The doc and shortvrb Packages*

Frank Mittelbach†
Gutenberg Universität Mainz

September 6, 2008

Abstract

This package contains the definitions that are necessary to format the documentation of package files. The package was developed in Mainz in cooperation with the Royal Military College of Science. This is an update which documents various changes and new features in doc and integrates the features of newdoc.

Contents

*This file has version number v2.1d dated 2006/02/02.

†Further commentary added at Royal Military College of Science by B. Hamilton Kelly; English translation of parts of the original German commentary provided by Andrew Mills; fairly substantial additions, particularly from newdoc, and documentation of post-v1.5q features added at v1.7a by Dave Love (SERC Daresbury Lab). Extraction of shortvrb package added by Joachim Schrod (TU Darmstadt).

Preface to version 1.7

This version of `doc.dtx` documents changes which have occurred since the last published version [5] but which have been present in distributed versions of `doc.sty` for some time. It also integrates the (undocumented) features of the distributed `newdoc.sty`.

The following changes and additions have been made to the user interface since the published version [5]. See §2 for more details.

Driver mechanism `\DocInput` is now used in the driver file to input possibly multiple independent `doc` files and `doc` no longer has to be the last package. `\IndexListing` is replaced by `\IndexInput`;

Indexing is controlled by `\PageIndex` and `\CodelineIndex`, one of which must be specified to produce an index—there is no longer a `\makeindex` in the default `\DocstyleParms`;

The `macro` environment now takes as argument the macro name *with* the backslash;

Verbatim text Newlines are now forbidden inside `\verb` and commands `\MakeShortVerb` and `\DeleteShortVerb` are provided for verbatim shorthand;

`\par` can now be used in `\DoNotIndex`;

Checksum/character table support for ensuring the integrity of distributions is added;

`\printindex` becomes `\PrintIndex`;

`multicol.sty` is no longer necessary to use `doc` or print the documentation (although it is recommended);

'Docstrip' modules are recognised and formatted specially.

As well as adding some completely new stuff, the opportunity has been taken to add some commentary to the code formerly in `newdoc.sty` and that added after version 1.5k of `doc.sty`. Since (as noted in the sections concerned) this commentary wasn't written by Frank Mittelbach but the code was, it is probably *not* true in this case that "if the code and comments disagree both are probably wrong"!

Bugs

There are some known bugs in this version:

- The `\DoNotIndex` command doesn't work for some single character commands most noticeable `\%`.
- The 'General changes' glossary entry would come out after macro names with a leading `!` and possibly a leading `"`;
- If you have an old version of makeindex long `\changes` entries will come out strangely and you may find the section header amalgamated with the first changes entry. Try to get an up-to-date one (see p. 8);
- Because the accompanying makeindex style files support the inconsistent attribute specifications of older and newer versions makeindex always complains about three 'unknown specifier's when sorting the index and changes entries.
- If `\MakeShortVerb` and `\DeleteShortVerb` are used with single character arguments, e.g., `{|}` instead of `{\|}` chaos may happen.

(Some 'features' are documented below.)

Wish list

- Hooks to allow `\DescribeMacro` and `\DescribeEnv` to write out to a special file information about the package's 'exported' definitions which they describe. This could subsequently be included in the `docstripped` `.sty` file in a suitable form for use by smart editors in command completion, spelling checking etc., based on the packages used in a document.

This would need agreement on a 'suitable form'.

- Indexing of the modules used in `docstrip`'s `%<` directives. I'm not sure how to index directives containing module combinations;
- Writing out bibliographic information about the package;
- Allow turning off use of the special font for, say, the next guarded block.

---

1 Introduction

The TEX macros which are described here allow definitions and documentation to be held in one and the same file. This has the advantage that normally very complicated instructions are made simpler to understand by comments inside the definition. In addition to this, updates are easier and only one source file needs to be changed. On the other hand, because of this, the package files are considerably longer: thus TEX takes longer to load them. If this is a problem, there is an easy remedy: one needs only to run the `docstrip.tex` program that removes nearly all lines that begin with a percent sign.

The idea of integrated documentation was born with the development of the TEX program; it was crystallized in Pascal with the WEB system. The advantages of this method are plain to see (it's easy to make comparisons [2]). Since this development, systems similar to WEB have been developed for other programming languages. But for one of the most complicated programming languages (TEX) the documentation has however been neglected. The TEX world seems to be divided between:—

- a couple of "wizards", who produce many lines of completely unreadable code "off the cuff", and
- many users who are amazed that it works just how they want it to do. Or rather, who despair that certain macros refuse to do what is expected of them.

I do not think that the WEB system is *the* reference work; on the contrary, it is a prototype which suffices for the development of programs within the TEX world. It is sufficient, but not totally sufficient.[1] As a result of WEB, new programming perspectives have been demonstrated; unfortunately, though, they haven't been developed further for other programming languages.

The method of documentation of TEX macros which I have introduced here should also only be taken as a first sketch. It is designed explicitly to run under LATEX alone. Not because I was of the opinion that this was the best starting point, but because from this starting point it was the quickest to develop.[2] As a result of this design decision, I had to move away from the concept of modularization; this was certainly a step backward.

I would be happy if this article could spark off discussion over TEX documentation. I can only advise anyone who thinks that they can cope without documentation to "Stop Time" until he or she completely understands the $\mathcal{AMS}$-TEX source code.

1.1 Using the `doc` package

Just like any other package, invoke it by requesting it with a `\usepackage` command in the preamble. Doc's use of `\reversemarginpars` may make it incompatible with some classes.

[1] I know that this will be seen differently by a few people, but this product should not be seen as the finished product, at least as far as applications concerning TEX are concerned. The long-standing debate over 'multiple change files' shows this well.

[2] This argument is a bad one, however, it is all too often trotted out.

2 The User Interface

2.1 The driver file

If one is going to document a set of macros with the `doc` package one has to prepare a special driver file which produces the formatted document. This driver file has the following characteristics:

`\documentclass[`⟨*options*⟩`]{`⟨*document-class*⟩`}`

`\usepackage{doc}`

⟨*preamble*⟩

`\begin{document}`

⟨*special input commands*⟩

`\end{document}`

The ⟨*document-class*⟩ might be any document class, I normally use `article`.

In the ⟨*preamble*⟩ one should place declarations which manipulate the behavior of the `doc` package like `\DisableCrossrefs` or `\OnlyDescription`.

`\DocInput` `\IndexInput`

Finally the ⟨*special input commands*⟩ part should contain one or more `\DocInput`⟨*file name*⟩ and/or `\IndexInput`⟨*file name*⟩ commands. The `\DocInput` command is used for files prepared for the `doc` package whereas `\IndexInput` can be used for all kinds of macro files. See page 10 for more details of `\IndexInput`. Multiple `\DocInput`s can be used with a number of included files which are each self-contained self-documenting packages—for instance, each containing `\maketitle`.

As an example, the driver file for the `doc` package itself is the following text surrounded by `%<*driver>` and `%</driver>`. To produce the documentation you can simply run the `.dtx` file through LaTeX in which case this code will be executed (loading the document class `ltxdoc`, etc.) or you can extract this into a separate file by using the `docstrip` program. The line numbers below are added by `doc`'s formatting. Note that the class ltxdoc has the `doc` package preloaded.

```
1 ⟨*driver⟩
2 \documentclass{ltxdoc}
3 \EnableCrossrefs
4  %\DisableCrossrefs % Say \DisableCrossrefs if index is ready
5 \CodelineIndex
6 \RecordChanges                  % Gather update information
7  %\OnlyDescription  % comment out for implementation details
8  %\OldMakeindex     % use if your MakeIndex is pre-v2.9
9 \setlength\hfuzz{15pt}  % dont make so many
10 \hbadness=7000          % over and under full box warnings
11 \begin{document}
12    \DocInput{doc.dtx}
13 \end{document}
14 ⟨/driver⟩
```

2.2 General conventions

A TeX file prepared to be used with the 'doc' package consists of 'documentation parts' intermixed with 'definition parts'.

Every line of a 'documentation part' starts with a percent sign (`%`) in column one. It may contain arbitrary TeX or LaTeX commands except that the character '`%`' cannot be used as a comment character. To allow user comments, the `^^A` character is defined as a comment character later on. Such 'metacomments' may be also be included simply by surrounding them with `\iffalse` ... `\fi`.

All other parts of the file are called 'definition parts'. They contain fractions of the macros described in the 'documentation parts'.

If the file is used to define new macros (e.g. as a package file in the `\usepackage` macro), the 'documentation parts' are bypassed at high speed and the macro definitions are pasted together, even if they are split into several 'definition parts'.

macrocode

On the other hand, if the documentation of these macros is to be produced, the 'definition parts' should be typeset verbatim. To achieve this, these parts are surrounded by the macrocode environment. More exactly: before a 'definition part' there should be a line containing

```
%␣␣␣␣\begin{macrocode}
```

and after this part a line

```
%␣␣␣␣\end{macrocode}
```

There must be *exactly* four spaces between the % and \end{macrocode} — TEX is looking for this string and not for the macro while processing a 'definition part'.

Inside a 'definition part' all TEX commands are allowed; even the percent sign could be used to suppress unwanted spaces etc.

macrocode*

Instead of the macrocode environment one can also use the macrocode∗ environment which produces the same results except that spaces are printed as ␣ characters.

2.3 Describing the usage of new macros

\DescribeMacro

When you describe a new macro you may use \DescribeMacro to indicate that at this point the usage of a specific macro is explained. It takes one argument which will be printed in the margin and also produces a special index entry. For example, I used \DescribeMacro{\DescribeMacro} to make clear that this is the point where the usage of \DescribeMacro is explained.

\DescribeEnv

An analogous macro \DescribeEnv should be used to indicate that a LATEX environment is explained. It will produce a somewhat different index entry. Below I used \DescribeEnv{verbatim}.

verbatim
verbatim*
\verb

It is often a good idea to include examples of the usage of new macros in the text. Because of the % sign in the first column of every row, the verbatim environment is slightly altered to suppress those characters.[3] The verbatim∗ environment is changed in the same way. The \verb command is re-implemented to give an error report if a newline appears in its argument. The verbatim and verbatim∗ environments set text in the style defined by \MacroFont (§2.4).

2.4 Describing the definition of new macros

macro

To describe the definition of a new macro we use the macro environment. It has one argument: the name of the new macro.[4] This argument is also used to print the name in the margin and to produce an index entry. Actually the index entries for usage and definition are different to allow an easy reference. This environment might be nested. In this case the labels in the margin are placed under each other. There should be some text—even if it's just an empty \mbox{}—in this environment before \begin{macrocode} or the marginal label won't print in the right place.

\MacrocodeTopsep
\MacroTopsep
\MacroIndent
\MacroFont

There also exist four style parameters: \MacrocodeTopsep and \MacroTopsep are used to control the vertical spacing above and below the macrocode and the macro environment, \MacroIndent is used to indent the lines of code and \MacroFont holds the font and a possible size change command for the code lines, the verbatim[*] environment and the macro names printed in the margin. If you want to change their default values in a class file (like ltugboat.cls) use the \DocstyleParms command described below. Starting with release 2.0a it can now be changed directly as long as the redefinition happens before the \begin{document}.

[3] These macros were written by Rainer Schöpf [8]. He also provided a new verbatim environment which can be used inside of other macros.

[4] This is a change to the style design I described in *TUGboat* 10#1 (Jan. 89). We finally decided that it would be better to use the macro name *with* the backslash as an argument.

2.5 Formatting the margins

\PrintDescribeMacro
\PrintDescribeEnv
\PrintMacroName
\PrintEnvName

As mentioned earlier, some macros and the macro environment print their arguments in the margin. This is actually done by four macros which are user definable.[5] They are named \PrintDescribeMacro, \PrintDescribeEnv, \PrintMacroName (called by the macro environment) and \PrintEnvName (called by the environment environment).

2.6 Using a special escape character

\SpecialEscapechar

If one defines complicated macros it is sometimes necessary to introduce a new escape character because the '\' has got a special \catcode. In this case one can use \SpecialEscapechar to indicate which character is actually used to play the rôle of the '\'. A scheme like this is needed because the macrocode environment and its counterpart macrocode∗ produce an index entry for every occurrence of a macro name. They would be very confused if you didn't tell them that you'd changed \catcodes. The argument to \SpecialEscapechar is a single-letter control sequence, that is, one has to use \| for example to denote that '|' is used as an escape character. \SpecialEscapechar only changes the behavior of the next macrocode or macrocode∗ environment.

The actual index entries created will all be printed with \ rather than |, but this probably reflects their usage, if not their definition, and anyway must be preferable to not having any entry at all. The entries *could* be formatted appropriately, but the effort is hardly worth it, and the resulting index might be more confusing (it would certainly be longer!).

2.7 Cross-referencing all macros used

\DisableCrossrefs
\EnableCrossrefs

As already mentioned, every new macro name used within a macrocode or macrocode∗ environment will produce an index entry. In this way one can easily find out where a specific macro is used. Since TEX is considerably slower when it has to produce such a bulk of index entries one can turn off this feature by using \DisableCrossrefs in the driver file. To turn it on again just use \EnableCrossrefs.[6]

\DoNotIndex

But also finer control is provided. The \DoNotIndex macro takes a list of macro names separated by commas. Those names won't show up in the index. You might use several \DoNotIndex commands: their lists will be concatenated. In this article I used \DoNotIndex for all macros which are already defined in LATEX.

All three above declarations are local to the current group.

Production (or not) of the index (via the \makeindex commend) is controlled by using or omitting the following declarations in the driver file preamble; if neither is used, no index is produced. Using \PageIndex makes all index entries refer to their page number; with \CodelineIndex, index entries produced by \DescribeMacro and \DescribeEnv refer to page number but those produced by the macro environment refer to the code lines, which will be numbered automatically.[7] The style of this numbering can be controlled by defining the macro \theCodelineNo. Its default definition is to use scriptsize arabic numerals; a user-supplied definition won't be overwritten.

\PageIndex
\CodelineIndex
\theCodelineNo

\CodelineNumbered

When you don't wish to get an index but want your code lines numbered use

[5] You may place the changed definitions in a separate package file or at the beginning of the documentation file. For example, if you don't like any names in the margin but want a fine index you can simply \let these macros equal \@gobble. The doc package won't redefine any existing definitions of these macros.

[6] Actually, \EnableCrossrefs changes things more drastically; any following \DisableCrossrefs which might be present in the source will be ignored.

[7] The line number is actually that of the first line of the first macrocode environment in the macro environment.

\CodelineNumbered instead of \CodelineIndex. This prevents the generation of an unnecessary .idx file.

2.8 Producing the actual index entries

Several of the aforementioned macros will produce some sort of index entries. These entries have to be sorted by an external program—the current implementation assumes that the makeindex program by Chen [4] is used.

But this isn't built in: one has only to redefine some of the following macros to be able to use any other index program. All macros which are installation dependent are defined in such a way that they won't overwrite a previous definition. Therefore it is safe to put the changed versions in a package file which might be read in before the doc package.

To allow the user to change the specific characters recognized by his or her index program all characters which have special meaning in the makeindex program are given symbolic names.[8] However, all characters used should be of \catcode other than 'letter' (11).

\actualchar
\quotechar
\encapchar
\levelchar

The \actualchar is used to separate the 'key' and the actual index entry. The \quotechar is used before a special index program character to suppress its special meaning. The \encapchar separates the indexing information from a letter string which makeindex uses as a TEX command to format the page number associated with a special entry. It is used in this package to apply the \main and the \usage commands. Additionally \levelchar is used to separate 'item', 'subitem' and 'subsubitem' entries.

It is a good idea to stick to these symbolic names even if you know which index program is used. In this way your files will be portable.

\SpecialMainIndex
\SpecialMainEnvIndex
\SpecialIndex
\SpecialUsageIndex
\SpecialEnvIndex
\SortIndex

To produce a main index entry for a macro the \SpecialMainIndex macro[9] may be used. It is called 'special' because it has to print its argument verbatim. A similar macro, called \SpecialMainEnvIndex is used for indexing the main definition point of an environment.[10] If you want a normal index entry for a macro name \SpecialIndex might be used.[11] To index the usage of a macro or an environment \SpecialUsageIndex and \SpecialEnvIndex may be used. Additionally a \SortIndex command is provided. It takes two arguments—the sort key and the actual index entry.

