined in such a way that depending on the result of the test \if1\Expr{⟨*options*⟩}, the current line is either copied to the output stream or removed. Then the test is computed for all active output files.

```
\def\doOption#1>#2\endLine{%
  \maybeMsg{<#1 . >}%
  \Evaluate{#1}%
  \def\do##1##2##3{%
    \if1\Expr{##2}\StreamPut##1{#2}\fi
    }%
  \activefiles
  }
```

\plusOption When a '+' is found as a guard modifier, \plusOption is called. This macro is very similar to \doOption, the only difference being that displayed message now contains '+'.

```
\def\plusOption+#1>#2\endLine{%
  \maybeMsg{<+#1 . >}%
  \Evaluate{#1}%
  \def\do##1##2##3{%
    \if1\Expr{##2}\StreamPut##1{#2}\fi
    }%
  \activefiles
  }
```

\minusOption When a '-' is found as a guard modifier, \minusOption is called. This macro is very similar to \plusOption, the difference is that condition is negated.

```
\def\minusOption-#1>#2\endLine{%
  \maybeMsg{<-#1 . >}%
  \Evaluate{#1}%
  \def\do##1##2##3{%
    \if1\Expr{##2}\else \StreamPut##1{#2}\fi
    }%
  \activefiles
  }
```

\starOption When a '*' is found as a guard modifier, \starOption is called. In this case a block of code will be included in the output on the condition that the guard expression evaluates to ⟨*true*⟩.

The current line is gobbled as #2, because it only contains the guard and possibly a comment.

```
\def\starOption*#1>#2\endLine{%
```

First we optionally write a message to the terminal to indicate that a new option starts here.

```
  \maybeMsg{<*#1}%
```

Then we push the current contents of \blockHead on the stack of blocks, \guardStack and increment the counter \blockLevel to indicate that we are now one level of nesting deeper.

```
  \expandafter\push\expandafter\guardStack\expandafter{\blockHead}%
  \advance\blockLevel\@ne
```

The guard for this block of code is now stored in \blockHead.

```
  \def\blockHead{#1}%
```

Now we evaluate guard expression for all output files updating off-counters. Then we create new list of active output files. Only files that were active in the outer block can remain active now.

```
  \Evaluate{#1}%
  \let\do\checkguard@do
  \outputfiles
```

```
\let\do\findactive@do
\edef\activefiles{\activefiles}
}
```

`\checkguard@do` This form of `\do` updates off-counts according to the value of guard expression.

```
\def\checkguard@do#1#2#3{%
```

If this block of code occurs inside another block of code that is *not* included in the output, we increment the off counter. In that case the guard expression will not be evaluated, because a block inside another block that is excluded from the output will also be excluded, regardless of the evaluation of its guard.

```
\ifnum#3>0
  \advance#3\@ne
```

When the off count has value 0, we have to evaluate the guard expression. If the result is ⟨*false*⟩ we increase the off-counter.

```
\else
  \if1\Expr{#2}\else
    \advance#3\@ne\fi
\fi}
```

`\findactive@do` This form of `\do` picks elements of output files list which have off-counters equal to zero.

```
\def\findactive@do#1#2#3{%
\ifnum#3=0
  \noexpand\do#1{#2}#3\fi}
```

`\slashOption` The macro `\slashOption` is the counterpart to `\starOption`. It indicates the end of a block of conditionally included code. We store the argument in the temporary control sequence `\tmp`.

```
\def\slashOption/#1>#2\endLine{%
\def\tmp{#1}%
```

When the counter `\blockLevel` has a value less than 1, this 'end-of-block' line has no corresponding 'start-of-block'. Therefore we signal an error and ignore this end of block.

```
\ifnum\blockLevel<\@ne
  \errmessage{Spurious end block </\tmp> ignored}%
```

Next we compare the contents of `\tmp` with the contents of `\blockHead`. The latter macro contains the last guard for a block of code that was encountered. If the contents match, we pop the previous guard from the stack.

```
\else
  \ifx\tmp\blockHead
    \pop\guardStack\blockHead
```

Is this the desired behaviour??

When the contents of the two macros don't match something is amiss. We signal this to the user, but accept the 'end-of-block'.

```
  \else
    \errmessage{Found </\tmp> instead of </\blockHead>}%
  \fi
```

When the end of a block of optionally included code is encountered we optionally signal this on the terminal and decrement the counter `\blockLevel`.

```
  \maybeMsg{>}%
  \advance\blockLevel\m@ne
```

The last thing that has to be done is to decrement off-counters and make new list of active files. Now whole list of output files has to be searched since some inactive files could have been reactivated.

```
\let\do\closeguard@do
\outputfiles
\let\do\findactive@do
```

```
\edef\activefiles{\outputfiles}
\fi
}
```

`\closeguard@do` This macro decrements non-zero off-counters.

```
\def\closeguard@do#1#2#3{%
  \ifnum#3>0
    \advance#3\m@ne
  \fi}
```

`\verbOption` This macro is called when a line starts with `%<<`. It reads a bunch of lines in verbatim mode: the lines are passed unchanged to the output without even checking for starting `%`. This way of processing ends when a line containing only a percent sign followed by stop mark given on the `%<<` line is found.

```
\def\verbOption<#1\endLine{{%
  \edef\verbStop{\perCent#1}\maybeMsg{<<<}%
  \let\do\putline@do
  \loop
    \ifeof\inFile\errmessage{Source file ended while in verbatim
                             mode!}\fi
    \read\inFile to \inLine
  \if 1\ifx\inLine\verbStop 0\fi 1% if not inLine==verbStop
    \activefiles
    \maybeMsg{.}%
  \repeat
  \maybeMsg{>}%
  }}
```

8.7 Batchfile commands

DocStrip keeps information needed to control inclusion of sources in several list structures. Lists are macros expanding to a series of calls to macro `\do` with two or three parameters. Subsequently `\do` is redefined in various ways and list macros sometimes are executed to perform some action on every element, and sometimes are used inside an `\edef` to make new list of elements having some properties. For every input file ⟨*infile*⟩ the following lists are kept:

`\b@`⟨*infile*⟩ the "open list"—names of all output files such that their generation should start with reading of ⟨*infile*⟩,

`\o@`⟨*infile*⟩ the "output list"—names of all output files generated from that source together with suitable sets of options (guards),

`\e@`⟨*infile*⟩ the "close list"—names of all output files that should be closed when this source is read.

For every output file name ⟨*outfile*⟩ DocStrip keeps following information:

`\pth@`⟨*outfile*⟩ full pathname (including file name),

`\ref@`⟨*outfile*⟩ reference lines for the file,

`\in@`⟨*outfile*⟩ names of all source files separated with spaces (needed by `\InFileName`),

`\pre@`⟨*outfile*⟩ preamble template (as defined with `\declarepreamble`),

`\post@`⟨*outfile*⟩ postamble template.

`\generate` This macro executes its argument in a group. `\inputfiles` is a list of files to be read, `\filestogenerate` list of names of output files (needed for the message below). `\file`s contained in `#1` define `\inputfiles` in such a way that all that has to be done when the parameter is executed is to call this macro. `\inputfiles` command is called over and over again until no output files had to be postponed.

```
\def\generate#1{\begingroup
  \let\inputfiles\empty \let\filestogenerate\empty
  \let\file\@file
  #1
  \ifx\filestogenerate\empty\else
  \Msg{^^JGenerating file(s) \filestogenerate}\fi
  \def\inFileName{\csname in@\outFileName\endcsname}%
  \def\ReferenceLines{\csname ref@\outFileName\endcsname}%
  \processinputfiles
  \endgroup}
```

`\processinputfiles` This is a recurrent function which processes input files until they are all gone.

```
\def\processinputfiles{%
  \let\newinputfiles\empty
  \inputfiles
  \let\inputfiles\newinputfiles
  \ifx\inputfiles\empty\else
    \expandafter\processinputfiles
  \fi
  }
```

`\file` The first argument is the file to produce, the second argument contains the list of input files. Each entry should have the format `\from{`⟨*filename.ext*⟩`}{`⟨*options*⟩`}`.

The switch `\ifGenerate` is initially set to ⟨*true*⟩.

```
\def\file#1#2{\errmessage{Command `\string\file' only allowed in
                          argument to `\string\generate'}}
\def\@file#1{%
  \Generatetrue
```

Next we construct full path name for output file and check if we have to be careful about overwriting existing files. If the user specified `\askforoverwritetrue` we will ask him if he wants to overwrite an existing file. Otherwise we simply go ahead.

```
  \makepathname{#1}%
  \ifaskforoverwrite
```

We try to open a file with the name of the output file for reading. If this succeeds the file exists and we ask the user if he wants to overwrite the file.

```
    \immediate\openin\inFile\@pathname\relax
    \ifeof\inFile\else
      \Ask\answer{File \@pathname\space already exists
                  \ifx\empty\destdir somewhere \fi
                  on the system.^^J%
                  Overwrite it%
                  \ifx\empty\destdir\space if necessary\fi
                  ? [y/n]}%
```

We set the switch `\ifGenerate` according to his answer. We allow for both "`y`" and "`yes`".

```
      \ifx\y  \answer \else
      \ifx\yes\answer \else
                      \Generatefalse\fi\fi\fi
```

Don't forget to close the file just opened as we want to write to it.

```
    \closein\inFile
  \fi
```

If file is to be generated we save its destination pathname and pass control to macro `\@fileX`. Note that file name is turned into control sequence and `\else` branch is skipped before calling `\@fileX`.

```
  \ifGenerate
  \Name\let{pth@#1}\@pathname
  \@addto\filestogenerate{\@pathname\space}%
```

```
  \Name\@fileX{#1\expandafter}%
  \else
```

In case we were not allowed to overwrite an existing file we inform the user that we are *not* generating his file and we gobble `\from` specifications.

```
    \Msg{Not generating file \@pathname^^J}%
  \expandafter\@gobble
  \fi
  }
```

`\@fileX` We put name of current output file in `\curout` and initialize `\curinfiles` (the list of source files for this output file) to empty—these will be needed by `\from`. Then we start defining preamble for the current file.

```
\def\@fileX#1#2{%
  \chardef#1=\stream@closed
  \def\curout{#1}%
  \let\curinfiles\empty
  \let\curinnames\empty
  \def\curref{\MetaPrefix ^^J%
              \MetaPrefix\space The original source files were:^^J%
              \MetaPrefix ^^J}%
```

Next we execute second parameter. `\from`s will add reference lines to the preamble.

```
  \let\from\@from \let\needed\@needed
  #2%
  \let\from\err@from \let\needed\err@needed
```

We check order of input files.

```
  \checkorder
```

Each `\from` clause defines `\curin` to be its first parameter. So now `\curin` holds name of last input file for current output file. This means that current output file should be closed after processing `\curin`. We add `#1` to proper 'close list'.

```
  \Name\@addto{e@\curin}{\noexpand\closeoutput{#1}}%
```

Last we save all the interesting information about current file.

```
  \Name\let{pre@\@stripstring#1\expandafter}\currentpreamble
  \Name\let{post@\@stripstring#1\expandafter}\currentpostamble
  \Name\edef{in@\@stripstring#1}{\expandafter\iden\curinnames}
  \Name\edef{ref@\@stripstring#1}{\curref}
  }
```

`\checkorder` This macro checks if the order of files in `\curinfiles` agrees with that of `\inputfiles`. The coding is somewhat clumsy.

```
\def\checkorder{%
  \expandafter\expandafter\expandafter
  \checkorderX\expandafter\curinfiles
  \expandafter\qStop\inputfiles\qStop
  }
\def\checkorderX(#1)#2\qStop#3\qStop{%
  \def\tmp##1\readsource(#1)##2\qStop{%
    \ifx!##2! \order@error
    \else\ifx!#2!\else
       \checkorderXX##2%
    \fi\fi}%
  \def\checkorderXX##1\readsource(#1)\fi\fi{\fi\fi
    \checkorderX#2\qStop##1\qStop}%
  \tmp#3\readsource(#1)\qStop
  }
\def\order@error#1\fi\fi{\fi
  \errmessage{DOCSTRIP error: Incompatible order of input
              files specified for file
              `\iden{\expandafter\uptospace\curin} \qStop'.^^J
```

```
              Read DOCSTRIP documentation for explanation.^^J
              This is a serious problem, I'm exiting}\end
}
```