All these macros are normally used by other macros; you will need them only in an emergency.

\verbatimchar

But there is one characteristic worth mentioning: all macro names in the index are typeset with the \verb* command. Therefore one special character is needed to act as a delimiter for this command. To allow a change in this respect, again this character is referenced indirectly, by the macro \verbatimchar. It expands by default to + but if your code lines contain macros with '+' characters in their names (e.g. when you use \+) you will end up with an index entry containing \verb+\++ which will be typeset as '\+' and not as '\+'. In this case you should redefine \verbatimchar globally or locally to overcome this problem.

*

We also provide a * macro. This is intended to be used for index entries like

> index entries
> Special macros for ~

Such an entry might be produced with the line:

```
\index{index entries\levelchar Special macros for \*}
```

\OldMakeindex

Versions of makeindex prior to 2.9 had some bugs affecting doc. One of these, pertaining to the % character doesn't have a work-around appropriate for versions

[8] I don't know if there exists a program which needs more command characters, but I hope not.

[9] This macro is called by the macro environment.

[10] This macro is called by the environment environment.

[11] This macro is called within the macrocode environment when encountering a macro name.

with and without the bug. If you have an old version, invoke `\OldMakeindex` in a package file or the driver file to prevent problems with index entries such as `\%`, although you'll probably normally want to turn off indexing of `\%` anyway. Try to get an up-to-date makeindex from one of the TEX repositories.

2.9 Setting the index entries

After the first formatting pass through the `.dtx` file you need to sort the index entries written to the `.idx` file using makeindex or your favourite alternative. You need a suitable style file for makeindex (specified by the `-s` switch). A suitable one is supplied with `doc`, called `gind.ist`.

`\PrintIndex` To read in and print the sorted index, just put the `\PrintIndex` command as the last (commented-out, and thus executed during the documentation pass through the file) command in your package file. Precede it by any bibliography commands necessary for your citations. Alternatively, it may be more convenient to put all such calls amongst the arguments of the `\StopEventually` macro, in which case a `\Finale` command should appear at the end of your file.

`theindex` Contrary to standard LATEX, the index is typeset in three columns by default. This is controlled by the LATEX counter 'IndexColumns' and can therefore be changed with a `\setcounter` declaration. Additionally one doesn't want to start a new page unnecessarily. Therefore the theindex environment is redefined.

`\IndexMin` When the theindex environment starts it will measure how much space is left on the current page. If this is more than `\IndexMin` then the index will start on this page. Otherwise `\newpage` is called.

Then a short introduction about the meaning of several index entries is typeset (still in onecolumn mode). Afterwards the actual index entries follow in multi-

`\IndexPrologue` column mode. You can change this prologue with the help of the `\IndexPrologue` macro. Actually the section heading is also produced in this way, so you'd better write something like:

```
\IndexPrologue{\section*{Index} The index entries underlined ...}
```

When the theindex environment is finished the last page will be reformatted to produce balanced columns. This improves the layout and allows the next article to

`\IndexParms` start on the same page. Formatting of the index columns (values for `\columnssep` etc.) is controlled by the `\IndexParms` macro. It assigns the following values:

`\parindent`	= 0.0pt	`\columnsep`	= 15.0pt
`\parskip`	= 0.0pt plus 1.0pt	`\rightskip`	= 15.0pt
`\mathsurround`	= 0.0pt	`\parfillskip`	= −15.0pt

`\@idxitem` Additionally it defines `\@idxitem` (which will be used when an `\item` command is encountered) and selects `\small` size. If you want to change any of these values you have to define them all.

`\main` The page numbers for main index entries are encapsulated by the `\main` macro

`\usage` (underlining its argument) and the numbers denoting the description are encapsulated by the `\usage` macro (which produces *italics*). As usual these commands are user definable.

2.10 Changing the default values of style parameters

`\DocstyleParms` If you want to overwrite some default settings made by the `doc` package, you can either put your declarations in the driver file (that is after `doc.sty` is read in) or use a separate package file for doing this work. In the latter case you can define the macro `\DocstyleParms` to contain all assignments. This indirect approach is necessary if your package file might be read before the `doc.sty`, when some of the registers are not allocated. Its default definition is null.

The doc package currently assigns values to the following registers:

\IndexMin	= 80.0pt	\MacroTopsep	= 7.0pt plus 2.0pt minus 2.0pt
\marginparwidth	= 126.0pt	\MacroIndent	= 14.65285pt
\marginparpush	= 0.0pt	\MacrocodeTopsep	= 3.0pt plus 1.2pt minus 1.0pt
\tolerance	= 1000		

2.11 Short input of verbatim text pieces

\MakeShortVerb
\MakeShortVerb*
\DeleteShortVerb

It is awkward to have to type, say, `\verb|...|` continually when quoting verbatim bits (like macro names) in the text, so an abbreviation mechanism is provided. Pick a character ⟨*c*⟩—one which normally has catcode 'other' unless you have very good reason not to—which you don't envisage using in the text, or not using often. (I like `"`, but you may prefer `|` if you have `"` active to do umlauts, for instance.) Then if you say `\MakeShortVerb{\`⟨*c*⟩`}` you can subsequently use ⟨*c*⟩⟨*text*⟩⟨*c*⟩ as the equivalent of `\verb`⟨*c*⟩⟨*text*⟩⟨*c*⟩; analogously, the `*`-form `\MakeShortVerb*{\`⟨*c*⟩`}` gives you the equivalent of `\verb*`⟨*c*⟩⟨*text*⟩⟨*c*⟩. Use `\DeleteShortVerb{\`⟨*c*⟩`}` if you subsequently want ⟨*c*⟩ to revert to its previous meaning—you can always turn it on again after the unusual section. The 'short verb' commands make global changes. The abbreviated `\verb` may not appear in the argument of another command just like `\verb`. However the 'short verb' character may be used freely in the verbatim and macrocode environments without ill effect. `\DeleteShortVerb` is silently ignored if its argument does not currently represent a short verb character. Both commands type a message to tell you the meaning of the character is being changed.

Please remember that the command `\verb` cannot be used in arguments of other commands. Therefore abbreviation characters for `\verb` cannot be used there either.

This feature is also available as a sole package, `shortvrb`.

2.12 Additional bells and whistles

We provide macros for logos such as Web, $\mathcal{AMS}$-TEX, BibTEX, SliTEX and Plain TEX. Just type `\Web`, `\AmSTeX`, `\BibTeX`, `\SliTeX` or `\PlainTeX`, respectively. LaTeX and TEX are already defined in `latex.tex`.

\meta

Another useful macro is `\meta` which has one argument and produces something like ⟨*dimen parameter*⟩.

\OnlyDescription
\StopEventually

You can use the `\OnlyDescription` declaration in the driver file to suppress the last part of your document (which presumably exhibits the code). To make this work you have to place the command `\StopEventually` at a suitable point in your file. This macro has one argument in which you put all information you want to see printed if your document ends at this point (for example a bibliography which is normally printed at the very end). When the `\OnlyDescription` declaration is missing the `\StopEventually` macro saves its argument in a macro called `\Finale` which can afterwards be used to get things back (usually at the very end). Such a scheme makes changes in two places unnecessary.

\Finale

Thus you can use this feature to produce a local guide for the TEX users which describes only the usage of macros (most of them won't be interested in your definitions anyway). For the same reason the `\maketitle` command is slightly changed to allow multiple titles in one document. So you can make one driver file reading in several articles at once. To avoid an unwanted pagestyle on the title page the `\maketitle` command issues a `\thispagestyle{titlepage}` declaration which produces a plain page if the titlepage page style is undefined. This allows class files like ltugboat.cls to define their own page styles for title pages.

\maketitle
\ps@titlepage

\AlsoImplementation

Typesetting the whole document is the default. However, this default can also be explicitly selected using the declaration `\AlsoImplementation`. This overwrites any previous `\OnlyDescription` declaration. The LaTeX 2_ε distribution, for example, is documented using the `ltxdoc` class which allows for a configuration file `ltxdoc.cfg`. In such a file one could then add the statement

```
\AtBeginDocument{\AlsoImplementation}
```

to make sure that all documents will show the code part.

\IndexInput Last but not least I defined an `\IndexInput` macro which takes a file name as an argument and produces a verbatim listing of the file, indexing every command as it goes along. This might be handy, if you want to learn something about macros without enough documentation. I used this feature to cross-reference `latex.tex` getting a verbatim copy with about 15 pages index.[12]

\changes To maintain a change history within the file, the `\changes` command may be placed amongst the description part of the changed code. It takes three arguments, thus:

`\changes{`⟨*version*⟩`}{`⟨*date*⟩`}{`⟨*text*⟩`}`

The changes may be used to produce an auxiliary file (LaTeX's `\glossary` mechanism is used for this) which may be printed after suitable formatting. The `\changes` macro generates the printed entry in such a change history; because old versions[13] of the makeindex program limit such fields to 64 characters, care should be taken not to exceed this limit when describing the change. The actual entry consists of the ⟨*version*⟩, the `\actualchar`, the current macro name, a colon, the `\levelchar`, and, finally, the ⟨*text*⟩. The result is a glossaryentry for the ⟨*version*⟩, with the name of the current macro as subitem. Outside the `macro` environment, the text `\generalname` is used instead of the macro name. When referring to macros in change descriptions it is conventional to use `\cs{`⟨*macroname*⟩`}` rather than attempting to format it properly and using up valuable characters in the entry with old makeindex versions.

\RecordChanges \PrintChanges To cause the change information to be written out, include `\RecordChanges` in the driver file. To read in and print the sorted change history (in two columns), just put the `\PrintChanges` command as the last (commented-out, and thus executed during the documentation pass through the file) command in your package file. Alternatively, this command may form one of the arguments of the `\StopEventually` command, although a change history is probably *not* required if only the description is being printed. The command assumes that makeindex or some other program has processed the `.glo` file to generate a sorted `.gls` file. You need a special makeindex style file; a suitable one is supplied with `doc`, called `gglo.ist`. The `\GlossaryMin`, `\GlossaryPrologue` and `\GlossaryParms` macros are analagous to the `\Index...` versions. (The LaTeX 'glossary' mechanism is used for the change entries.)

\GlossaryMin \GlossaryPrologue \GlossaryParms

\CharacterTable \CheckSum To overcome some of the problems of sending files over the networks we developed two macros which should detect corrupted files. If one places the lines

```
%%\CharacterTable
%% {Upper-case    \A\B\C\D\E\F\G\H\I\J\K\L\M\N\O\P\Q\R\S\T\U\V\W\X\Y\Z
%%  Lower-case    \a\b\c\d\e\f\g\h\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\y\z
%%  Digits        \0\1\2\3\4\5\6\7\8\9
%%  Exclamation   \!     Double quote  \"     Hash (number) \#
%%  Dollar        \$     Percent       \%     Ampersand     \&
%%  Acute accent  \'     Left paren    \(     Right paren   \)
%%  Asterisk      \*     Plus          \+     Comma         \,
%%  Minus         \-     Point         \.     Solidus       \/
%%  Colon         \:     Semicolon     \;     Less than     \<
%%  Equals        \=     Greater than  \>     Question mark \?
%%  Commercial at \@     Left bracket  \[     Backslash     \\
%%  Right bracket \]     Circumflex    \^     Underscore    \_
%%  Grave accent  \`     Left brace    \{     Vertical bar  \|
%%  Right brace   \}     Tilde         \~}
%%
```

[12] It took quite a long time and the resulting `.idx` file was longer than the `.dvi` file. Actually too long to be handled by the makeindex program directly (on our MicroVAX) but the final result was worth the trouble.

[13] Before 2.6.

at the beginning of the file then character translation failures will be detected, provided of course, that the used `doc` package has a correct default table. The percent signs[14] at the beginning of the lines should be typed in, since only the `doc` package should look at this command.

Another problem of mailing files is possible truncation. To detect these sort of errors we provide a `\CheckSum` macro. The check-sum of a file is simply the number of backslashes in the code, i.e. all lines between the macrocode environments. But don't be afraid: you don't have count the code-lines yourself; this is done by the `doc` package for you. You simply have to use the `\StopEventually` (which starts looking for backslashes) and the `\Finale` command. The latter will inform you either that your file has no check-sum (telling you the right number) or that your number is incorrect (this time telling you both the correct and the incorrect one). Then you go to the top of your file inserting the line

```
%%  \CheckSum{⟨number⟩}
```

and that's all. If you precede it only with one percent then the line will not show up in `docstrip` versions of the file. You should do so whenever you are using conditional code (see `docstrip` documentation) since then the check-sum will not reflect the number of backslashes in the stripped of versions.

`\bslash` From time to time, it is necessary to print a `\` without being able to use the `\verb` command because the `\catcode`s of the symbols are already firmly established. In this instance we can use the command `\bslash` presupposing, of course, that the actual font in use at this point contains a 'backslash' as a symbol. Note that this definition of `\bslash` is expandable; it inserts a $\backslash_{12}$. This means that you have to `\protect` it if it is used in 'moving arguments'.

`\MakePrivateLetters` If your macros `\catcode` anything other than `@` to 'letter', you should redefine `\MakePrivateLetters` so that it also makes the relevant characters 'letters' for the benefit of the indexing. The default definition is just `\makeatletter`.

`\DontCheckModules` `\CheckModules` `\Module` `\AltMacroFont` The 'module' directives of the docstrip system [6] are normally recognised and invoke special formatting. This can be turned on and off in the `.dtx` file or the driver file using `\CheckModules` and `\DontCheckModules`. If checking for module directives is on (the default) then code in the scope of the directives is set as determined by the hook `\AltMacroFont`, which gives *small italic typewriter* by default in the New Font Selection Scheme but just ordinary `small typewriter` in the old one, where a font such as italic typewriter can't be used portably (plug for NFSS); you will need to override this if you don't have the italic typewriter font available. Code is in such a scope if it's on a line beginning with `%<` or is between lines starting with `%<*`⟨*name list*⟩`>` and `%</`⟨*name list*⟩`>`. The directive is formatted by the macro `\Module` whose single argument is the text of the directive between, but not including, the angle brackets; this macro may be re-defined in the driver or package file and by default produces results like ⟨+foo | bar⟩ with no following space.

`StandardModuleDepth` Sometimes (as in this file) the whole code is surrounded by modules to produce several files from a single source. In this case it is clearly not appropriate to format all code lines in a special `\AltMacroFont`. For this reason a counter `StandardModuleDepth` is provided which defines the level of module nesting which is still supposed to be formatted in `\MacroFont` rather then `\AltMacroFont`. The default setting is `0`, for this documentation it was set to

```
\setcounter{StandardModuleDepth}{1}
```

at the beginning of the file.

2.13 Basic usage summary

To sum up, the basic structure of a `.dtx` file without any refinements is like this:

[14] There are two percent signs in each line. This has the effect that these lines are not removed by the `docstrip.tex` program.

```
% ⟨waffle⟩...
  ...
% \DescribeMacro{\fred}
% ⟨description of fred's use⟩
  ...
% \StopEventually{⟨finale code⟩}
  ...
% \begin{macro}{\fred}
% ⟨commentary on macro fred⟩
%␣␣␣␣\begin{macrocode}
⟨code for macro fred⟩
%␣␣␣␣\end{macrocode}
% \end{macro}
  ...
% \Finale \PrintIndex \PrintChanges
```

For examples of the use of most—if not all—of the features described above consult the `doc.dtx` source itself.

2.14 Acknowledgements

I would like to thank all folks at Mainz and at the Royal Military College of Science for their help in this project. Especially Brian and Rainer who pushed everything with their suggestions, bug fixes, etc.

A big thank you to David Love who brought the documentation up-to-date again, after I neglected this file for more than two years. This was most certainly a tough job as many features added to `doc.dtx` after its publication in *TUGboat* have been never properly described. Beside this splendid work he kindly provided additional code (like "docstrip" module formatting) which I think every doc.dtx user will be grateful for.