`\needed` `\@needed` This macro uniquizes name of an input file passed as a parameter and marks it as needed to be input. It is used internally by `\from`, but can also be issued in argument to `\file` to influence the order in which files are read.

```
\def\needed#1{\errmessage{\string\needed\space can only be used in
                argument to \string\file}}
\let\err@needed\needed
\def\@needed#1{%
  \edef\reserved@a{#1}%
  \expandafter\@need@d\expandafter{\reserved@a}}
\def\@need@d#1{%
  \@ifpresent{(#1)}\curinfiles
```

If `#1` is present on list of input files for current output file we add a space on end of its name and try again. The idea is to construct a name that will look different for TEX but will lead to the same file when seen by operating system.

```
    {\@need@d{#1 }}%
```

When it is not we check if `#1` is present in the list of files to be processed. If not we add it and initialize list of output files for that input and list of output files that should be closed when this file closes. We also add constructed name to `\curinfiles` and define `\curin` to be able to access constructed name from `\@from`.

```
    {\@ifpresent{\readsource(#1)}\inputfiles
       {}{\@addto\inputfiles{\noexpand\readsource(#1)}%
       \Name\let{b@#1}\empty
       \Name\let{o@#1}\empty
       \Name\let{e@#1}\empty}%
     \@addto\curinfiles{(#1)}%
     \def\curin{#1}}%
}
```

`\from` `\from` starts by adding a line to preamble for output file.

```
\def\from#1#2{\errmessage{Command `\string\from' only allowed in
                          argument to `\string\file'}}
\let\err@from\from
\def\@from#1#2{%
  \@addto\curref{\MetaPrefix\space #1 \if>#2>\else
                              \space (with options: `#2')\fi^^J}%
```

Then we mark the file as needed input file.

```
  \needed{#1}%
```

If this is the first `\from` in current `\file` (i.e. if the `\curinnames` so far is empty) the file name is added to the "open list" for the current input file. And `\do`⟨*current output*⟩`{`⟨*options*⟩`}` is appended to the list of output files for current input file.

```
  \ifx\curinnames\empty
     \Name\@addto{b@\curin}{\noexpand\openoutput\curout}%
  \fi
  \@addto\curinnames{ #1}%
  \Name\@addto{o@\curin}{\noexpand\do\curout{#2}}%
  }
```

`\readsource` This macro is called for each input file that is to be processed.

```
\def\readsource(#1){%
```

We try to open the input file. If this doesn't succeed, we tell the user so and nothing else happens.

```
  \immediate\openin\inFile\uptospace#1 \qStop\relax
  \ifeof\inFile
```

```
    \errmessage{Cannot find file \uptospace#1 \qStop}%
  \else
```

If statistics are included we nullify line counters

```
⟨*stats⟩
    \processedLines\z@
    \commentsRemoved\z@
    \commentsPassed\z@
    \codeLinesPassed\z@
⟨/stats⟩
```

When the input file was successfully opened, we try to open all needed output files by executing the "open list". If any of files couldn't be opened because of number of streams limits, their names are put into `\refusedfiles` list. This list subsequently becomes the open list for the next pass.

```
    \let\refusedfiles\empty
    \csname b@#1\endcsname
    \Name\let{b@#1}\refusedfiles
```

Now all output files that could be opened are open. So we go through the "output list" and for every open file we display a message and zero the off-counter, while closed files are appended to `\refusedfiles`.

```
    \Msg{} \def\@msg{Processing file \uptospace#1 \qStop}
    \def\change@msg{%
      \edef\@msg{\@spaces\@spaces\@spaces\space
        \expandafter\tospaces\uptospace#1 \qStop\secapsot}
      \let\change@msg\relax}
    \let\do\showfiles@do
    \let\refusedfiles\empty
    \csname o@#1\endcsname
```

If `\refusedfiles` is nonempty current source file needs reread, so we append it to `\newinputfiles`.

```
    \ifx\refusedfiles\empty\else
      \@addto\newinputfiles{\noexpand\readsource(#1)}
    \fi
```

Finally we define `\outputfiles` and construct off-counters names. Now `\do`s will have 3 parameters! All output files become active.

```
    \let\do\makeoutlist@do
    \edef\outputfiles{\csname o@#1\endcsname}%
    \let\activefiles\outputfiles
    \Name\let{o@#1}\refusedfiles
```

Now we change the category code of a lot of characters to ⟨*other*⟩ and make sure that no extra spaces appear in the lines read by setting the `\endlinechar` to -1.

```
    \makeOther\ \makeOther\\\makeOther\$%
    \makeOther\#\makeOther\^\makeOther\^^K%
    \makeOther\_\makeOther\^^A\makeOther\%%
    \makeOther\~\makeOther\{\makeOther\}\makeOther\&%
    \endlinechar-1\relax
```

Then we start a loop to process the lines in the file one by one.

```
    \loop
      \read\inFile to\inLine
```

The first thing we check is whether the current line contains an `\endinput`. To allow also real `\endinput` commands in the source file, `\endinput` is only recognized when it occurs directly at the beginning of a line.

```
      \ifx\inLine\Endinput
```

In this case we output a message to inform the programmer (in case this was a mistake) and end the loop immediately by setting `Continue` to ⟨*false*⟩. Note that `\endinput` is not placed into the output file. This is important in cases where the output file is generated from several `doc` files.

```
        \Msg{File #1 ended by \string\endinput.}%
        \Continuefalse
      \else
```

When the end of the file is found we have to interrupt the loop.

```
       \ifeof\inFile
         \Continuefalse
```

If the file did not end we check if the input line is empty. If it is, the counter `\emptyLines` is incremented.

```
       \else
         \Continuetrue
         \ifx\inLine\empty
            \advance\emptyLines\@ne
          \else
            \emptyLines\z@
          \fi
```