3 The Description of Macros

Most of the following code is destined for `doc.sty` after processing with `docstrip` to include the module **style** indicated here. (All code in this file not appropriate to `doc.sty` has to be included explicitly by docstrip so that this `.dtx` file can be used as directly as a package file rather than the stripped version.) The usual font change for the conditionally-included lines between the ⟨∗style⟩ and ⟨/style⟩ directives is suppressed since only the lines with an explicit directive are special in this file.

```
15 ⟨∗package⟩
```

Under LaTeX 2ε the test to avoid reading `doc` in twice is normally unnecessary. It was kept to only to stay compatible with LaTeX209 styles that `\input doc` directly.

```
16 \@ifundefined{macro@cnt}{}{\endinput}
```

`\fileversion` `\filedate` `\docdate`

As you can see I used macros like `\fileversion` to denote the version number and the date. They are defined at the very beginning of the package file (without a surrounding macrocode environment), so I don't have to search for this place here when I change the version number. You can see their actual outcome in a footnote to the title.

The first thing that we do next is to get ourselves a new comment sign. Because all sensible signs are already occupied, we will choose one that can only be entered indirectly:

```
17 \catcode‘\^^A=14
```

We repeat this statement at the beginning of the document in case the `inputenc` package is used disabling it again.

```
18 \AtBeginDocument{\catcode‘\^^A=14\relax}
```

3.1 Options supported by doc

Not options available at the moment

3.2 Macros surrounding the 'definition parts'

\macrocode

Parts of the macro definition will be surrounded by the environment macrocode. Put more precisely, they will be enclosed by a macro whose argument (the text to be set 'verbatim') is terminated by the string `%␣␣␣␣\end{macrocode}`. Carefully note the number of spaces. `\macrocode` is defined completely analogously to `\verbatim`, but because a few small changes were carried out, almost all internal macros have got new names. We start by calling the macro `\macro@code`, the macro which bears the brunt of most of the work, such as `\catcode` reassignments, etc.

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

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

```
    \frenchspacing \@vobeyspaces
```

Before closing, we need to call `\xmacro@code`. It is this macro that expects an argument which is terminated by the above string. This way it is possible to keep the `\catcode` changes local.

```
    \xmacro@code}
```

\macro@code

We will now begin with the macro that does the actual work:

```
\def\macro@code{%
```

In theory it should consist of a trivlist environment, but the empty space before and after the environment should not be too large.

```
    \topsep \MacrocodeTopsep
```

The next parameter we set is `\@beginparpenalty`, in order to prevent a page break before such an environment.

```
    \@beginparpenalty \predisplaypenalty
```

We then start a `\trivlist`, set `\parskip` back to zero and start an empty `\item`.

```
    \if@inlabel\leavevmode\fi
    \trivlist \parskip \z@ \item[]%
```

Additionally, everything should be set in `typewriter` font. Some people might prefer it somewhat differently; because of this the font choice is macro-driven.[15]

```
    \macro@font
```

Because `\item` sets various parameters, we have found it necessary to alter some of these retrospectively.

```
    \leftskip\@totalleftmargin \advance\leftskip\MacroIndent
    \rightskip\z@ \parindent\z@ \parfillskip\@flushglue
```

The next line consists of the LaTeX definition of `\par` used in `\verbatim` and should result in blank lines being shown as blank lines.

```
    \blank@linefalse \def\par{\ifblank@line
                              \leavevmode\fi
                              \blank@linetrue\@@par
                              \penalty\interlinepenalty}
```

What use is this definition of `\par`? We use the macro `\obeylines` of [3] which changes all `^^M` to `\par` so that each can control its own indentation. Next we must also ensure that all special signs are normalized; that is, they must be given `\catcode` 12.

[15] The font change has to be placed *after* the `\item`. Otherwise a change to `\baselineskip` will affect the paragraph above.

```
  \obeylines
  \let\do\do@noligs \verbatim@nolig@list
  \let\do\@makeother \dospecials
```

If indexing by code lines is switched on the line number is incremented and set appropriately. We also check whether the start of the next line indicates a `docstrip` module directive and process it appropriately if so using `\check@module`.

```
  \global\@newlistfalse
  \global\@minipagefalse
  \ifcodeline@index
    \everypar{\global\advance\c@CodelineNo\@ne
              \llap{\theCodelineNo\ \hskip\@totalleftmargin}%
              \check@module}%
  \else \everypar{\check@module}%
  \fi
```

We also initialize the cross-referencing feature by calling `\init@crossref`. This will start the scanning mechanism when encountering an escape character.

```
  \init@crossref}
```

`\ifblank@line` `\blank@linetrue` `\blank@linefalse`

`\ifblank@line` is the switch used in the definition above. In the original verbatim environment the `\if@tempswa` switch is used. This is dangerous because its value may change while processing lines in the macrocode environment.

```
\newif\ifblank@line
```

`\endmacrocode`

Because we have begun a trivlist environment in the macrocode environment, we must also end it. We must also act on the value of the `pm@module` flag (see below) and empty `\everypar`.

```
\def\endmacrocode{%
                  \ifpm@module \endgroup \pm@modulefalse \fi
                  \everypar{}%
                  \global\@inlabelfalse
                  \endtrivlist
```

Additionally `\close@crossref` is used to do anything needed to end the cross-referencing mechanism.

```
                  \close@crossref}
```

`\MacroFont`

Here is the default definition for the `\MacroFont` macro. With the new math font handling in NFSS2 it isn't any longer correct to suppress the math font setup since this is now handled differently. But to keep the font change fast we use only a single `\selectfont` (in `\small`) and do the rest by hand.

```
\@ifundefined{MacroFont}{%
  \if@compatibility
```

Despite the above statement we will call `\small` first if somebody is using a LaTeX2.09 document with doc. I wouldn't have bothered since doc-sources should be up-to-date but since the request came from someone called David Carlisle . . . :-)

```
    \def\MacroFont{\small
                   \usefont\encodingdefault
                           \ttdefault
                           \mddefault
                           \updefault
                   }%
  \else
    \def\MacroFont{\fontencoding\encodingdefault
                   \fontfamily\ttdefault
                   \fontseries\mddefault
                   \fontshape\updefault
                   \small}%
  \fi
  }{}
```

\AltMacroFont
\macro@font

Although most of the macro code is set in \MacroFont we want to be able to switch to indicate module code set in \AltMacroFont. \macro@font keeps track of which one we're using. We can't do the same thing sensibly in OFSS as in NFSS.

```
\@ifundefined{AltMacroFont}{%
  \if@compatibility
```

Again have \small first if we are in compat mode.

```
    \def\AltMacroFont{\small
                      \usefont\encodingdefault
                              \ttdefault
                              \mddefault
                              \sldefault
                      }%
  \else
    \def\AltMacroFont{\fontencoding\encodingdefault
                      \fontfamily\ttdefault
                      \fontseries\mddefault
                      \fontshape\sldefault
                      \small
                      }%
\fi
 }{}
```

To allow changing the \MacroFont in the preamble we defer defining the internally used \macro@font until after the preamble.

```
\AtBeginDocument{\let\macro@font\MacroFont}
```

\check@module
\ifpm@module

This is inserted by \everypar at the start of each macrocode line to check whether it starts with module information. (Such information is of the form %<⟨*switch*⟩>, where the % must be at the start of the line and ⟨*switch*⟩ comprises names with various possible separators and a possible leading +, -, * or / [6]. All that concerns us here is what the first character of ⟨*switch*⟩ is.) First it checks the pm@module flag in case the previous line had a non-block module directive i.e., not %<* or %</; if it did we need to close the group it started and unset the flag. \check@module looks ahead at the next token and then calls \ch@percent to take action depending on whether or not it's a %; we don't want to expand the token at this stage. This is all done conditionally so it can be turned off if it causes problems with code that wasn't designed to be docstripped.

```
\def\check@module{%
  \ifcheck@modules
    \ifpm@module \endgroup \pm@modulefalse \fi
    \expandafter\futurelet\expandafter\next\expandafter\ch@percent
  \fi}
\newif\ifpm@module
```

\DontCheckModules
\CheckModules
\ifcheck@modules

Here are two driver-file interface macros for turning the module checking on and off using the check@modules switch.

```
\def\DontCheckModules{\check@modulesfalse}
\def\CheckModules{\check@modulestrue}
\newif\ifcheck@modules  \check@modulestrue
```

\ch@percent

If the lookahead token in \next is $\%_{12}$ we go on to check whether the following one is < and otherwise do nothing. Note the \expandafter to get past the \fi.

```
\def\ch@percent{%
  \if \percentchar\next
    \expandafter\check@angle
  \fi}
```

\check@angle

Before looking ahead for the < the % is gobbled by the argument here.

```
\def\check@angle#1{\futurelet\next\ch@angle}
```

\ch@angle If the current lookahead token is < we are defined to be processing a module directive can go on to look for + etc.; otherwise we must put back the gobbled %. With LaTeX 2_ε < is active so we have to be a bit careful.

```
\begingroup
\catcode`\<\active
\gdef\ch@angle{\ifx<\next
    \expandafter\ch@plus@etc
  \else \percentchar \fi}
```

\ch@plus@etc \check@plus@etc We now have to decide what sort of a directive we're dealing with and do the right thing with it.

```
\gdef\ch@plus@etc<{\futurelet\next\check@plus@etc}
\gdef\check@plus@etc{%
    \if +\next
      \let\next\pm@module
    \else\if -\next
      \let\next\pm@module
    \else\if *\next
      \let\next\star@module
    \else\if /\next
      \let\next\slash@module
```

At some point in the past the docstrip program was partly rewritten and at that time it also got support for a very special directive of the form %<< followed by an arbitrary string. This is used for "verbatim" inclusion in case of certain problem. We do not really attempt to pretty print that case but we need at least account for it since otherwise we get an error message since this is the only case where we will not have a closing >.

```
    \else\ifx <\next
      \percentchar
    \else
      \let\next\pm@module
    \fi\fi\fi\fi\fi
    \next}
\endgroup
```

\pm@module If we're not dealing with a block directive (* or /) i.e., it's a single special line, we set everything up to the next > appropriately and then change to the special macro font inside a group which will be ended at the start of the next line. If the apparent module directive is missing the terminating > this will lose, but then so will the docstrip implementation. An alternative strategy would be to have \pm@module make > active and clear a flag set here to indicate processing the directive. Appropriate action could then be taken if the flag was found still to be set when processing the next line.

```
\begingroup
\catcode`\~=\active
\lccode`\~=`\>
\lowercase{\gdef\pm@module#1~}{\pm@moduletrue
   \Module{#1}\begingroup
```

We switch to a special font as soon the nesting is higher than the current value of \c@StandardModuleDepth. We do a local update to the \guard@level here which will be restored after the current input line.

```
     \advance\guard@level\@ne
     \ifnum\guard@level>\c@StandardModuleDepth\AltMacroFont\fi
}
```

\star@module \slash@module If the start or end of a module *block* is indicated, after setting the guard we have to check whether a change in the macrocode font should be done. This will be the case if we are already inside a block or are ending the outermost block. If so,

we globally toggle the font for subsequent macrocode sections between the normal and special form, switching to the new one immediately.

```
\lowercase{\gdef\star@module#1~}{%
  \Module{#1}%
  \global \advance \guard@level\@ne
  \ifnum \guard@level>\c@StandardModuleDepth
    \global\let\macro@font=\AltMacroFont \macro@font
  \fi}
\catcode`\>=\active
\gdef\slash@module#1>{%
  \Module{#1}%
  \global \advance \guard@level\m@ne
  \ifnum \guard@level=\c@StandardModuleDepth
    \global\let\macro@font\MacroFont  \macro@font
  \fi
}
\endgroup
```

`\c@StandardModuleDepth` Counter defining up to which level modules are considered part of the main code. If, for example, the whole code is surrounded by a `%<*package>` module we better set this counter to `1` to avoid getting the whole code be displayed in typewriter italic.

```
\newcounter{StandardModuleDepth}
```

`\guard@level` We need a counter to keep track of the guard nesting.

```
\newcount \guard@level
```

`\Module` This provides a hook to determine the way the module directive is set. It gets as argument everything between the angle brackets. The default is to set the contents in sans serif text between ⟨ ⟩ with the special characters suitably `\mathcode`d by `\mod@math@codes`. (You can't just set it in a sans text font because normally `|` will print as an em-dash.) This is done differently depending on whether we have the NFSS or the old one. In the latter case we can easily change `\fam` appropriately.

```
\@ifundefined{Module}{%
```

With NFSS what we probably *should* do is change to a new `\mathversion` but I (Dave Love) haven't spotted an easy way to do so correctly if the document uses a version other than `normal`. (We need to know in what font to set the other groups.) This uses a new math alphabet rather than version and consequently has to worry about whether we're using oldlfnt or not. I expect there's a better way...

```
      \def\Module#1{\mod@math@codes$\langle\mathsf{#1}\rangle$}
  }{}
```

`\mod@math@codes` As well as 'words', the module directive text might contain any of the characters `*/+-,&|!()` for the current version of docstrip. We only need special action for two of them in the math code changing required above: `|` is changed to a `\mathop` (it's normally `"026A`) and `&` is also made a `\mathop`, but in family 0. Remember that `&` will not have a special catcode when it's encountered.

```
\def\mod@math@codes{\mathcode`\|="226A \mathcode`\&="2026}
```

`\mathsf` If NFSS is in use we need a new math alphabet which uses a sans serif font. To support both the release one and two of NFSS the alphabet was renamed to `\mathsf` which is defined in NFSS2.

```
%\ifx\selectfont\undefined
%\else
%  \ifx\mathsf\undefined
%    \newmathalphabet*{\mathsf}{\sfdefault}{m}{n}\fi
%\fi
```

\MacrocodeTopsep
\MacroIndent

In the code above, we have used two registers. Therefore we have to allocate them. The default values might be overwritten with the help of the \DocstyleParms macro.

```
157 \newskip\MacrocodeTopsep \MacrocodeTopsep = 3pt plus 1.2pt minus 1pt
158 \newdimen\MacroIndent
159 \settowidth\MacroIndent{\rmfamily\scriptsize 00\ }
```

\macrocode*
\endmacrocode*

Just as in the verbatim environment, there is also a 'star' variant of the macrocode environment in which a space is shown by the symbol ␣. Until this moment, I have not yet used it (it will be used in the description of the definition of \xmacro@code below) but it's exactly on this one occasion *here* that you can't use it (cf. Münchhausens Marsh problem)[16] directly. Because of this, on this one occasion we'll cheat around the problem with an additional comment character. But now back to \macrocode*. We start with the macro \macro@code which prepares everything and then call the macro \sxmacro@code whose argument is terminated by the string %␣␣␣␣\end{macrocode*}.

```
160 \@namedef{macrocode*}{\macro@code\sxmacro@code}
```

As we know, \sxmacro@code and then \end{macrocode*} (the macro, not the string), will be executed, so that for a happy ending we still need to define the macro \endmacrocode*.

```
161 \expandafter\let\csname endmacrocode*\endcsname = \endmacrocode
```

\xmacro@code

As already mentioned, the macro \xmacro@code expects an argument delimited by the string %␣␣␣␣\end{macrocode}. At the moment that this macro is called, the \catcode of TEX's special characters are 12 ('other') or 13 ('active'). Because of this we need to utilize a different escape character during the definition. This happens locally.

```
162 \begingroup
163 \catcode`\|=\z@␣\catcode`\[=\@ne␣\catcode`\]=\tw@
```

Additionally, we need to ensure that the symbols in the above string contain the \catcodes which are available within the macrocode environment.

```
164 \catcode`\{=12␣\catcode`\}=12
165 \catcode`\%=12␣\catcode`\␣=\active␣\catcode`\\=\active
```

Next follows the actual definition of \macro@code; notice the use of the new escape character. We manage to get the argument surrounded by the string \end{macrocode}, but at the end however, in spite of the actual characters used during the definition of this macro, \end with the argument {macrocode} will be executed, to ensure a balanced environment.

```
166 |gdef|xmacro@code#1%␣␣␣␣\end{macrocode}[#1|end[macrocode]]
```

\sxmacro@code

The definition of \sxmacro@code is completely analogous, only here a slightly different terminating string will be used. Note that the space is not active in this environment.

```
167 |catcode`| =12
168 |gdef|sxmacro@code#1%     \end{macrocode*}[#1|end[macrocode*]]
```

because the \catcode changes have been made local by commencing a new group, there now follows the matching \endgroup in a rather unusual style of writing.

```
169 |endgroup
```

3.3 Macros for the 'documentation parts'

\DescribeMacro
\Describe@Macro
\DescribeEnv
\Describe@Env

The \DescribeMacro and \DescribeEnv macros should print their arguments in the margin and produce an index entry. We simply use \marginpar to get the desired result. This is however not the best solution because the labels might be

[16] Karl Friedrich Hieronymus Frhr. v. Münchhausen (*1720, †1797). Several books were written about fantastic adventures supposedly told by him (see [7] or [1]). In one story he escaped from the marsh by pulling himself out by his hair.

slightly misplaced. One also might get a lot of 'marginpar moved' messages which are hard-wired into the LaTeX output routine.[17] First we change to horizontal mode if necessary. The LaTeX macros `\@bsphack` and `\@esphack` are used to make those commands invisible (i.e. to normalize the surrounding space and to make the `\spacefactor` transparent).

```
\def\DescribeMacro{\leavevmode\@bsphack
```

When documenting the code for the `amstex.sty` option we encountered a bug: the `\catcode` of `@` was active and therefore couldn't be used in command names. So we first have to make sure that we get all `\catcode`s right by calling `\MakePrivateLetters` inside a group. Then we call `\Describe@Macro` to do the work.

```
   \begingroup\MakePrivateLetters\Describe@Macro}
\def\Describe@Macro#1{\endgroup
                \marginpar{\raggedleft\PrintDescribeMacro{#1}}%
```

Note the use of `\raggedleft` to place the output flushed right. Finally we call a macro which produces the actual index entry and finish with `\@esphack` to leave no trace.[18]

```
                \SpecialUsageIndex{#1}\@esphack\ignorespaces}
```

The `\DescribeEnv` macro is completely analogous.

```
\def\DescribeEnv{\leavevmode\@bsphack\begingroup\MakePrivateLetters
  \Describe@Env}
\def\Describe@Env#1{\endgroup
                \marginpar{\raggedleft\PrintDescribeEnv{#1}}%
                \SpecialEnvIndex{#1}\@esphack\ignorespaces}
```

To put the labels in the left margin we have to use the `\reversemarginpar` declaration. (This means that the `doc.sty` can't be used with all classes or packages.) We also make the `\marginparpush` zero and `\marginparwidth` suitably wide.

```
\reversemarginpar
\setlength\marginparpush{0pt}   \setlength\marginparwidth{8pc}
```

`\bslash` We start a new group in which to hide the alteration of `\catcode`s, and make `|` introduce commands, whilst `\` becomes an 'other' character.

```
{\catcode`\|=\z@ \catcode`\\=12
```

Now we are able to define `\bslash` (globally) to generate a backslash of `\catcode` 'other'. We then close this group, restoring original `\catcode`s.

```
|gdef|bslash{\}}
```

`\verbatim` `\verbatim*` The verbatim environment holds no secrets; it consists of the normal LaTeX environment. We also set the `\@beginparpenalty` and change to the font given by `\MacroFont`.

```
\def\verbatim{\@beginparpenalty \predisplaypenalty \@verbatim
                \MacroFont \frenchspacing \@vobeyspaces \@xverbatim}
```

We deal in a similar way with the star form of this environment.

```
\@namedef{verbatim*}{\@beginparpenalty \predisplaypenalty \@verbatim
                \MacroFont \@sxverbatim}
```

`\@verbatim` Additionally we redefine the `\@verbatim` macro so that it suppresses `%` characters at the beginning of the line. The first lines are copied literally from `latex.tex`.

```
\def\@verbatim{\trivlist \item[]\if@minipage\else\vskip\parskip\fi
      \leftskip\@totalleftmargin\rightskip\z@
      \parindent\z@\parfillskip\@flushglue\parskip\z@
      \@@par
      \@tempswafalse
```

[17] It might be better to change these macros into environments like the macro environment.

[18] The whole mechanism won't work because of the `\leavevmode` in front. As a temporary change `\ignorespaces` is added.

`\@verbatim` sets `^^M`, the end of line character, to be equal to `\par`. This control sequence is redefined here; `\@@par` is the paragraph primitive of TEX.

```
\def\par{\if@tempswa\hbox{}\fi\@tempswatrue\@@par
         \penalty\interlinepenalty
```

We add a control sequence `\check@percent` to the definition of `\par` whose task it is to check for a percent character.

```
  \check@percent}%
```

The rest is again copied literally from `latex.tex` (less `\tt`).

```
\obeylines
\let\do\do@noligs \verbatim@nolig@list
\let\do\@makeother \dospecials}
```

`\check@percent` Finally we define `\check@percent`. Since this must compare a character with a percent sign we must first (locally) change percent's `\catcode` so that it is seen by TEX. The definition itself is nearly trivial: grab the following character, check if it is a `%`, and insert it again if not. At the end of the verbatim environment this macro will peek at the next input line. In that case the argument to `\check@percent` might be a `\par` or a macro with arguments. Therefore we make the definition `\long` (`\par` allowed) and use the normal `\next` mechanism to reinsert the argument after the `\fi` if necessary. There is a subtle problem here, the equal sign between `\next` and `#1` is actually necessary. Do you see why? The omission of this token once caused a funny error.

```
{\catcode`\%=12
 \long\gdef\check@percent#1{\ifx #1%\let\next\@empty \else
                                    \let\next=#1\fi \next}}
```

`\verb` We re-define `\verb` to check for newlines in its argument since a missing delimiter is difficult to detect in `doc` source. The code is the saem as in latex.tex of September 19, 1993. Perhaps there should be a font-changing hook rather than just using `\tt`, but if so it probably should be different from `\MacroFont` since that normally includes `\small` and would look wrong inline.

```
\def\verb{\relax\ifmmode\hbox\else\leavevmode\null\fi
  \bgroup \let\do\do@noligs \verbatim@nolig@list
    \ttfamily \verb@eol@error \let\do\@makeother \dospecials
    \@ifstar{\@sverb}{\@vobeyspaces \frenchspacing \@sverb}}
```

`\verb@balance@group`
`\verb@egroup`
`\verb@eol@error`

```
\let\verb@balance@group\@empty

\def\verb@egroup{\global\let\verb@balance@group\@empty\egroup}

\begingroup
  \obeylines%
  \gdef\verb@eol@error{\obeylines%
    \def^^M{\verb@egroup\@latex@error{%
           Text for \noexpand\verb command ended by end of line}\@ehc}}%
\endgroup
```

`\@sverb` See [8] for commentary.

```
\def\@sverb#1{%
  \catcode`#1\active  \lccode`\~`#1%
  \gdef\verb@balance@group{\verb@egroup
     \@latex@error{Illegal use of \noexpand\verb command}\@ehc}%
  \aftergroup\verb@balance@group
  \lowercase{\let~\verb@egroup}}
```

`\verbatim@nolig@list`
`\do@noligs` These macros replace the old `\@noligs` mechanism by an extensible version to allow more ligatures to be added.

```
\def\verbatim@nolig@list{\do\`\do\<\do\>\do\,\do\'\do\-}
```

```
\def\do@noligs#1{%
  \catcode`#1\active
  \begingroup
    \lccode`\~=`#1\relax
    \lowercase{\endgroup\def~{\leavevmode\kern\z@\char`#1}}}
```

\macro
\m@cro@
\macro@cnt

The macro environment is implemented as a trivlist environment, whereby in order that the macro names can be placed under one another in the margin (corresponding to the macro's nesting depth), the macro \makelabel must be altered. In order to store the nesting depth, we use a counter. We also need a counter to count the number of nested macro environments.

```
\newcount\macro@cnt \macro@cnt=0
```

The environment takes an argument—the macro name to be described. Since this name may contain special 'letters' we have to re-\catcode them before scanning the argument. This is done by the \MakePrivateLetters macro.

```
\def\macro{\begingroup
  \catcode`\\12
  \MakePrivateLetters \m@cro@ \iftrue}
```

\environment

The "environment" envrionment will be implemented just like the "macro" environment flagging any differences in the code by passing \iffalse or \iftrue to the \m@cro@ environment doing the actual work.

```
\def\environment{\begingroup
  \catcode`\\12
  \MakePrivateLetters \m@cro@ \iffalse}
```

After scanning the argument we close the group to get the normal \catcodes back. Then we assign a special value to \topsep and start a trivlist environment.

```
\long\def\m@cro@#1#2{\endgroup \topsep\MacroTopsep \trivlist
```

We also save the name being described in \saved@macroname for use in conjunction with the \changes macro.

```
  \edef\saved@macroname{\string#2}%
```

Now there follows a variation of \makelabel which is used should the environment not be nested, or should it lie between two successive \begin{macro} instructions or explanatory text. One can recognize this with the switch \if@inlabel which will be true in the case of successive \item commands.

```
  \def\makelabel##1{\llap{##1}}%
```

If @inlabel is true and if \macro@cnt > 0 then the above definition needs to be changed, because in this case LaTeX would otherwise put the labels all on the same line and this would lead to them being overprinted on top of each other. Because of this \makelabel needs to be redefined in this case.

```
  \if@inlabel
```

If \macro@cnt has the value 1, then we redefine \makelabel so that the label will be positioned in the second line of the margin. As a result of this, two macro names appear correctly, one under the other. It's important whilst doing this that the generated label box is not allowed to have more depth than a normal line since otherwise the distance between the first two text lines of TeX will be incorrectly calculated. The definition should then look like:

```
\def\makelabel##1{\llap{\vtop to \baselineskip
     {\hbox{\strut}\hbox{##1}\vss}}}
```

Completely analogous to this is the case where labels need to be placed one under the other. The lines above are only an example typeset with the verbatim environment. To produce the real definition we save the value of \macro@cnt in \count@ and empty the temp macro \@tempa for later use.

```
    \let\@tempa\@empty \count@\macro@cnt
```

In the following loop we append for every already typeset label an `\hbox{\strut}` to the definition of `\@tempa`.

```
    \loop \ifnum\count@>\z@
      \edef\@tempa{\@tempa\hbox{\strut}}\advance\count@\m@ne \repeat
```

Now be put the definition of `\makelabel` together.

```
    \edef\makelabel##1{\llap{\vtop to\baselineskip
                                {\@tempa\hbox{##1}\vss}}}%
```

Next we increment the value of the nesting depth counter. This value inside the macro environment is always at least one after this point, but its toplevel definition is zero. Provided this environment has been used correctly, `\macro@cnt` = 0 should not occur when `@inlabel`=true. It is however possible if this environment is used within other list environments (but this would have little point).

```
    \advance \macro@cnt \@ne
```

If `@inlabel` is false we reset `\macro@cnt` assuming that there is enough room to print the macro name without shifting.

```
  \else  \macro@cnt\@ne  \fi
```

Now the label will be produced using `\item`. The following line is only a hack saving the day until a better solution is implemented. We have to face two problems: the argument might be a `\par` which is forbidden in the argument of other macros if they are not defined as `\long`, or it is something like `\iffalse` or `\else`, i.e. something which will be misinterpreted when TEX is skipping conditional text. In both cases `\item` will bomb, so we protect the argument by using `\string`.

```
  \edef\@tempa{\noexpand\item[%
```

Depending on whether we are inside a “macro” or “environment” environment we use `\PrintMacroName` or `\PrintEnvName` to display the name.

```
    #1%
      \noexpand\PrintMacroName
    \else
      \noexpand\PrintEnvName
    \fi
    {\string#2}]}%
  \@tempa
```

At this point we also produce an index entry. Because it is not known which index sorting program will be used, we do not use the command `\index`, but rather a command `\SpecialMainIndex` after advancing the counter for indexing by line number. This may be redefined by the user in order to generate an index entry which will be understood by the index program in use (note the definition of `\SpecialMainIndex` for our installation). We advance the current codeline number and after producing an index entry revert to the original value

```
  \global\advance\c@CodelineNo\@ne
```

Again the macro to call depends on the environment we are actually in.

```
  #1%
     \SpecialMainIndex{#2}\nobreak
     \DoNotIndex{#2}%
  \else
     \SpecialMainEnvIndex{#2}\nobreak
  \fi
  \global\advance\c@CodelineNo\m@ne
```

The `\nobreak` is needed to prevent a page break after the `\write` produced by the `\SpecialMainIndex` macro. We exclude the new macro in the cross-referencing feature, to prevent spurious non-main entry references. Regarding possibly problematic arguments, the implementation takes care of `\par` and the conditionals are uncritical.

Because the space symbol should be ignored between the `\begin{macro}{...}` and the following text we must take care of this with `\ignorespaces`.

```
  \ignorespaces}
```

\endmacro
\endenvironment

Older releases of this environment omit the \endgroup token, when being nested. This was done to avoid unnessary stack usage. However it does not work if macro and environment environments are mixed, therefore we now use a simpler approach.

```
\let\endmacro \endtrivlist
\let\endenvironment\endmacro
```

\MacroTopsep

Here is the default value for the \MacroTopsep parameter used above.

```
\newskip\MacroTopsep      \MacroTopsep = 7pt plus 2pt minus 2pt
```

3.4 Formatting the margin

The following three macros should be user definable. Therefore we define those macros only if they have not already been defined.

\PrintMacroName
\PrintEnvName
\PrintDescribeMacro
\PrintDescribeEnv

The formatting of the macro name in the left margin is done by these macros. We first set a \strut to get the height and depth of the normal lines. Then we change to the \MacroFont using \string to \catcode the argument to other (assuming that it is a macro name). Finally we print a space. The font change remains local since this macro will be called inside an \hbox.

```
\@ifundefined{PrintMacroName}
   {\def\PrintMacroName#1{\strut \MacroFont \string #1\ }}{}
```

We use the same formatting conventions when describing a macro.

```
\@ifundefined{PrintDescribeMacro}
   {\def\PrintDescribeMacro#1{\strut \MacroFont \string #1\ }}{}
```

To format the name of a new environment there is no need to use \string.

```
\@ifundefined{PrintDescribeEnv}
   {\def\PrintDescribeEnv#1{\strut \MacroFont #1\ }}{}
\@ifundefined{PrintEnvName}
   {\def\PrintEnvName#1{\strut \MacroFont #1\ }}{}
```

3.5 Creating index entries by scanning 'macrocode'

The following macros ensure that index entries are created for each occurrence of a TEX-like command (something starting with '\') providing indexing has been turned on with \PageIndex or \CodelineIndex. With the default definitions of \SpecialMainIndex, etc., the index file generated is intended to be processed by Chen's makeindex program [4].

Of course, in *this* package file itself we've sometimes had to make | take the rôle of TEX's escape character to introduce command names at places where \ has to belong to some other category. Therefore, we may also need to recognize | as the introducer for a command when setting the text inside the macrocode environment. Other users may have the need to make similar reassignments for their macros.

\SpecialEscapechar
\active@escape@char
\special@escape@char

The macro \SpecialEscapechar is used to denote a special escape character for the next macrocode environment. It has one argument—the new escape character given as a 'single-letter' control sequence. Its main purpose is defining \special@escape@char to produce the chosen escape character \catcoded to 12 and \active@escape@char to produce the same character but with \catcode 13.

The macro \special@escape@char is used to *print* the escape character while \active@escape@char is needed in the definition of \init@crossref to start the scanning mechanism.