When the number of empty lines seen so far exceeds 1, we skip them. If it doesn't, the expansion of `\inLine` is fed to `\processLine` with `\endLine` appended to indicate the end of the line.

```
          \ifnum \emptyLines<2
            \expandafter\processLine\inLine\endLine
          \else
            \maybeMsg{/}%
          \fi
       \fi
      \fi
```

When the processing of the line is finished, we check if there is more to do, in which case we repeat the loop.

```
    \ifContinue
    \repeat
```

The input file is closed.

```
    \closein\inFile
```

We close output files for which this was the last input file.

```
  \csname e@#1\endcsname
```

If the user was interested in statistics, we inform him of the number of lines processed, the number of comments that were either removed or passed and the number of codelines that were written to the output file. Also the totals are updated.

```
⟨*stats⟩
    \Msg{Lines \space processed: \the\processedLines^^J%
         Comments removed: \the\commentsRemoved^^J%
         Comments \space passed: \the\commentsPassed^^J%
         Codelines passed: \the\codeLinesPassed^^J}%
      \global\advance\TotalprocessedLines  by \processedLines
      \global\advance\TotalcommentsRemoved by \commentsRemoved
      \global\advance\TotalcommentsPassed  by \commentsPassed
      \global\advance\TotalcodeLinesPassed by \codeLinesPassed
⟨/stats⟩
```

The `\NumberOfFiles` need to be known even if no statistics are gathered so we update it always.

```
      \global\advance\NumberOfFiles by \@ne
    \fi}
```

`\showfiles@do` A message is displayed on the terminal telling the user what we are about to do. For each open output file we display one line saying what options it is generated with and the off-counter associated with the file is zeroed. First line contains also name of input file. Names of output files that are closed are appended to `\refusedfiles`.

```
\def\showfiles@do#1#2{%
  \ifnum#1=\stream@closed
    \@addto\refusedfiles{\noexpand\do#1{#2}}%
  \else
    \Msg{\@msg
         \ifx>#2>\else\space(#2)\fi
         \space -> \@stripstring#1}
    \change@msg
  \csname off@\number#1\endcsname=\z@
  \fi
}
```

`\makeoutlist@do` This macro selects only open output files and constructs names for off-counters.

```
\def\makeoutlist@do#1#2{%
  \ifnum#1=\stream@closed\else
    \noexpand\do#1{#2}\csname off@\number#1\endcsname
  \fi}
```

`\openoutput` This macro opens output streams if possible.

```
\def\openoutput#1{%
```

If both maxfile counters are non-zero...

```
  \if 1\ifnum\@maxfiles=\z@ 0\fi
       \ifnum\@maxoutfiles=\z@ 0\fi1%
```

...the stream may be opened and counters decremented. But if that cannot be done...

```
    \advance\@maxfiles\m@ne
    \advance\@maxoutfiles\m@ne
    \StreamOpen#1%
    \WritePreamble#1%
  \else
```

...the file is added to the "refuse list".

```
     \@addto\refusedfiles{\noexpand\openoutput#1}%
  \fi
  }
```

`\closeoutput` This macro closes open output stream when it is no longer needed and increments maxfiles counters.

```
\def\closeoutput#1{%
  \ifnum#1=\stream@closed\else
    \WritePostamble#1%
    \StreamClose#1%
    \advance\@maxfiles\@ne
    \advance\@maxoutfiles\@ne
  \fi}
```

8.7.1 Preamble and postamble

`\ds@heading` This is a couple of lines, stating what file is being written and how it was created.

```
\def\ds@heading{%
  \MetaPrefix ^^J%
  \MetaPrefix\space This is file `\outFileName',^^J%
  \MetaPrefix\space  generated with the docstrip utility.^^J%
  }
```

`\AddGenerationDate` Older versions of DocStrip added the date that any file was generated and the version number of DocStrip. This confused some people as they mistook this for the version/date of the file that was being written. So now this information is not normally written, but a batch file may call this to get an old style header.

```
\def\AddGenerationDate{%
```

```
\def\ds@heading{%
  \MetaPrefix ^^J%
  \MetaPrefix\space This is file `\outFileName', generated %
        on <\the\year/\the\month/\the\day> ^^J%
  \MetaPrefix\space with the docstrip utility (\fileversion).^^J%
}}
```

\declarepreamble When a batch file is used the user can specify a preamble of his own that will be written to each file that is created. This can be useful to include an extra copyright notice in the stripped version of a file. Also a warning that both versions of a file should *always* be distributed together could be written to a stripped file by including it in such a preamble.

Every line that is written to `\outFile` that belongs to the preamble is preceded by two percent characters. This will prevent DocStrip from stripping these lines off the file.

The preamble should be started with the macro `\declarepreamble`; it is ended by `\endpreamble`. All processing is done within a group in order to be able to locally change some values.

`\ReferenceLines` is let equal `\relax` to be unexpandable.

```
\let\inFileName\relax
\let\outFileName\relax
\let\ReferenceLines\relax
\def\declarepreamble{\begingroup
\catcode`\^^M=13 \catcode`\ =12 %
\declarepreambleX}
{\catcode`\^^M=13 %
\gdef\declarepreambleX#1#2
\endpreamble{\endgroup%
  \def^^M{^^J\MetaPrefix\space}%
  \edef#1{\ds@heading%
          \ReferenceLines%
          \MetaPrefix\space\checkeoln#2\empty}}%
\gdef\checkeoln#1{\ifx^^M#1\else\expandafter#1\fi}%
}
```

\declarepostamble Just as a preamble can be specified in a batch file, the same can be done for a *post*amble.