In the definition of \SpecialEscapechar we need an arbitrary character with \catcode 13. We use '~' and ensure that it is active. The \begingroup is used to make a possible change local to the expansion of \SpecialEscapechar.

```
\begingroup
\catcode`\~\active
\gdef\SpecialEscapechar#1{%
    \begingroup
```

Now we are ready for the definition of `\active@escape@char`. It's a little tricky: we first define locally the uppercase code of '~' to be the new escape character.

```
    \uccode`\~`#1%
```

Around the definition of `\active@escape@char` we place an `\uppercase` command. Recall that the expansion of `\uppercase` changes characters according to their `\uccode`, but leaves their `\catcode`s untouched (cf. TEXbook page 41).

```
    \uppercase{\gdef\active@escape@char{~}}%
```

The definition of `\special@escape@char` is easier, we use `\string` to `\catcode` the argument of `\SpecialEscapechar` to 12 and suppress the preceding `\escapechar`.

```
    \escapechar\m@ne  \xdef\special@escape@char{\string#1}%
```

Now we close the group and end the definition: the value of `\escapechar` as well as the `\uccode` and `\catcode` of '~' will be restored.

```
  \endgroup}
\endgroup
```

`\init@crossref` The replacement text of `\init@crossref` should fulfill the following tasks:

1) `\catcode` all characters used in macro names to 11 (i.e. 'letter').
2) `\catcode` the '\' character to 13 (i.e. 'active').
3a) `\let` the '\' equal `\scan@macro` (i.e. start the macro scanning mechanism) if there is no special escape character (i.e. the `\special@escape@char` is '\').
3b) Otherwise `\let` it equal `\bslash`, i.e. produce a printable `\`.
4) Make the ⟨*special escape character*⟩ active.
5) `\let` the active version of the special escape character (i.e. the expansion of `\active@escape@char`) equal `\scan@macro`.

The reader might ask why we bother to `\catcode` the '\' first to 12 (at the end of `\macro@code`) then re-`\catcode` it to 13 in order to produce a $\backslash_{12}$ in case 3b) above. This is done because we have to ensure that '\' has `\catcode` 13 within the macrocode environment. Otherwise the delimiter for the argument of `\xmacro@code` would not be found (parameter matching depends on `\catcode`s).

Therefore we first re-`\catcode` some characters.

```
\begingroup   \catcode`\|=\z@  \catcode`\\=\active
```

We carry out tasks 2) and 3b) first.

```
|gdef|init@crossref{|catcode`|\|active   |let\|bslash
```

Because of the popularity of the '@' character as a 'letter' in macros, we normally have to change its `\catcode` here, and thus fulfill task 1). But the macro designer might use other characters as private letters as well, so we use a macro to do the `\catcode` switching.

```
    |MakePrivateLetters
```

Now we `\catcode` the special escape character to 13 and `\let` it equal `\scan@macro`, i.e. fulfill tasks 4) and 5). Note the use of `\expandafter` to insert the chosen escape character saved in `\special@escape@char` and `\active@escape@char`.

```
    |catcode|expandafter`|special@escape@char|active
    |expandafter|let|active@escape@char|scan@macro}
|endgroup
```

If there is no special escape character, i.e. if `\SpecialEscapechar` is `\\`, the second last line will overwrite the previous definition of $\backslash_{13}$. In this way all tasks are fulfilled.

For happy documenting we give default values to `\special@escape@char` and `\active@escape@char` with the following line:

```
\SpecialEscapechar{\\}
```

`\MakePrivateLetters` Here is the default definition of this command, which makes just the `@` into a letter. The user may change it if he/she needs more or other characters masquerading as letters.

```
\@ifundefined{MakePrivateLetters}
    {\let\MakePrivateLetters\makeatletter}{}
```

`\close@crossref` At the end of a cross-referencing part we prepare ourselves for the next one by setting the escape character to '\'.

```
\def\close@crossref{\SpecialEscapechar\\}
```

3.6 Macros for scanning macro names

`\scan@macro` `\macro@namepart` The `\init@crossref` will have made `\active` our `\special@escape@char`, so that each `\active@escape@char` will invoke `\scan@macro` when within the macrocode environment. By this means, we can automatically add index entries for every TEX-like command which is met whilst setting (in verbatim) the contents of macrocode environments.

```
\def\scan@macro{%
```

First we output the character which triggered this macro. Its version `\catcode`d to 12 is saved in `\special@escape@char`. We also call `\step@checksum` to generate later on a proper check-sum (see section 2.12 for details).

```
    \special@escape@char
    \step@checksum
```

If the macrocode environment contains, for example, the command `\\`, the second `\` should not start the scanning mechanism. Therefore we use a switch to decide if scanning of macro names is allowed.

```
    \ifscan@allowed
```

The macro assembles the letters forming a TEX command in `\macro@namepart` so this is initially cleared; we then set `\next` to the *first* character following the `\` and call `\macro@switch` to determine whether that character is a letter or not.

```
        \let\macro@namepart\@empty
        \def\next{\futurelet\next\macro@switch}%
```

As you recognize, we actually did something else, because we have to defer the `\futurelet` call until after the final `\fi`. If, on the other hand, the scanning is disabled we simply `\let \next` equal 'empty'.

```
    \else \let\next\@empty \fi
```

Now we invoke `\next` to carry out what's needed.

```
    \next}
```

`\ifscan@allowed` `\scan@allowedtrue` `\scan@allowedfalse` `\ifscan@allowed` is the switch used above to determine if the `\active@escape@char` should start the macro scanning mechanism.

```
\newif\ifscan@allowed    \scan@allowedtrue
```

`\EnableCrossrefs` `\DisableCrossrefs` At this point we might define two macros which allow the user to disable or enable the cross-referencing mechanism. Processing of files will be faster if only main index entries are generated (i.e., if `\DisableCrossrefs` is in force).

```
\def\DisableCrossrefs{\@bsphack\scan@allowedfalse\@esphack}
```

The macro \EnableCrossrefs will also disable any \DisableCrossrefs command encountered afterwards.

```
\def\EnableCrossrefs{\@bsphack\scan@allowedtrue
                     \def\DisableCrossrefs{\@bsphack\@esphack}\@esphack}
```

\macro@switch Now that we have the character which follows the escape character (in \next), we can determine whether it's a 'letter' (which probably includes @).

If it is, we let \next invoke a macro which assembles the full command name.

```
\def\macro@switch{\ifcat\noexpand\next a%
    \let\next\macro@name
```

Otherwise, we have a 'single-character' command name. For all those single-character names, we use \short@macro to process them into suitable index entries.

```
    \else \let\next\short@macro  \fi
```

Now that we know what macro to use to process the macro name, we invoke it ...

```
  \next}
```

\short@macro This macro will be invoked (with a single character as parameter) when a single-character macro name has been spotted whilst scanning within the macrocode environment.

First we take a look at the \index@excludelist to see whether this macro name should produce an index entry. This is done by the \ifnot@excluded macro which assumes that the macro name is saved in \macro@namepart. The character mustn't be stored with a special category code or exclusion from the index won't work, so we employ the case-changing trick used elsewhere. Since the argument might be an active character, \string is used to normalize it.

```
\begingroup
\catcode`\&=12
\gdef\short@macro#1{\begingroup
  \uccode`\&=\expandafter`\string#1%
  \uppercase{\def\x{\def\macro@namepart{&}}}%
  \expandafter\endgroup\x
  \ifnot@excluded
```

If necessary the index entry is produced by the macro \produce@index. Depending on the actual character seen, this macro has to do different things, so we pass the character as an argument.

```
    \produce@index{#1}\fi
```

Then we disable the cross-referencing mechanism with \scan@allowedfalse and print the actual character. The index entry was generated first to ensure that no page break intervenes (recall that a ^^M will start a new line).

```
    \scan@allowedfalse#1%
```

After typesetting the character we can safely enable the cross-referencing feature again. Note that this macro won't be called (since \macro@switch won't be called) if cross-referencing is globally disabled.

```
    \scan@allowedtrue }
\endgroup
```

\produce@index This macro is supposed to generate a suitable \SortIndex command for a given single-letter control sequence. We test first for the cases which involve active characters (i.e. the backslash, the special escape character (if any), the space and the ^^M). Using the \if test (testing for character codes), we have to ensure that the argument isn't expanded.

```
\def\produce@index#1{%
 \if\noexpand#1\special@escape@char
```

If the character is the special escape character (or the '\' in case there was none) the \it@is@a macro is used to produce the actual \SortIndex call.

```
    \scan@allowedfalse \it@is@a\special@escape@char \else
```

Next comes the test for a '\' which must be the _13 expanding to \bslash.

```
\if\noexpand#1\bslash \it@is@a\bslash \else
```

Another possibility is ␣13. Recall that \space produces a ␣10.

```
\if\noexpand#1\space \it@is@a\space \else
```

The last[19] possibility of an active character is ^^M. In this case we don't test for character codes, since it is easier to look if the character is equal to \par. (We are inside the macrocode environment.)

```
\ifx#1\par
```

If we end up here we have just scanned a \^^M or something similar. Since this will be treated like \␣ by TEX we produce a corresponding index entry.

```
\it@is@a\space \else
```

If it is the token \relax we do nothing. This can't happen when the 'doc' package is used in the way described here, but was added to allow extensions like the idxverb option.

```
\ifx#1\relax \else
```

The next three branches are needed because of bugs in our makeindex program. You can't produce unbalanced index entries[20] and you have to double a percent character. To get around these restrictions we use special macros to produce the \index calls.[21]

```
\if\noexpand#1\bgroup \LeftBraceIndex \else
  \if\noexpand#1\egroup \RightBraceIndex \else
    \if\noexpand#1\percentchar \PercentIndex \else
```

All remaining characters are used directly to produce their index entries. This is possible even for the characters which have special meanings in the index program, provided we quote the characters. (This is correctly done in \it@is@a.)

```
      \it@is@a{\string#1}%
```

We now need a whole pile of \fi s to match up with the \if s.

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

\macro@name We now come to the macro which assembles command names which consist of one or more 'letters' (which might well include @ symbols, or anything else which has a \catcode of 11).

To do this we add the 'letter' to the existing definition of \macro@namepart (which you will recall was originally set to \@empty).

```
\def\macro@name#1{\edef\macro@namepart{\macro@namepart#1}%
```

Then we grab hold of the *next* single character and let \more@macroname determine whether it belongs to the letter string forming the command name or is a 'non-letter'.

```
\futurelet\next\more@macroname}
```

\more@macroname This causes another call of \macro@name to add in the next character, if it is indeed a 'letter'.

```
\def\more@macroname{\ifcat\noexpand\next a%
    \let\next\macro@name
```

Otherwise, it finishes off the index entry by invoking \macro@finish.

```
\else \let\next\macro@finish \fi
```

Here's where we invoke whatever macro was \let equal to \next.

```
\next}
```

[19] Well, it isn't the last active character after all. I added \@noligs some days ago and now ‘ too is active. So we have to make sure that such characters don't get expanded in the index.

[20] This is possible for TEX if you use {_12 or }_12, but makeindex will complain.

[21] Brian Hamilton Kelly has written fixes for all three bugs. When they've found their way through all installations, the lines above will be removed. See page 33 if you already have them. (I'm not sure which versions incorporate these, but 2.11 is OK. See also 8.)

\macro@finish When we've assembled the full 'letter'-string which forms the command name, we set the characters forming the entire command name, and generate an appropriate \index command (provided the command name is not on the list of exclusions). The '\' is already typeset; therefore we only have to output all 'letters' saved in \macro@namepart.

```
\def\macro@finish{%
  \macro@namepart
```

Then we call \ifnot@excluded to decide whether we have to produce an index entry. The construction with \@tempa is needed because we want the expansion of \macro@namepart in the \index command.[22]

```
  \ifnot@excluded
    \edef\@tempa{\noexpand\SpecialIndex{\bslash\macro@namepart}}%
    \@tempa  \fi}
```

3.7 The index exclude list[23]

The internal form of the index exclude list is

⟨*macro name*⟩,⟨*macro name*⟩, ...,

where ⟨*macro name*⟩ is a macro name like \12p11@11 or \12$11. Note that the \ has category 'other' and the other characters in the name are all 'letter', regardless of their normal category.

\DoNotIndex This macro is used to suppress macro names in the index. It starts off with a new group because we have to change the \catcodes of all characters which belong to 'letters' while macros are defined.

```
\def\DoNotIndex{\begingroup \MakePrivateLetters
    \catcode`\\12
```

Then we call the macro which actually reads the argument given by the user.

```
    \do@not@index}
```

\do@not@index We make the \do@not@index macro \long since the user might want to exclude the \par macro.

```
\long\def\do@not@index#1{%
```

It just adds to a token list after finishing the group in which the catcodes were changed.

```
    \endgroup
    \addto@hook\index@excludelist{#1,}}
```

\addto@hook The code for adding tokens (the second argument) to a token list (the first argument) is taken from [8], but it needs to be \long in case \par is amongst the tokens.

```
\long\def\addto@hook#1#2{#1\expandafter{\the#1#2}}
```

\index@excludelist We need an initially-empty register for the excluded list.

```
\newtoks\index@excludelist
\index@excludelist{}
```

\ifnot@excluded
\expanded@notin Now we take a look at the \index@excludelist to see whether a macro name saved in \macro@namepart should produce an index entry. This macro is a pseudo \if; it should expand to \iftrue or \iffalse depending on the contents of \index@excludelist.

```
\begingroup
```

[22] The \index command will expand its argument in the \output routine. At this time \macro@namepart might have a new value.

[23] Warning: the incomplete commentary on \DoNotIndex and the macros it calls was written by Dave Love.

First we change \catcodes so that \ is 'other' and | a temporary for the escape character. This is necessary since our macro names are stored that way in the \index@excludelist.

```
\catcode`\|=0%
\catcode`\\=12
```

Then we define \ifnot@excluded to call \expanded@notin with two arguments: the first is the string \ followed by the contents of \macro@namepart followed by a , and the second is \the followed by \index@excludelist. To achieve the expansion of \macro@namepart, i.e. to pass its contents, we need a suitable number of \expandafters.

```
|gdef|ifnot@excluded{|expandafter
     |expanded@notin|expandafter{|expandafter
     \|macro@namepart,}{|the|index@excludelist}}
|endgroup
```

The macro \expanded@notin now does the dirty work. It first defines a macro \in@@ with a very special parameter text. If you look closely \in@@ has three arguments, the first one is delimited by the first argument of \expanded@notin (i.e. by the string starting with a \ and ending with a , above), the second is undelimited (which means that it will get the next token after our string, and the third is delimited again and will get the rest up to the token \in@@. In other words the token \in@@ is also used as a delimiter.

```
\def\expanded@notin#1#2{%
 \def\in@@##1#1##2##3\in@@{%
```

Now the replacement text simply compares the second argument (i.e. the undelimited one after our string) to the token \expanded@notin. This is an unclosed \ifx statement which means that this macro behaves similar to a normal TEX conditional.

```
  \ifx\expanded@notin##2}%
```

After all these preparations we call \in@@. First we expand the token after \in@@ (which is \the from the second argument to \expanded@notin). As a result we get the contents of the \index@excludelist inserted after \in@@. After this contents we add once more the string we are looking for, then the token \expanded@notin and finally the token \in@@.

```
  \expandafter\in@@#2#1\expanded@notin\in@@}
```

Now what happens when the macro \in@@ above gets called? The first argument to \in@@ is delimited by our string. In other words it will get everything from the contents of \index@excludelist before this string. If the string is not in \index@excludelist then it gets the whole contents, since after it we had inserted the string one more. In this case the next token is \expanded@notin which gets assigned to the second argument and the third argument will be empty. If, on the other hand, the string was inside \index@excludelist then the second argument will not be the token \expanded@notin and the third argument will be all the garbage up to \in@@. Therefore testing the seconded argument, as done in the definition of \in@@ will tell us whether or not the string is in \index@includelist and this was exactly what we wanted. (Deep breath.) You got that?[24]

3.8 Macros for generating index entries

Here we provide default definitions for the macros invoked to create index entries; these are either invoked explicitly, or automatically by \scan@macro. As already mentioned, the definitions given here presuppose that the .idx file will be processed by Chen's makeindex program — they may be redefined for use with the user's favourite such program.

[24] TEXbook page 125. The code described above is originally due to Michael Spivak who used a similar method within the $\mathcal{AMS}$-TEX macros.