The definition of `\declarepostamble` is very much like the definition above of `\declarepreamble`.

```
\def\declarepostamble{\begingroup
\catcode`\ =12 \catcode`\^^M=13
\declarepostambleX}
{\catcode`\^^M=13 %
\gdef\declarepostambleX#1#2
\endpostamble{\endgroup%
  \def^^M{^^J\MetaPrefix\space}%
  \edef#1{\MetaPrefix\space\checkeoln#2\empty^^J%
          \MetaPrefix ^^J%
          \MetaPrefix\space End of file `\outFileName'.%
  }}%
}
```

\usepreamble \usepostamble Macros for selecting [pre/post]amble to be used.

```
\def\usepreamble#1{\def\currentpreamble{#1}}
\def\usepostamble#1{\def\currentpostamble{#1}}
```

\nopreamble \nopostamble Shortcuts for disabling the writing of [pre/post]ambles. This is not done by disabling `\WritePreamble` or `\WritePostamble` since that wouldn't revertable afterwards. Instead the empty [pre/post]ambles are handled specially in those macros.

```
\def\nopreamble{\usepreamble\empty}
\def\nopostamble{\usepostamble\empty}
```

\preamble
\postamble

For backward compatibility we provide these macros defining default preamble and postamble.

```
\def\preamble{\usepreamble\defaultpreamble
  \declarepreamble\defaultpreamble}
\def\postamble{\usepostamble\defaultpostamble
  \declarepostamble\defaultpostamble}
```

\org@preamble
\org@postamble

Default values to use if nothing different is provided.

```
\declarepreamble\org@preamble

IMPORTANT NOTICE:

For the copyright see the source file.

Any modified versions of this file must be renamed
with new filenames distinct from \outFileName.

For distribution of the original source see the terms
for copying and modification in the file \inFileName.

This generated file may be distributed as long as the
original source files, as listed above, are part of the
same distribution. (The sources need not necessarily be
in the same archive or directory.)
\endpreamble

\edef\org@postamble{\string\endinput^^J%
  \MetaPrefix ^^J%
  \MetaPrefix\space End of file `\outFileName'.%
  }

\let\defaultpreamble\org@preamble
\let\defaultpostamble\org@postamble

\usepreamble\defaultpreamble
\usepostamble\defaultpostamble
```

\originaldefault

The default preamble header changed in v2.5 to allow distribution of generated files as long as source also distributed. If you need the original default, not allowing distribution of generated files add `\usepreamble\originaldefault` to your .ins files. Note then that your file can not be included in most TeX distributions on CD which are distributed 'pre-installed' with all LaTeX files extracted form th edocumented sources and moved to a suitable directory in TeX's search path.

```
\declarepreamble\originaldefault

IMPORTANT NOTICE:

For the copyright see the source file.

You are *not* allowed to modify this file.

You are *not* allowed to distribute this file.
For distribution of the original source see the terms
for copying and modification in the file \inFileName.

\endpreamble
```

\WritePreamble

```
\def\WritePreamble#1{%
```

We write out only non-empty preambles.

```
  \expandafter\ifx\csname pre@\@stripstring#1\endcsname\empty
  \else
    \edef\outFileName{\@stripstring#1}%
```

Then the reference lines that tell from what source file(s) the stripped file was created and user supplied preamble.

```
    \StreamPut#1{\csname pre@\@stripstring#1\endcsname}%
  \fi}
```

\WritePostamble Postamble attributed to #1 is written out. The last line written identifies the file again.

```
\def\WritePostamble#1{%
```

We write out only non-empty postambles.

```
  \expandafter\ifx\csname post@\@stripstring#1\endcsname\empty
  \else
    \edef\outFileName{\@stripstring#1}%
    \StreamPut#1{\csname post@\@stripstring#1\endcsname}%
  \fi}
```

8.8 Support for writing to specified directories

As we've seen before every output file is written to directory specifed by the value of \destdir current at the moment of this file's \file declaration.

\usedir This macro when called should translate its one argument into a directory name and define \destdir to that value. The default for \usedir is to ignore its argument and return name of current directory (if known). This can be changed by commands from docstrip.cfg file.

\showdirectory is used just to display directory name for user's information.

```
\def\usedir#1{\edef\destdir{\WriteToDir}}
\def\showdirectory#1{\WriteToDir}
```

\BaseDirectory This is config file command for specifing root directory of the TEX hierarchy. It enables the whole directory selecting mechanism by redefining \usedir. First make sure that the directory syntax commands have been set up by calling \@setwritedir, so that the value of \dirsep used by the \edef is (hopefully) correct.

```
\def\BaseDirectory#1{%
  \@setwritetodir
  \let\usedir\alt@usedir
  \let\showdirectory\showalt@directory
  \edef\basedir{#1\dirsep}}
```

\convsep This macro loops through slashes in its argument replacing them with current \dirsep. It should be called \convsep some/directory/name/\qStop (with slash on the end).

```
\def\convsep#1/#2\qStop{%
  #1\ifx\qStop#2\qStop \pesvnoc\fi\convsep\dirsep#2\qStop}
\def\pesvnoc#1\qStop{\fi}
```

\alt@usedir Directory name construction macro enabling writing to various directories.

```
\def\alt@usedir#1{%
  \Name\ifx{dir@#1}\relax
    \undefined@directory{#1}%
  \else
    \edef\destdir{\csname dir@#1\endcsname}%
  \fi}
\def\showalt@directory#1{%
  \Name\ifx{dir@#1}\relax
    \showundef@directory{#1}%
  \else\csname dir@#1\endcsname\fi}
```

\undefined@directory This macro comes into action when undefined label is spotted. The action is to raise an error and define \destdir to point to the current directory.

```
\def\undefined@directory#1{%
  \errhelp{docstrip.cfg should specify a target directory for^^J%
   #1 using \DeclareDir or \UseTDS.}%
  \errmessage{You haven't defined the output directory for `#1'.^^J%
            Subsequent files will be written to the current directory}%
  \let\destdir\WriteToDir
  }
\def\showundef@directory#1{UNDEFINED (label is #1)}
```

\undefined@TDSdirectory This happens when label is undefined while using TDS. The label is converted to use proper separators and appended to base directory name.

```
\def\undefined@TDSdirectory#1{%
  \edef\destdir{%
    \basedir\convsep#1/\qStop
  }}
\def\showundef@TDSdirectory#1{\basedir\convsep#1/\qStop}
```

\UseTDS Change of behaviour for undefined labels is done simply:

```
\def\UseTDS{%
  \@setwritetodir
  \let\undefined@directory\undefined@TDSdirectory
  \let\showundef@directory\showundef@TDSdirectory
  }
```

\DeclareDir This macro remaps some directory name to another.

```
\def\DeclareDir{\@ifnextchar*{\DeclareDirX}{\DeclareDirX\basedir*}}
\def\DeclareDirX#1*#2#3{%
  \@setwritetodir
  \Name\edef{dir@#2}{#1#3}}
```

8.8.1 Compatibility with older versions

\generateFile Main macro of previous versions of DocStrip.

```
\def\generateFile#1#2#3{{%
  \ifx t#2\askforoverwritetrue
  \else\askforoverwritefalse\fi
  \generate{\file{#1}{#3}}%
  }}
```

To support command files that were written for the first version of DocStrip the commands \include and \processFile are defined here. The use of this interface is not recommended as it may be removed in a future release of DocStrip.

\include To provide the DocStrip program with a list of options that should be included in the output the command \include{⟨*Options*⟩} can be used. This macro is meant to be used in conjunction with the \processFile command.

```
\def\include#1{\def\Options{#1}}
```

\processFile The macro \processFile{⟨*filename*⟩}{⟨*inext*⟩}{⟨*outext*⟩}{⟨*t*|*f*⟩} can be used when a single input file is used to produce a single output file. The macro is also used in the interactive mode of the DocStrip program.

The arguments ⟨*inext*⟩ and ⟨*outext*⟩ denote the extensions of the input and output files respectively. The fourth argument can be used to specify if an existing file should be overwritten without asking. If ⟨*t*⟩ is specified the program will ask for permission before overwriting an existing file.

This macro is defined using the more generic macro \generateFile.

```
\def\processFile#1#2#3#4{%
  \generateFile{#1.#3}{#4}{\from{#1.#2}{\Options}}}
```

\processfile
\generatefile

Early versions of DocStrip defined \processfile and \generatefile instead of the commands as they are defined in this version. To remain upwards compatible, we still provide these commands, but issue a warning when they are used.

```
\def\processfile{\Msg{%
    ^^Jplease use \string\processFile\space instead of
      \string\processfile!^^J}%
    \processFile}
\def\generatefile{\Msg{%
    ^^Jplease use \string\generateFile\space instead of
      \string\generatefile!^^J}%
    \generateFile}
```

8.9 Limiting open file streams

(This section was written by Mark Wooding)

\maxfiles

Some operating systems with duff libraries or other restrictions can't cope with all the files which DocStrip tries to output at once. A configuration file can say \maxfiles{⟨*number*⟩} to describe the maximum limit for the environment.

I'll need a counter for this value, so I'd better allocate one.

```
\newcount\@maxfiles
```

The configuration command \maxfiles is just slightly prettier than an assignment, for LaTeX people. It also gives me an opportunity to check that the limit is vaguely sensible. I need at least 4 streams:

1. A batch file.
2. A sub batch file, which LaTeX's installation utility uses.
3. An input stream for reading an unstripped file.
4. An output stream for writing a stripped file.

```
\def\maxfiles#1{%
  \@maxfiles#1\relax
  \ifnum\@maxfiles<4
    \errhelp{I'm not a magician.  I need at least four^^J%
             streams to be able to work properly, but^^J%
             you've only let me use \the\@maxfiles.}%
    \errmessage{\noexpand\maxfiles limit is too strict.}%
    \@maxfiles4
  \fi
}
```

Since batchfiles are now \inputed there should be no default limit here. I'll just use some abstract large number.

```
\maxfiles{1972} % year of my birth (MW)
```

\maxoutfiles

Maybe there's a restriction on just output streams. (Well, there is: I know, because TeX only allows 16.) I may as well allow the configuration to set this up.

Again, I need a counter.

```
\newcount\@maxoutfiles
```

And now the macro. I need at least one output stream which I think is reasonable.

```
\def\maxoutfiles#1{%
  \@maxoutfiles=#1\relax
  \ifnum\@maxoutfiles<1
    \@maxoutfiles1
    \errhelp{I'm not a magician.  I need at least one output^^J%
             stream to be able to do anything useful at all.^^J%
```

```
            Please be reasonable.}%
    \errmessage{\noexpand\maxoutfiles limit is insane}%
  \fi
}
```

The default limit is 16, because that's what TEX says.

```
\maxoutfiles{16}
```

`\checkfilelimit` This checks the file limit when a new batch file is started. If there's fewer than two files left here, we're not going to be able to strip any files. The file limit counter is local to the group which is set up around `\batchinput`, so that's all pretty cool.

```
\def\checkfilelimit{%
  \advance\@maxfiles\m@ne
  \ifnum\@maxfiles<2 %
    \errhelp{There aren't enough streams left to do any unpacking.^^J%
             I can't do anything about this, so complain at the^^J%
             person who made such a complicated installation.}%
    \errmessage{Too few streams left.}%
    \end
  \fi
}
```

8.10 Interaction with the user

`\strip@meaning` Throw away the first part of `\meaning` output.

```
\def\strip@meaning#1>{}
```

`\processbatchFile` When DocStrip is run it always tries to use a batch file.

For this purpose it calls the macro `\processbatchFile`.

The first thing is to check if there are any input streams left.

```
\def\processbatchFile{%
  \checkfilelimit
  \let\next\relax
```

Now we try to open the batch file for reading.

```
  \openin\inputcheck \batchfile\relax
  \ifeof\inputcheck
```

If we didn't succeed in opening the file, we assume that it does not exist. If we tried the default filename, we silently continue; the DocStrip program will switch to interactive mode in this case.

```
    \ifDefault
    \else
```

If we failed to open the user-supplied file, something is wrong and we warn him about it. This will also result in a switch to interactive mode.

```
      \errhelp
        {A batchfile specified in \batchinput could not be found.}%
      \errmessage{^^J%
           ************************************************^^J%
           * Could not find your \string\batchfile=\batchfile.^^J%
           ************************************************}%
    \fi
  \else
```

When we were successful in opening a file, we again have to check whether it was the default file. In that case we tell the user we found that file and ask him if he wants to use it.

```
    \ifDefault
      \Msg{************************************************^^J%
           * Batchfile \DefaultbatchFile\space found Use it? (y/n)?}%
      \Ask\answer{%
            ************************************************}%
    \else
```