To assist the reader in locating items in the index, all such entries are sorted alphabetically *ignoring* the initial '\'; this is achieved by issuing an `\index` command which contains the 'actual' operator for makeindex. The default value for the latter operator is '`@`', but the latter character is so popular in LaTeX package files that it is necessary to substitute another character. This is indicated to makeindex by means of an 'index style file'; the character selected for this function is `=`, and therefore this character too must be specially treated when it is met in a TeX command. A suitable index style file is provided amongst the supporting files for this style file in `gind.ist` and is generated from this source by processing with `docstrip` to extract the module **gind**. A similar style file `gglo.ist` is supplied for sorting the change information in the glossary file and is extracted as module **gglo**. First of all we add some information to the front of the `.ist` files.

```
⟨/package⟩
⟨+gind | gglo⟩%% This is a MAKEINDEX style file which should be used to
⟨+gind⟩%% generate the formatted index for use with the doc
⟨+gglo⟩%% generate the formatted change history for use with the doc
⟨+gind | gglo⟩%% package. The TeX commands used below are defined in
⟨+gind | gglo⟩%% doc.sty.  The commands for MAKEINDEX like 'level'
⟨+gind | gglo⟩%% 'item_x1' are described in '' Makeindex, A General
⟨+gind | gglo⟩%% Purpose, Formatter-Independent Index Processor'' by
⟨+gind | gglo⟩%% Pehong Chen.
⟨+gind | gglo⟩
```

`\actualchar` `\quotechar` `\levelchar` First come the definitions of `\actualchar`, `\quotechar` and `\levelchar`. Note, that our defaults are not the ones used by the makeindex program without a style file.

```
⟨*package⟩
\@ifundefined{actualchar}{\def\actualchar{=}}{}
\@ifundefined{quotechar}{\def\quotechar{!}}{}
\@ifundefined{levelchar}{\def\levelchar{>}}{}
⟨/package⟩
⟨+gind | gglo⟩actual '='
⟨+gind | gglo⟩quote '!'
⟨+gind | gglo⟩level '>'
⟨*package⟩
```

`\encapchar` The makeindex default for the `\encapchar` isn't changed.

```
\@ifundefined{encapchar}{\def\encapchar{|}}{}
```

`\verbatimchar` We also need a special character to be used as a delimiter for the `\verb*` command used in the definitions below.

```
\@ifundefined{verbatimchar}{\def\verbatimchar{+}}{}
```

`\SpecialIndex` The `\SpecialIndex` command creates index entries for macros. If the argument is $\backslash xyz$, the command produces `\indexentry{`xyz`=\verb!*+\`xyz`+}{`n`}` given the above defined defaults for `\actualchar`, `\quotechar` and `\verbatimchar`. We first remove the initial '\' to get a better index.

```
\def\SpecialIndex#1{\@bsphack\special@index{\expandafter\@gobble
                                            \string#1\actualchar
```

Then follows the actual entry. A `\quotechar` is placed before the `*` to allow its use as a special makeindex character. Again `\@bsphack` and `\@esphack` are used to make the macros invisible.

```
      \string\verb\quotechar*\verbatimchar\string#1\verbatimchar}%
    \@esphack}
```

`\SpecialMainIndex` `\SpecialMainEnvIndex` `\SpecialUsageIndex` The `\SpecialMainIndex` macro is used to cross-reference the names introduced by the macro environment. The action is as for `\SpecialIndex`, except that makeindex is instructed to 'encap'sulate the entry with the string `|main` to cause it to generate a call of the `\main` macro.

`\SpecialMainIndex` passes the macro name to be indexed on to the macro `\SpecialIndex@`.

```
\def\SpecialMainIndex#1{\@bsphack\SpecialIndex@{#1}{\encapchar main}%
                        \@esphack}
```

`\SpecialIndex@` The macro `\SpecialIndex@` does the real work for `\SpecialMainIndex` and `\SpecialUsageIndex`. It takes two arguments: the macro to be indexed (as a control sequence or list of character tokens) and the additional text for the index.

```
\begingroup
 \catcode`\|=0
 \catcode`\\=12
 |gdef|@SpecialIndexHelper@#1#2|@nil{%
   |if |noexpand#1\%
     |gdef|@gtempa{#2}%
   |else
     |begingroup
       |escapechar|m@ne
       |expandafter|gdef|expandafter|@gtempa|expandafter{|string#1#2}%
     |endgroup
   |fi}
|endgroup
\def\SpecialIndex@#1#2{%
```

The first thing it does is to convert the macro into a list of characters. Note that a character token list remains (mostly) unchanged.

```
   \@SpecialIndexHelper@#1\@nil
```

The macro name \␣ has to handled in a special way. The reason is that the space token is skipped when TEX is scanning macro parameters, so that the trick used below will not work. So, we check whether the replacement text of `\@tempa` starts with a space token and write the appropriate index entry.

```
   \def\@tempb{ }%
   \ifcat \@tempb\@gtempa
      \special@index{\quotechar\space\actualchar
                     \string\verb\quotechar*\verbatimchar
                     \quotechar\bslash\quotechar\space\verbatimchar#2}%
   \else
```

Having handled this special case we have to distinguish control sequences consisting of one or more letters and those that consists of exactly one nonletter. As character tokens in the replacement text of the macro `\@gtempa` have all category code 12 (other), this is difficult. For simplicity, we treat all single character control sequences alike, irregardless of whether the character is a letter or not. This has the advantage that it works even for strange settings of the category codes.

We define a utility macro `\@tempb` with two arguments, the second delimited by `\relax`. It will be called later so that the first argument is the first character of the macro name, and the second argument receives the rest of the characters. So we distinguish the two cases above by checking whether the second argument is empty.

```
     \def\@tempb##1##2\relax{\ifx\relax##2\relax
```

If so, we define the helper macro `\@tempc` in a way that it adds quotechars in critical places.

```
         \def\@tempc{\special@index{\quotechar##1\actualchar
                     \string\verb\quotechar*\verbatimchar
                     \quotechar\bslash\quotechar##1\verbatimchar#2}}%
```

Otherwise we write the characters as in `\SpecialIndex`.

```
       \else
         \def\@tempc{\special@index{##1##2\actualchar
                     \string\verb\quotechar*\verbatimchar
```

```
                        \bslash##1##2\verbatimchar#2}}%
      \fi}%
```

Now pass the list of characters to tempb and call tempc to do the work.

```
    \expandafter\@tempb\@gtempa\relax
    \@tempc
  \fi}
```

Slightly simpler is the main entry for environments

```
\def\SpecialMainEnvIndex#1{\@bsphack\special@index{%
                                    #1\actualchar
                                    {\string\ttfamily\space#1}
                                       (environment)%
                                    \encapchar main}%
   \special@index{environments:\levelchar#1\actualchar{%
                  \string\ttfamily\space#1}\encapchar
          main}\@esphack}
```

The `\SpecialUsageIndex` is similar to `\SpecialMainIndex`, except that it uses the standard `\index` command. `usage` instead of `main`.

```
\def\SpecialUsageIndex#1{\@bsphack
  {\let\special@index\index\SpecialIndex@{#1}{\encapchar usage}}%
  \@esphack}
```

`\SpecialEnvIndex` Indexing environments is done a little bit differently; we produce two index entries with the `\SpecialEnvIndex` macro:

```
\def\SpecialEnvIndex#1{\@bsphack
```

First we sort the environment under its own name stating in the actual entry that this is an environment.

```
   \index{#1\actualchar{\protect\ttfamily#1}
          (environment)\encapchar usage}%
```

The second entry is sorted as a subitem under the key 'environments:'.

```
   \index{environments:\levelchar#1\actualchar{\protect\ttfamily#1}\encapchar
          usage}\@esphack}
```

Because both entries correspond to 'descriptions' of the environment, we encapsulate the page numbers with the `\usage` macro.

`\SortIndex` This macro is used to generate the index entries for any single-character command that `\scan@macro` encounters. The first parameter specifies the lexical order for the character, whilst the second gives the actual characters to be printed in the entry. It can also be used directly to generate index entries which differ in sort key and actual entry.

```
\def\SortIndex#1#2{\index{#1\actualchar#2}}
```

`\it@is@a` This macro is supposed to produce a correct `\SortIndex` entry for a given character. Since this character might be recognised as a 'command' character by the index program used, all characters are quoted with the `\quotechar`.

```
\def\it@is@a#1{\special@index{\quotechar #1\actualchar
                          \string\verb\quotechar*\verbatimchar
                          \quotechar\bslash\quotechar#1\verbatimchar}}
```

`\LeftBraceIndex` `\RightBraceIndex` These two macros fix the problems with makeindex. Note the 'hack' with `\iffalse}\fi` to satisfy both TEX and the makeindex program. When this is written to the `.idx` file TEX will see both braces (so we get a balanced text). makeindex will also see balanced braces but when the actual index entry is again processed by TEX the brace in between `\iffalse \fi` will vanish.

```
\@ifundefined{LeftBraceIndex}{\def\LeftBraceIndex{%
  \special@index{\bgroup\actualchar\string\verb\quotechar*\verbatimchar
        \quotechar\bslash{\verbatimchar\string\iffalse}\string\fi}}}{}
```

```
\@ifundefined{RightBraceIndex}{\def\RightBraceIndex{%
 \special@index{\egroup\actualchar\string\iffalse{\string\fi\string\verb
           \quotechar*\verbatimchar\quotechar\bslash}\verbatimchar}}}{}
```

\PercentIndex By default we assume a version of makeindex without the percent bug is being used.

```
\@ifundefined{PercentIndex}
  {\def\PercentIndex{\it@is@a\percentchar}}{}
```

\OldMakeindex \percentchar Here is one solution for the percent bug in makeindex. The macro \percentchar denotes a %12. Calling this from a package or the driver file sets things up appropriately.

```
\def\OldMakeindex{\def\PercentIndex{%
    \special@index{\quotechar\percentchar\actualchar\string\verb
           \quotechar*\verbatimchar\quotechar\bslash
           \percentchar\percentchar\verbatimchar}}}
{\catcode`\%=12 \gdef\percentchar{%}}
```

3.9 Redefining the index environment

\ifhave@multicol By default the index is set in three columns, and will start on the same page as, and underneath, the last part of the text of the documented package file, if possible. The last page will be reformatted with balanced columns. This requires the multicols environment which is described elsewhere. So that doc can be run independently of multicol.sty we first check for its existence and set the have@multicol flag appropriately for use below.

```
\newif\ifhave@multicol
```

If we found multicol.sty we use it. It would be nice to delay this (and the re-definition of theindex) until we knew whether an index was actually required ...

```
\IfFileExists{multicol.sty}{\have@multicoltrue
                             \RequirePackage{multicol}%
                            }{}
```

\IndexMin \c@IndexColumns If multicol is in use, when the index is started we compute the remaining space on the current page; if it is greater than \IndexMin, the first part of the index will then be placed in the available space. The number of columns set is controlled by the counter \c@IndexColumns which can be changed with a \setcounter declaration.

```
\newdimen\IndexMin          \IndexMin       = 80pt
\newcount\c@IndexColumns    \c@IndexColumns = 3
```

\theindex Now we start the multi-column mechanism, if appropriate. We use the LaTeX counter \c@IndexColumns declared above to denote the number of columns and insert the 'index prologue' text (which might contain a \section call, etc.). See the default definition for an example.

```
\ifhave@multicol
  \renewenvironment{theindex}
    {\begin{multicols}\c@IndexColumns[\index@prologue][\IndexMin]%
```

Then we make a few last minute assignments to read the individual index \items and finish off by ignoring any initial space.

```
      \IndexParms \let\item\@idxitem \ignorespaces}%
```

\endtheindex At the end of the index, we have only to end the multicols environment.

```
    {\end{multicols}}
```

If we can't use multicols we warn the user and use an environment that's basically the one from article.sty.

```
\else
  \typeout{Can't find multicol.sty -- will use normal index layout if
```

```
    necessary.}
  \def\theindex{\@restonecoltrue\if@twocolumn\@restonecolfalse\fi
    \columnseprule \z@  \columnsep 35\p@
    \twocolumn[\index@prologue]%
    \IndexParms \let\item\@idxitem \ignorespaces}
  \def\endtheindex{\if@restonecol\onecolumn\else\clearpage\fi}
\fi
```

Here are the necessary makeindex declarations. We disable scanning of macro names inside the index with `\scan@allowedfalse\n` to avoid recursion.

```
⟨/package⟩
⟨+gind⟩preamble
⟨+gind⟩"\n \\begin{theindex} \n \\makeatletter\\scan@allowedfalse\n"
⟨+gind⟩postamble
⟨+gind⟩"\n\n \\end{theindex}\n"
⟨*package⟩
```

`\IndexPrologue` `\index@prologue`

The `\IndexPrologue` macro is used to place a short message into the document above the index. It is implemented by redefining `\index@prologue`, a macro which holds the default text. We'd better make it a `\long` macro to allow `\par` commands in its argument.

```
\long\def\IndexPrologue#1{\@bsphack\def\index@prologue{#1}\@esphack}
```

Now we test whether the default is already defined by another package file. If not we define it.

```
\@ifundefined{index@prologue}
     {\def\index@prologue{\section*{Index}%
                 \markboth{Index}{Index}%
                 Numbers written in italic refer to the page
                 where the corresponding entry is described;
                 numbers underlined refer to the
                 \ifcodeline@index
                   code line of the
                 \fi
                 definition; numbers in roman refer to the
                 \ifcodeline@index
                   code lines
                 \else
                   pages
                 \fi
                 where the entry is used.
                 }}{}
```

`\IndexParms`

These are some last-minute assignments for formatting the index entries. They are defined in a separate macro so that a user can substitute different definitions. We start by defining the various parameters controlling leading and the separation between the two columns. The entire index is set in `\small` size.

```
\@ifundefined{IndexParms}
    {\def\IndexParms{%
       \parindent \z@
       \columnsep 15pt
       \parskip 0pt plus 1pt
       \rightskip 15pt
       \mathsurround \z@
       \parfillskip=-15pt
        \small
```

`\@idxitem` `\subitem` `\subsubitem`

Index items are formatted with hanging indentation for any items which may require more than one line.

```
       \def\@idxitem{\par\hangindent 30pt}%
```

Any sub-item in the index is formatted with a 15pt indentation under its main heading.

```
\def\subitem{\@idxitem\hspace*{15pt}}%
```

Whilst sub-sub-items go in a further 10pt.

```
\def\subsubitem{\@idxitem\hspace*{25pt}}%
```

`\indexspace` The makeindex program generates an `\indexspace` before each new alphabetic section commences. After this final definition we end the `\@ifundefined` and the definition of `\IndexParms`.

```
\def\indexspace{\par\vspace{10pt plus 2pt minus 3pt}}%
}}{}
```

`\efill` This definition of `\efill` is intended to be used after index items which have no following text (for example, " *see*" entries). It just ensures that the current line is filled, preventing "`Underfull \hbox`" messages.

```
\def\efill{\hfill\nopagebreak}%
⟨/package⟩
⟨+gind | gglo⟩item_x1   "\\efill \n \\subitem "
⟨+gglo⟩item_x2   "\\ "
⟨+gind⟩item_x2   "\\efill \n \\subsubitem "
⟨*package⟩
```

`\pfill` `\dotfil` `\dotfill` The following definitions provide the `\pfill` command; if this is specified in the index style file to makeindex as the delimiter to appear after index items, then the intervening space before the referenced page numbers will be filled with dots, with a little white space interpolated at each end of the dots. If the line is broken the dots will show up on both lines.

```
\def\dotfill{\leaders\hbox to.6em{\hss .\hss}\hskip\z@ plus  1fill}%
\def\dotfil{\leaders\hbox to.6em{\hss .\hss}\hfil}%
\def\pfill{\unskip~\dotfill\penalty500\strut\nobreak
                \dotfil~\ignorespaces}%
⟨/package⟩
⟨+gind | gglo⟩delim_0   "\\pfill "
⟨+gind | gglo⟩delim_1   "\\pfill "
⟨+gind | gglo⟩delim_2   "\\pfill "
⟨*package⟩
```

`\*` Here is the definition for the `\*` macro. It isn't used in this set of macros.

```
\def\*{\leavevmode\lower.8ex\hbox{$\,\widetilde{\ }\,$}}
```

`\main` The *defining* entry for a macro name is flagged with the string `|main`[25] in the `\index` command; makeindex processes this so that the `\main` macro will be invoked to underline the page number(s) on which the *definition* of the macro will be found.

```
\@ifundefined{main}{\def\main#1{\underline{#1}}}{}
```

`\usage` The `\usage` macro is used to indicate entries describing the usage of a macro. The corresponding page number(s) will be set in *italics*.

```
\@ifundefined{usage}{\def\usage#1{\textit{#1}}}{}
```

`\PrintIndex` This is the same as `\printindex` in the makeidx package.

```
\def\PrintIndex{\@input@{\jobname.ind}%
                \global\let\PrintIndex\@empty}
```

`\printindex` Since the above macro was called `\printindex` in older versions of `doc.sty` the following definition was provided up to version 1.9y.

```
%\def\printindex{\typeout{\string\printindex\space is obsolete!}%
```

[25] With the current definition of `\encapchar` substituted for `|`

```
% \typeout{Please use \string\PrintIndex\space
%          if you are a macro implementor^^J
%          or get a newer version of the documented
%          software if you are a user}%
% \PrintIndex}
```

We want headings in the index (and changes list) according to the initial character of the next block of entries and have to instruct makeindex appropriately. Unfortunately the specification for this changed sometime between versions 2.4 and 2.11 of makeindex. We provide both ways of doing it but unfortunately this will always produce a warning message from makeindex. This is for older versions:

```
⟨/package⟩
⟨+gind, gglo⟩% The next lines will produce some warnings when
⟨+gind, gglo⟩% running Makeindex as they try to cover two different
⟨+gind, gglo⟩% versions of the program:
⟨+gind, gglo⟩lethead_prefix   "{\\bfseries\\hfil "
⟨+gind, gglo⟩lethead_suffix   "\\hfil}\\nopagebreak\n"
⟨+gind⟩lethead_flag       1
⟨+gglo⟩lethead_flag       0
```

This works for newer ones:

```
⟨+gind, gglo⟩heading_prefix   "{\\bfseries\\hfil "
⟨+gind, gglo⟩heading_suffix   "\\hfil}\\nopagebreak\n"
⟨+gind⟩headings_flag       1
⟨+gglo⟩headings_flag       0
⟨*package⟩
```

3.10 Dealing with the change history[26]

To provide a change history log, the `\changes` command has been introduced. This takes three arguments, respectively, the version number of the file, the date of the change, and some detail regarding what change has been made. The second of these arguments is otherwise ignored, but the others are written out and may be used to generate a history of changes, to be printed at the end of the document. However, note that older versions of Chen's standard makeindex program limit any textual field to just 64 characters; therefore, is important that the number of characters in the second and third parameters should not exceed 61 altogether (to allow for the parentheses placed around the date).

`\changes` The output of the `\changes` command goes into the ⟨*Glossary_File*⟩ and therefore uses the normal `\glossaryentry` commands.[27] Thus makeindex or a similar program can be used to process the output into a sorted "glossary". The `\changes` command commences by taking the usual measures to hide its spacing, and then redefines `\protect` for use within the argument of the generated `\indexentry` command.

We re-code nearly all chars found in `\sanitize` to letter since the use of special package which make some characters active might upset the `\changes` command when writing its entries to the file. However we have to leave `%` as comment and ␣ as ⟨*space*⟩ otherwise chaos will happen. And, of course the `\` should be available as escape character.

```
\def\changes{\@bsphack\begingroup\@sanitize
   \catcode`\\\z@ \catcode`\ 10 \MakePercentIgnore
   \changes@}
\def\changes@#1#2#3{%
  \protected@edef\@tempa{\noexpand\glossary{#1\levelchar
```

[26] The whole section was proposed by Brian Hamilton Kelly. He also documented and debugged the macros as well as many other parts of this package.

[27] Note that a recent change in LaTeX 2.09 changed the command name in the `.glo` file from `\indexentry` to `\glossaryentry`. It is therefore necessary to have a special makeindex style file called `gglo.ist` to process this file correctly.

If the macro \saved@macroname doesn't contain any macro name (ie is empty) the current changes entry was done at top-level. In this case we preceed it by \generalname.

```
                                \ifx\saved@macroname\@empty
                                  \space
                                  \actualchar
                                  \generalname
                                \else
                                  \expandafter\@gobble
                                  \saved@macroname
                                  \actualchar
                                  \string\verb\quotechar*%
                                  \verbatimchar\saved@macroname
                                  \verbatimchar
                                \fi
                                :\levelchar #3}}%
  \@tempa\endgroup\@esphack}
```

\saved@macroname The entries are sorted for convenience by the name of the most recently introduced macroname (i.e., that in the most recent \begin{macro} command). We therefore provide \saved@macroname to record that argument, and provide a default definition in case \changes is used outside a macro environment. (This is a *wicked* hack to get such entries at the beginning of the sorted list! It works providing no macro names start with ! or ".)

```
\def\saved@macroname{}
```

\generalname This macro holds the string placed before changes entries on top-level.

```
\def\generalname{General}
```

\RecordChanges To cause the changes to be written (to a .glo) file, we define \RecordChanges to invoke LaTeX's usual \makeglossary command.

```
\let\RecordChanges\makeglossary
```

\GlossaryMin
\c@GlossaryColumns The remaining macros are all analogues of those used for the theindex environment. When the glossary is started we compute the space which remains at the bottom of the current page; if this is greater than \GlossaryMin then the first part of the glossary will be placed in the available space. The number of columns set are controlled by the counter \c@GlossaryColumns which can be changed with a \setcounter declaration.

```
\newdimen\GlossaryMin         \GlossaryMin       = 80pt
\newcount\c@GlossaryColumns   \c@GlossaryColumns = 2
```

\theglossary
\endglossary The environment theglossary is defined in the same manner as the theindex environment.

```
\ifhave@multicol
  \newenvironment{theglossary}{%
    \begin{multicols}\c@GlossaryColumns
                     [\glossary@prologue][\GlossaryMin]%
    \GlossaryParms \let\item\@idxitem \ignorespaces}%
   {\end{multicols}}
\else
  \newenvironment{theglossary}{%
      \@restonecoltrue\if@twocolumn\@restonecolfalse\fi
      \columnseprule \z@  \columnsep 35\p@
      \twocolumn[\glossary@prologue]%
      \GlossaryParms \let\item\@idxitem \ignorespaces}
    {\if@restonecol\onecolumn\else\clearpage\fi}
\fi
```

Here are the necessary makeindex declarations with scanning disabled as for the index.

```
⟨/package⟩
⟨+gglo⟩preamble
⟨+gglo⟩"\n \\begin{theglossary} \n
⟨+gglo⟩     \\makeatletter\\scan@allowedfalse\n"
⟨+gglo⟩postamble
⟨+gglo⟩"\n\n \\end{theglossary}\n"
```

This difference from `gind.ist` is necessary if you have an up-to-date LaTeX.

```
⟨+gglo⟩keyword "\\glossaryentry"
⟨*package⟩
```

`\GlossaryPrologue` `\glossary@prologue`

The `\GlossaryPrologue` macro is used to place a short message above the glossary into the document. It is implemented by redefining `\glossary@prologue`, a macro which holds the default text. We better make it a long macro to allow `\par` commands in its argument.

```
\long\def\GlossaryPrologue#1{\@bsphack
                             \def\glossary@prologue{#1}%
                             \@esphack}
```

Now we test whether the default is already defined by another package file. If not we define it.

```
\@ifundefined{glossary@prologue}
    {\def\glossary@prologue{\section*{{Change History}}%
                \markboth{{Change History}}{{Change History}}%
                }}{}
```

`\GlossaryParms`

Unless the user specifies otherwise, we set the change history using the same parameters as for the index.

```
\@ifundefined{GlossaryParms}{\let\GlossaryParms\IndexParms}{}
```

`\PrintChanges`

To read in and print the sorted change history, just put the `\PrintChanges` command as the last (commented-out, and thus executed during the documentation pass through the file) command in your package file. Alternatively, this command may form one of the arguments of the `\StopEventually` command, although a change history is probably *not* required if only the description is being printed.

The command assumes that makeindex or some other program has processed the `.glo` file to generate a sorted `.gls` file.

```
\def\PrintChanges{\@input@{\jobname.gls}%
                  \global\let\PrintChanges\@empty}
```

3.11 Bells and whistles

`\StopEventually` `\Finale` `\AlsoImplementation` `\OnlyDescription`

If `\AlsoImplementation` is in force the whole documentation including the code part will be typeset. This is the default.

```
\newcommand\AlsoImplementation{%
```

To make this happen we have to define `\StopEventually` in a way that its argument is typeset at the very end or more exactly at `\Finale`. For this we save its argument in the macro `\Finale`.

```
  \long\def\StopEventually##1{\@bsphack\gdef\Finale{##1%
```

But `\Finale` will be called at the very end of a file. This is exactly the point were we want to know if the file is uncorrupted. Therefore we also call `\check@checksum` at this point.

```
                  \check@checksum}%
```

On the other hand: `\StopEventually` is more or less a dividing point between description and code. So we start to look for the check-sum of the documented file by calling `\init@checksum`.

```
              \init@checksum
```

```
615                \@esphack}%
616          }
```

Since `\AlsoImplementation` should be the default we execute it and thus `\StopEventually` gets the desired meaning.

```
617 \AlsoImplementation
```

When the user places an `\OnlyDescription` declaration in the driver file the document should only be typeset up to `\StopEventually`. We therefore have to redefine this macro.

```
618 \def\OnlyDescription{\@bsphack\long\def\StopEventually##1{%
```

In this case the argument of `\StopEventually` should be set and afterwards TEX should stop reading from this file. Therefore we finish this macro with

```
619             ##1\endinput}\@esphack}
```

If no `\StopEventually` command is given we silently ignore a `\Finale` issued.

```
620 \let\Finale\relax
```

`\meta` The `\meta` macro is a bit tricky. We want to allow line breaks at blanks in the argument but we don't want a break in between. In the past this was done by defining `\meta` in a way that a ␣ is active when the argument is scanned. Words are then scanned into `\hbox`es. The active ␣ will end the preceding `\hbox` add an ordinary space and open a new `\hbox`. In this way breaks are only possible at spaces. The disadvantage of this method was that `\meta` was neither robust nor could it be `\protect`ed. The new implementation fixes this problem by defining `\meta` in a radically different way: we prevent hypenation by defining a `\language` which has no patterns associated with it and use this to typeset the words within the angle brackets.

```
621 \ifx\l@nohyphenation\undefined
622   \newlanguage\l@nohyphenation
623 \fi
```

```
624 \DeclareRobustCommand\meta[1]{%
```

Since the old implementation of `\meta` could be used in math we better ensure that this is possible with the new one as well. So we use `\ensuremath` around `\langle` and `\rangle`. However this is not enough: if `\meta@font@select` below expands to `\itshape` it will fail if used in math mode. For this reason we hide the whole thing inside an `\nfss@text` box in that case.

```
625      \ensuremath\langle
626      \ifmmode \expandafter \nfss@text \fi
627      {%
628       \meta@font@select
```

Need to keep track of what we changed just in case the user changes font inside the argument so we store the font explicitly.

```
629       \edef\meta@hyphen@restore
630         {\hyphenchar\the\font\the\hyphenchar\font}%
631       \hyphenchar\font\m@ne
632       \language\l@nohyphenation
633       #1\/%
634       \meta@hyphen@restore
635      }\ensuremath\rangle
636 }
```

`\meta@font@select` Maske font used inside `\meta` customizable.

```
637 \def\meta@font@select{\itshape}
```

`\IndexInput` This next macro may be used to read in a separate file (possibly a package file that is *not* documented by this means) and set it verbatim, whilst scanning for macro names and indexing the latter. This could be a useful first pass in preparing to generate documentation for the file read.

```
638 \def\IndexInput#1{%
```

We commence by setting up a group, and initializing a `\trivlist` as is normally done by a `\begin{macrocode}` command.

```
\begingroup \macro@code
```

We also make spacing behave as in the macrocode environment, because otherwise all the spaces will be shown explicitly.

```
\frenchspacing \@vobeyspaces
```

Then it only remains to read in the specified file, and finish off the `\trivlist`.

```
\input{#1}\endmacrocode
```

Of course, we need to finish off the group as well.

```
\endgroup}
```

`\maketitle` The macro to generate titles is easily altered in order that it can be used more than once (an article with many titles). In the original, diverse macros were concealed after use with `\relax`. We must cancel anything that may have been put into `\@thanks`, etc., otherwise *all* titles will carry forward any earlier such setting!

```
\def\maketitle{\par
  \begingroup \def \thefootnote {\fnsymbol {footnote}}%
  \setcounter {footnote}\z@
  \def\@makefnmark{\hbox to\z@{$\m@th^{\@thefnmark}$\hss}}%
  \long\def\@makefntext##1{\parindent 1em\noindent
        \hbox to1.8em{\hss$\m@th^{\@thefnmark}$}##1}%
  \if@twocolumn \twocolumn [\@maketitle ]%
  \else \newpage \global \@topnum \z@ \@maketitle \fi
```

For special formatting requirements (such as in TUGboat), we use pagestyle `titlepage` for this; this is later defined to be `plain`, unless already defined, as, for example, by `ltugboat.sty`.

```
  \thispagestyle{titlepage}\@thanks \endgroup
```

If the driver file documents many files, we don't want parts of a title of one to propagate to the next, so we have to cancel these:

```
  \setcounter {footnote}\z@
  \gdef\@date{\today}\gdef\@thanks{}%
  \gdef\@author{}\gdef\@title{}}
```

`\ps@titlepage` When a number of articles are concatenated into a journal, for example, it is not usual for the title pages of such documents to be formatted differently. Therefore, a class such as ltugboat can define this macro in advance. However, if no such definition exists, we use pagestyle `plain` for title pages.

```
\@ifundefined{ps@titlepage}
   {\let\ps@titlepage=\ps@plain}{}
```

`\MakeShortVerb` This arranges an abbreviation for `\verb` such that if you say `\MakeShortVerb{\⟨c⟩}` subsequently using ⟨*c*⟩⟨*text*⟩⟨*c*⟩ is equivalent to `\verb`⟨*c*⟩⟨*text*⟩⟨*c*⟩.[28] In addition, the fact that ⟨*c*⟩ is made active is recorded for the benefit of the verbatim and macrocode environments. Note particularly that the definitions below are global. The first thing we do (it needn't be first) is to record the—presumably new—special character in `\dospecials` and `\@sanitize` using `\add@special`.

Some unwary user might issue `\MakeShortVerb` for a second time, we better protect against this. We assume that this happened if a control sequence `\cc\`⟨*c*⟩ is bound, the probability that this name is used by another module is low. We will output a warning below, so that a possible error might be noticed by the programmer if he reads the `LOG` file. (Should have used module internal names, 'though.)

`\MakeShortVerb*` This arranges an abbreviation for `\verb*` such that if you say `\MakeShortVerb*{\⟨c⟩}` subsequently using ⟨*c*⟩⟨*text*⟩⟨*c*⟩ is equivalent to `\verb*`⟨*c*⟩⟨*text*⟩⟨*c*⟩.