When it was the user-supplied file we can safely assume he wants to use it so we set `\answer` to `y`.

```
      \let\answer\y
    \fi
```

If the macro `\answer` contains a `y` we can read in the batchfile. We do it in an indirect way—after completing `\if`s.

```
    \ifx\answer\y
      \closein\inputcheck
      \def\next{\@@input\batchfile\relax}%
    \fi
  \fi
  \next}
```

`\ReportTotals` The macro `\ReportTotals` can be used to report total statistics for all files processed. This code is only included in the program if the option `stats` is included.

```
⟨*stats⟩
\def\ReportTotals{%
  \ifnum\NumberOfFiles>\@ne
    \Msg{Overall statistics:^^J%
         Files \space processed: \the\NumberOfFiles^^J%
         Lines \space processed: \the\TotalprocessedLines^^J%
         Comments removed: \the\TotalcommentsRemoved^^J%
         Comments \space passed: \the\TotalcommentsPassed^^J%
         Codelines passed: \the\TotalcodeLinesPassed}%
  \fi}
⟨/stats⟩
```

`\SetFileNames` The macro `\SetFileNames` is used when the program runs in interactive mode and the user was asked to supply extensions and a list of filenames.

```
\def\SetFileNames{%
    \edef\sourceFileName{\MainFileName.\infileext}%
    \edef\destFileName{\MainFileName.\outfileext}}
```

`\CheckFileNames` In interactive mode, the user gets asked for the extensions for the input and output files. Also the name or names of the input files (without extension) is asked for. Then the names of the input and output files are constructed from this information by `\SetFileNames`. This assumes that the name of the input file is the same as the name of the output file. But we should not write to the same file we're reading from so the extensions should differ.

The macro `\CheckFileNames` makes sure that the output goes to a different file to the one where the input comes from.

```
\def\CheckFileNames{%
    \ifx\sourceFileName\destFileName
```

If input and output files are the same we signal an error and stop processing.

```
      \Msg{^^J%
     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J%
     ! It is not possible to read from and write to the same file !^^J%
     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J}%
      \Continuefalse
    \else
```

If they are not the same we check if the input file exists by trying to open it for reading.

```
      \Continuetrue
      \immediate\openin\inFile \sourceFileName\relax
      \ifeof\inFile
```

If an end of file was found, the file couldn't be opened, so we signal an error and stop processing.

```
        \Msg{^^J%
```

```
              !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J%
              ! Your input file `\sourceFileName' was not found !^^J%
              !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J}%
        \Continuefalse
      \else
```

The last check we have to make is if the output file already exists. Therefore we try to open it for reading. As a precaution we first close the input stream.

```
        \immediate\closein\inFile
        \immediate\openin\inFile\destdir \destFileName\relax
        \ifeof\inFile
```

If this fails, it didn't exist and all is well.

```
          \Continuetrue
        \else
```

If opening of the output file for reading succeeded we have to ask the user if he wants to overwrite it. We assume he doesn't want to overwrite it, so the switch `\ifContinue` is initially set to ⟨*false*⟩. Only if he answers the question positively with '`y`' or '`yes`' we set the switch back to ⟨*true*⟩.

```
          \Continuefalse
          \Ask\answer{File \destdir\destFileName\space already
                      exists
                      \ifx\empty\destdir somewhere \fi
                      on the system.^^J%
                      Overwrite it%
                      \ifx\empty\destdir\space if necessary\fi
                      ? [y/n]}%
          \ifx\y  \answer \Continuetrue \else
          \ifx\yes\answer \Continuetrue \else
          \fi\fi
        \fi
```

All checks have been performed now, so we can close any file that was opened just for this purpose.

```
      \fi
    \fi
    \closein\inFile}
```

`\interactive` The macro `\interactive` implements the interactive mode of the DocStrip program. The macro is implemented using the ⟨*while*⟩ construction. While the switch `\ifMoreFiles` remains true, we continue processing.

```
\def\interactive{%
  \whileswitch\ifMoreFiles\fi%
```

To keep macro redefinitions local we start a group and ask the user some questions about what he wants us to do.

```
   {\begingroup
      \AskQuestions
```

The names of the files that have to be processed are stored as a comma-separated list in the macro `\filelist` by `\AskQuestions`. We use a ⟨*for*⟩ loop to process the files one by one.

```
      \forlist\MainFileName:=\filelist
      \do
```

First the names of the input and output files are constructed and a check is made if all filename information is correct.

```
        \SetFileNames
        \CheckFileNames
        \ifContinue
```

If everything was well, produce output file.

```
        \generateFile{\destFileName}{f}%
                     {\from{\sourceFileName}{\Options}}
        \fi%
```

This process is repeated until `\filelist` is exhausted.

```
    \od
  \endgroup
```

Maybe the user wants more files to be processed, possibly with another set of options, so we give him the opportunity.

```
  \Ask\answer{More files to process (y/n)?}%
  \ifx\y  \answer\MoreFilestrue \else
  \ifx\yes\answer\MoreFilestrue \else
```

If he didn't want to process any more files, the switch `\ifMoreFiles` is set to ⟨*false*⟩ in order to interrupt the ⟨*while*⟩ loop.

```
                  \MoreFilesfalse\fi\fi
}}
```

`\AskQuestions` The macro `\AskQuestions` is called by `\interactive` to get some information from the user concerning the files that need to be processed.

```
\def\AskQuestions{%
  \Msg{^^J%
       ****************************************************}%
```

We want to know the extension of the input files,

```
  \Ask\infileext{%
       * First type the extension of your input file(s): \space  *}%
  \Msg{****************************************************^^J^^J%
       ****************************************************}%
```

the extension of the output files,

```
  \Ask\outfileext{%
       * Now type the extension of your output file(s) \space: *}%
  \Msg{****************************************************^^J^^J%
       ****************************************************}%
```

if options are to be included and

```
  \Ask\Options{%
     * Now type the name(s) of option(s) to include \space\space: *}%
  \Msg{****************************************************^^J^^J%
       ****************************************************^^J%
     * Finally give the list of input file(s) without \space\space*}%
```

the name of the input file or a list of names, separated by commas.

```
  \Ask\filelist{%
       * extension seperated by commas if necessary %
                                  \space\space\space\space: *}%
  \Msg{****************************************************^^J}}%
```

8.11 The main program

When TEX processes the DocStrip program it displays a message about the version of the program and its function on the terminal.

```
\Msg{Utility: `docstrip' \fileversion\space <\filedate>^^J%
     English documentation \space\space\space <\docdate>}%
\Msg{^^J%
     ************************************************************^^J%
     * This program converts documented macro-files into fast *^^J%
     * loadable files by stripping off (nearly) all comments! *^^J%
     ************************************************************^^J}%
```

`\WriteToDir` Macro `\WriteToDir` is either empty or holds the prefix necessary to read a file from the current directory. Under UNIX this is `./` but a lot of other systems addopted this concept. This macro is a default value for `\destdir`.

The definition of this macro is now delayed until `\@setwritedir` is called.

`\makepathname` This macro should define `\@pathname` to full path name made by combining current value of `\destdir` with its one argument being a file name. Its default value defined here is suitable for UNIX, MS-DOS and Macintosh, but for some systems it may be needed to redefine this in `docstrip.cfg` file. We provide such redefinition for VMS here.

Macro `\dirsep` holds directory separator specific for a system. Default value suitable for UNIX and DOS is slash. It comes in action when `\usedir` labels are used directly.

The definition of this macro is now delayed until `\@setwritedir` is called.

`\@setwritedir` The following tests try to automatically set `\WriteToDir`, `\dirname` and `\makepathname` in Unix, Mac, or VMS style. The tests are not run at the top level but held in this macro so that a configuration file has a chance to define `\WriteToDir` which allows the other two to be set automatically. The tests could more simply be run after the configuration file is read, but the configuration commands like `\BaseDirectory` need (at least at present) to have `\dirsep` correctly defined. It does not define any command that is already defined, so by defining these commands a configuration file can produce different effects for special needs. So this command is called by `BaseDirectory`, `\UseTDS`, `\DeclareDir` and finally at the top level after the cfg is run. It begins by redefining itself to be a no-op so it effectively is only called once.

```
\def\@setwritetodir{%
  \let\setwritetodir\relax
  \ifx\WriteToDir\@undefined
    \ifx\@currdir\@undefined
      \def\WriteToDir{}%
    \else
      \let\WriteToDir\@currdir
    \fi
  \fi
  \let\destdir\WriteToDir
```

VMS Style.

```
  \def\tmp{[]}%
  \ifx\tmp\WriteToDir
    \ifx\dirsep\@undefined
      \def\dirsep{.}%
    \fi
    \ifx\makepathname\@undefined
      \def\makepathname##1{%
        \edef\@pathname{\ifx\WriteToDir\destdir
          \WriteToDir\else[\destdir]\fi##1}}%
    \fi
  \fi
```

Unix and Mac styles.

```
  \ifx\dirsep\@undefined
    \def\dirsep{/}%
    \def\tmp{:}%
    \ifx\tmp\WriteToDir
      \def\dirsep{:}%
    \fi
  \fi
  \ifx\makepathname\@undefined
    \def\makepathname##1{%
      \edef\@pathname{\destdir\ifx\empty\destdir\else
           \ifx\WriteToDir\destdir\else\dirsep\fi\fi##1}}%
  \fi}
```

If the user has a `docstrip.cfg` file, use it now. This macro tries to read `docstrip.cfg` file. If this succeeds executes its first argument, otherwise the second.

```
\immediate\openin\inputcheck=docstrip.cfg\relax
\ifeof\inputcheck
  \Msg{%
     ********************************************************^^J%
     * No Configuration file found, using default settings. *^^J%
     ********************************************************^^J}%
\else
  \Msg{%
     ****************************************^^J%
     * Using Configuration file docstrip.cfg. *^^J%
     ****************************************^^J}%
  \closein\inputcheck
  \afterfi{\@@input docstrip.cfg\relax}
\fi
```

Now run `\@setwritedir` in case it has not already been run by a command in a configuration file.

```
\@setwritetodir
```

`\process@first@batchfile` Process the batch file, and then terminate cleanly. This may be set to `\relax` for 'new style' batch files that do not start with `\def\batchfile{`...

```
\def\process@first@batchfile{%
  \processbatchFile
  \ifnum\NumberOfFiles=\z@
    \interactive
  \fi
  \endbatchfile}
```

`\endbatchfile` User level command to end batch file processing. At the top level, returns totals and then stops TEX. At nested levels just does `\endinput`.

```
\def\endbatchfile{%
  \iftopbatchfile
⟨*stats⟩
    \ReportTotals
⟨/stats⟩
    \expandafter\end
  \else
    \endinput
  \fi}
```

Now we see whether to process a batch file.

`\@jobname` Jobname (catcode 12)

```
\edef\@jobname{\lowercase{\def\noexpand\@jobname{\jobname}}}%
\@jobname
```

`\@docstrip` docstrip (catcode 12)

```
\def\@docstrip{docstrip}%
\edef\@docstrip{\expandafter\strip@meaning\meaning\@docstrip}
```

First check whether the user has defined the control sequence `\batchfile`. If he did, it should contain the name of the file to process. If he didn't, try the current file, unless that is `docstrip.tex` in which case a default name is tried. Whether or not the default batch file is used is remembered by setting the switch `\ifDefault` to ⟨*true*⟩ or ⟨*false*⟩.

```
\Defaultfalse
\ifx\undefined\batchfile
```

\@jobname is lowercase jobname (catcode 12)
\@docstrip is docstrip (catcode 12)

```
\ifx\@jobname\@docstrip
```

Set the batchfile to the default

```
    \let\batchfile\DefaultbatchFile
    \Defaulttrue
```

Else don't process a new batchfile, just carry on with past the end of this file. In this case processing will move to the initial batchfile which *must* then be terminated by \endbatchfile or TEX will fall to the star prompt.

```
  \else
    \let\process@first@batchfile\relax
  \fi
\fi
\process@first@batchfile
```

⟨/program⟩

Change History