[28] Warning: the commentary in the rest of this section was written by Dave Love.

```
⟨/package⟩
⟨*package | shortvrb⟩
\def\MakeShortVerb{%
  \@ifstar
    {\def\@shortvrbdef{\verb*}\@MakeShortVerb}%
    {\def\@shortvrbdef{\verb}\@MakeShortVerb}}

\def\@MakeShortVerb#1{%
  \expandafter\ifx\csname cc\string#1\endcsname\relax

    \@shortvrbinfo{Made }{#1}\@shortvrbdef
    \add@special{#1}%
```

Then the character's current catcode is stored in `\cc\`⟨*c*⟩.

```
    \expandafter
    \xdef\csname cc\string#1\endcsname{\the\catcode`#1}%
```

The character is spliced into the definition using the same trick as used in `\verb` (for instance), having activated `~` in a group.

```
    \begingroup
      \catcode`\~\active  \lccode`\~`#1%
      \lowercase{%
```

The character's old meaning is recorded in `\ac\`⟨*c*⟩ prior to assigning it a new one.

```
        \global\expandafter\let
           \csname ac\string#1\endcsname~%
        \expandafter\gdef\expandafter~\expandafter{\@shortvrbdef~}}%
    \endgroup
```

Finally the character is made active.

```
    \global\catcode`#1\active
```

If we suspect that ⟨*c*⟩ is already a short reference, we tell the user. Now he or she is responsible if anything goes wrong . . .

```
  \else

    \@shortvrbinfo\@empty{#1 already}{\@empty\verb(*)}%
  \fi}
```

`\DeleteShortVerb` Here's the means of undoing a `\MakeShortVerb`, for instance in a region where you need to use the character outside a verbatim environment. It arranges for `\dospecials` and `\@sanitize` to be altered appropriately, restores the saved catcode and, if necessary, the character's meaning (as stored by `\MakeShortVerb`). If the catcode wasn't stored in `\cc\`⟨*c*⟩ (by `\MakeShortVerb`) the command is silently ignored.

```
\def\DeleteShortVerb#1{%
  \expandafter\ifx\csname cc\string#1\endcsname\relax

    \@shortvrbinfo\@empty{#1 not}{\@empty\verb(*)}%
  \else

    \@shortvrbinfo{Deleted }{#1 as}{\@empty\verb(*)}%
    \rem@special{#1}%
    \global\catcode`#1\csname cc\string#1\endcsname
```

We must not forget to reset `\cc\`⟨*c*⟩, otherwise the check in `\MakeShortVerb` for a repeated definition will not work.

```
    \global \expandafter\let \csname cc\string#1\endcsname \relax
    \ifnum\catcode`#1=\active
      \begingroup
        \catcode`\~\active   \lccode`\~`#1%
        \lowercase{%
          \global\expandafter\let\expandafter~%
          \csname ac\string#1\endcsname}%
      \endgroup \fi \fi}
```

\@shortvrbinfo Helper function for info messages.

```
\def\@shortvrbinfo#1#2#3{%
⟨shortvrb⟩   \PackageInfo{shortvrb}{%
⟨!shortvrb⟩   \PackageInfo{doc}{%
     #1\expandafter\@gobble\string#2 a short reference
                                          for \expandafter\string#3}}
```

\add@special This helper macro adds its argument to the \dospecials macro which is conventionally used by verbatim macros to alter the catcodes of the currently active characters. We need to add \do\⟨c⟩ to the expansion of \dospecials after removing the character if it was already there to avoid multiple copies building up should \MakeShortVerb not be balanced by \DeleteShortVerb (in case anything that uses \dospecials cares about repetitions).

```
\def\add@special#1{%
  \rem@special{#1}%
  \expandafter\gdef\expandafter\dospecials\expandafter
    {\dospecials \do #1}%
```

Similarly we have to add \@makeother\⟨c⟩ to \@sanitize (which is used in things like \index to re-catcode all special characters except braces).

```
  \expandafter\gdef\expandafter\@sanitize\expandafter
    {\@sanitize \@makeother #1}}
```

\rem@special The inverse of \add@special is slightly trickier. \do is re-defined to expand to nothing if its argument is the character of interest, otherwise to expand simply to the argument. We can then re-define \dospecials to be the expansion of itself. The space after ='##1 prevents an expansion to \relax!

```
\def\rem@special#1{%
  \def\do##1{%
    \ifnum'#1='##1 \else \noexpand\do\noexpand##1\fi}%
  \xdef\dospecials{\dospecials}%
```

Fixing \@sanitize is the same except that we need to re-define \@makeother which obviously needs to be done in a group.

```
  \begingroup
    \def\@makeother##1{%
      \ifnum'#1='##1 \else \noexpand\@makeother\noexpand##1\fi}%
    \xdef\@sanitize{\@sanitize}%
  \endgroup}
⟨/package | shortvrb⟩
⟨*package⟩
```

\MakeShortverb These commands from newdoc are now obsolete.

\DeleteShortverb

```
\def\MakeShortverb{\typeout{*** Switch to \noexpand\MakeShortVerb
                          syntax, this is obsolete ***}\MakeShortVerb}
\def\DeleteShortverb{\typeout{*** Switch to \noexpand\DeleteShortVerb
                          syntax, this is obsolete ***}\DeleteShortVerb}
```

3.12 Providing a checksum and character table[29]

\init@checksum The checksum mechanism works by counting backslashes in the macrocode. This initialises the count (when called from \StopEventually).

```
\def\init@checksum{\relax
    \global\bslash@cnt\z@}
```

\check@checksum This reports the sum compared with the value (\bslash@cnt) the file advertises. It's called from \Finale (if that hasn't been re-defined).

```
\def\check@checksum{\relax
     \ifnum\check@sum=\z@
```

[29] Warning: the commentary in this section was written by Dave Love.

```
      \typeout{**********************************}%
      \typeout{* This macro file has no checksum!}%
      \typeout{* The checksum should be \the\bslash@cnt!}%
      \typeout{**********************************}%
    \else
      \ifnum\check@sum=\bslash@cnt
        \typeout{********************}%
        \typeout{* Checksum passed *}%
        \typeout{********************}%
      \else
        \PackageError{doc}{Checksum not passed
                   (\the\check@sum<>\the\bslash@cnt)}%
         {The file currently documented seems to be wrong.^^J%
          Try to get a correct version.}%
      \fi
    \fi
    \global\check@sum\z@}
```

`\check@sum` `\bslash@cnt` We need to define counters, `\bslash@cnt` for the number of backslashes counted and `\check@sum` for the value advertised by the file.

```
\newcount\check@sum           \check@sum  = \z@
\newcount\bslash@cnt          \bslash@cnt = \z@
```

`\CheckSum` This is the interface to setting `\check@sum`.

```
\def\CheckSum#1{\@bsphack\global\check@sum#1\relax\@esphack}
```

`\step@checksum` This advances the count when a backslash is encountered in the macrocode.

```
\def\step@checksum{\global\advance\bslash@cnt\@ne}
```

`\CharacterTable` The user interface to the character table-checking does some `\catcode`ing and then compares the following table with the stored version. We need to have `@` of type "other" within the table since this is the way it is usually returned when reading in a normal document. To nevertheless have a private letter we use `~` for this purpose. `~` does no harm as a "letter" as it comes last in the table and therefore will not gobble following space.

```
\def\CharacterTable{\begingroup \CharTableChanges \character@table}
```

`\character@table` This does the work of comparing the tables and reporting the result. Note that the following code is enclosed in a group with `~` catcoded to letter.

```
\begingroup
  \catcode`\~=11
  \gdef\character@table#1{\def\used~table{#1}%
      \ifx\used~table\default~table
           \typeout{**************************}%
           \typeout{* Character table correct *}%
           \typeout{**************************}%
      \else
         \PackageError{doc}{Character table corrupted}
                           {\the\wrong@table}
         \show\default~table
         \show\used~table
      \fi
      \endgroup}
```

`\CharTableChanges` When the character table is read in we need to scan it with a fixed set of `\catcode`s. The reference table below was defined by assuming the normal `\catcode`s of TEX, i.e. `@` is of type other and the only token of type "letter" are the usual letters of the alphabet. If, for some reason, other characters are made "letters" then their `\catcode`s need to be restored before checking the table. Otherwise spaces in the table are gobbled and we get the information that the tables are different, even

if they are actually equal. For this reason `\CharTableChanges` can be set up to locally restore the `\catcode`s of such "letters" to "other".

```
\global\let\CharTableChanges\@empty
```

`\default~table` Here's what the table *should* look like (modulo spaces).

```
\makeatother
\gdef\default~table
   {Upper-case    \A\B\C\D\E\F\G\H\I\J\K\L\M\N\O\P\Q\R\S\T\U\V\W\X\Y\Z
    Lower-case    \a\b\c\d\e\f\g\h\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\y\z
    Digits        \0\1\2\3\4\5\6\7\8\9
    Exclamation   \!     Double quote  \"     Hash (number) \#
    Dollar        \$     Percent       \%     Ampersand     \&
    Acute accent  \’     Left paren    \(     Right paren   \)
    Asterisk      \*     Plus          \+     Comma         \,
    Minus         \-     Point         \.     Solidus       \/
    Colon         \:     Semicolon     \;     Less than     \<
    Equals        \=     Greater than  \>     Question mark \?
    Commercial at \@     Left bracket  \[     Backslash     \\
    Right bracket \]     Circumflex    \^     Underscore    \_
    Grave accent  \‘     Left brace    \{     Vertical bar  \|
    Right brace   \}     Tilde         \~}
\endgroup
```

`\wrong@table` We need a help message in case of problems.

```
\newhelp\wrong@table{Some of the ASCII characters are corrupted.^^J
           I now \string\show\space you both tables for comparison.}
```

3.13 Attaching line numbers to code lines[30]

The code in this section allows index entries to refer to code line numbers—the number of the first line of macrocode in the macro environment.

`\codeline@index` Indexing by code line is controlled by the `codeline@index` switch.

`\CodelineNumbered`

```
\newif\ifcodeline@index \codeline@indexfalse
\let\CodelineNumbered\codeline@indextrue
```

`\codeline@wrindex` The code index entries are written out by `\special@index`. If indexing is by code line this is `\let` to `\codeline@wrindex`; if indexing is by page it is just `\index`. However, if `\nofiles` is given, we omit writing such an index entry at all.

```
\def\codeline@wrindex#1{\if@filesw
        \immediate\write\@indexfile
            {\string\indexentry{#1}%
            {\number\c@CodelineNo}}\fi}
```

`\special@index` By default no index entries are written out.

```
\let\special@index = \@gobble
```

`\CodelineIndex` This switches on use of the index file with `\makeindex`, sets the switch to indicate code line numbering and defines `\special@index` appropriately.

```
\def\CodelineIndex{\makeindex
                   \codeline@indextrue
                   \let\special@index\codeline@wrindex}
```

`\PageIndex` `\PageIndex` is similar.

```
\def\PageIndex{\makeindex
               \codeline@indexfalse
               \let\special@index\index}
```

[30] Warning: the commentary was written by Dave Love.

\c@CodelineNo We need a counter to keep track of the line number.

```
\newcount\c@CodelineNo  \c@CodelineNo\z@
```

\theCodelineNo This provides a hook to control the format of line numbers which may be defined in a class file.

```
\@ifundefined{theCodelineNo}
  {\ifx\selectfont\undefined
     \def\theCodelineNo{\rmfamily\scriptsize\arabic{CodelineNo}}%
   \else
     \def\theCodelineNo{\reset@font\scriptsize\arabic{CodelineNo}}%
   \fi}
  {}
```

3.14 Layout Parameters for documenting package files

\tolerance People documenting package files would probably rather have things "sticking out" in overfull \hboxes and poorish spacing, because they probably don't want to spend a lot of time on making all the line breaks perfect!

```
        \tolerance=1000\relax
```

The following \mathcode definitions allow the characters '\' and '@' to appear in \ttfamily font when invoked in math mode;[31] particularly for something like \@abc = 1.

If an *old* version of the german package is in force, then the '"' character is active and would upset the definition of the ⟨*16-bit number*⟩ quantities below, therefore we change the \catcode of " inside a group, and use \global.

```
{ \catcode`\"=12
  \global\mathcode`\\="705C \global\mathcode`\@="7040 }
```

\DocstyleParms This macro can be used, for example, to assign new values to \MacrocodeTopsep and \MacroIndent and some other internal registers. If it is already defined, the default definition won't be carried out. Note that it is necessary to assign new values via this macro if it should be done in a class file (like ltugboat.cls for example) since the registers are undefined before doc.sty is read in. The default values for the internal registers are scattered over this file.

```
\@ifundefined{DocstyleParms}{}{}
```

Now we allow overwriting the values by calling \DocstyleParms.

```
\DocstyleParms      \let\DocstyleParms\relax
```

\AmSTeX \BibTeX \SliTeX Here are a few definitions which can usefully be employed when documenting package files: now we can readily refer to $\mathcal{AMS}$-TEX, BIBTEX and SLITEX, as well as the usual TEX and LATEX.

```
\@ifundefined{AmSTeX}
   {\def\AmSTeX{\leavevmode\hbox{$\mathcal A\kern-.2em\lower.376ex%
        \hbox{$\mathcal M$}\kern-.2em\mathcal S$-\TeX}}}{}
\@ifundefined{BibTeX}
   {\def\BibTeX{{\rmfamily B\kern-.05em%
    \textsc{i\kern-.025em b}\kern-.08em%
    T\kern-.1667em\lower.7ex\hbox{E}\kern-.125emX}}}{}
\@ifundefined{SliTeX}
   {\def\SliTeX{{\rmfamily S\kern-.06emL\kern-.18em\raise.32ex\hbox
                {\scshape i}\kern -.03em\TeX}}}{}
```

[31] You may wonder why the definitions state that both characters belong to the *variable family* (i.e. the number 7 in front). The reason is this: Originally the \mathcode of \ was defined to be "075C, i.e. ordinary character number 92 (hex 5C) in math family number 7 which is the typewriter family in standard LATEX. But this file should not depend on this specific setting, so I changed these \mathcodes to work with any family assignments. For an example see the article about the new font selection scheme.

\PlainTeX There's even a Plain TeX and a Web.

\Web
```
\@ifundefined{PlainTeX}{\def\PlainTeX{\textsc{Plain}\kern2pt\TeX}}{}
\@ifundefined{Web}{\def\Web{\textsc{Web}}}{}
```

3.15 Changing the \catcode of %

\MakePercentIgnore \MakePercentComment And finally the most important bit: we change the \catcode of '%' so that it is ignored (which is how we are able to produce this document!). We provide two commands to do the actual switching.

```
\def\MakePercentIgnore{\catcode`\%9\relax}
\def\MakePercentComment{\catcode`\%14\relax}
```

\DocInput The two macros above are now used to define the \DocInput macro which was introduced in version v1.5l (or so) of the doc package. In older versions \MakePercentIgnore was placed at the very end of doc.sty.

```
\def\DocInput#1{\MakePercentIgnore\input{#1}\MakePercentComment}
```

3.16 GetFileInfo

\GetFileInfo Define \filedate and friends from info in the \ProvidesPackage etc. commands.

```
\def\GetFileInfo#1{%
  \def\filename{#1}%
  \def\@tempb##1 ##2 ##3\relax##4\relax{%
    \def\filedate{##1}%
    \def\fileversion{##2}%
    \def\fileinfo{##3}}%
  \edef\@tempa{\csname ver@#1\endcsname}%
  \expandafter\@tempb\@tempa\relax? ? \relax\relax}
```

We can now finish the docstrip main module.

```
⟨/package⟩
```

References

[1] G. A. Bürger. Wunderbare Reisen zu Wasser und zu Lande, Feldzüge und lustige Abenteuer des Freyherrn v. Münchhausen. London, 1786 & 1788.

[2] D. E. Knuth. Literate Programming. Computer Journal, Vol. 27, *pp.* 97–111, May 1984.

[3] D. E. Knuth. Computers & Typesetting (The TeXbook). Addison-Wesley, Vol. A, 1986.

[4] L. Lamport. MakeIndex: An Index Processor for LaTeX. 17 February 1987. (Taken from the file makeindex.tex provided with the program source code.)

[5] Frank Mittelbach. The doc-option. *TUGboat*, Vol. 10(2), *pp.* 245–273, July 1989.

[6] Frank Mittelbach, Denys Duchier and Johannes Braams. docstrip.dtx (to appear). The file is part of the DOC package.

[7] R. E. Raspe (*1737, †1797). Baron Münchhausens narrative of his marvellous travels and campaigns in Russia. Oxford, 1785.

[8] Rainer Schöpf. A New Implementation of LaTeX's verbatim and verbatim* Environments. File verbatim.doc, version 1.4i.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

F

G

H

I

L

M

N

O

P

Change History

BHK

- \c@GlossaryColumns: Added to support \changes. 37
- \changes: Changed definition of \protect. 36
- Documented \changes command. 36
- \endglossary: Added to support \changes. 37
- \glossary@prologue: Added to support \changes. 38
- \GlossaryMin: Added to support \changes. 37
- \GlossaryParms: Added to support \changes. 38
- \GlossaryProlouge: Added to support \changes. 38
- \PrintChanges: Added to support \changes. 38
- \RecordChanges: Renames former \PrintChanges command. . . . 37
- \saved@macroname: Provided for sorting outside macro environment 37
- \theglossary: Added to support \changes. 37

v1.0p

- General: Use new error commands 1

v1.4?

- General: changes to the index env. 33
- use DEK's algorithm and implement a twocols env. 33

v1.4r

- General: twocols env. placed into separate file 33