The doc and shortvrb Packages*

Frank Mittelbach†‡§

Printed May 4, 2025

This file is maintained by the LaTeX Project team.
Bug reports can be opened (category latex) at
https://latex-project.org/bugs.html.

Abstract

Roughly 30 years ago (version 1.0 was dated 1988/05/05) I wrote the first version of the doc package, a package to provide code documentation for TeX code. Since then it has been used all over the place to document the LaTeX kernel and most of the packages that are nowadays available. The core code of version 2 (which is the current version) exists since 1998, i.e., for 20 years.

If I would restart from scratch I would do a lot of things differently these days and in fact several other people have tried to come up with better solutions. However, as the saying goes, a bad standard is better than none, doc has prevailed and changing it now in incompatible ways is probably not really helpful.

So this is version 3 of the package with some smaller extensions that are upwards compatible but hopefully serve well. Most important modifications are the integration of the hypdoc package to enable links within the document (in particular from the index) if so desired. Also integrated are the ideas from the DoX package by Didier Verna (although I offer a different interface that imho fits better with the rest of doc's interfaces). Finally I updated a few odds and ends.

*This file has version number v3.0q dated 2024/12/25.

†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).

§Version 3 now integrates code from Didier Verna's DoX package and some of his documentation was reused (a.k.a. stolen).

Contents

1 Introduction

This is a new version of the `doc` package, written roughly 30 years after the initial release. As the package has been used for so long (and largely unchanged) it is absolutely important to preserve existing interfaces, even if we can agree that they could have been done better.

So this is a light-weight change, basically adding hyperlink support and adding a way to provide generally `doc` elements (not just macros and environments) and try to do this properly (which wasn't the case for environments either in the past). The ideas for this have been stolen from the `DoX` package by Didier Verna even though I didn't keep his interfaces.

Most of the documentation below is from the earlier release which accounts for some inconsistencies in presentation, mea culpa.

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 usually 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 13 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
4 \usepackage[T1]{fontenc}
```

```
\usepackage{xspace}

\OnlyDescription

\EnableCrossrefs
 %\DisableCrossrefs     % Say \DisableCrossrefs if index is ready
\CodelineIndex
\RecordChanges          % Gather update information
\SetupDoc{reportchangedates}
 %\OnlyDescription      % comment out for implementation details
\setlength\hfuzz{15pt}  % don't show so many
\hbadness=7000          % over- and underfull box warnings
\begin{document}
   \DocInput{doc.dtx}
\end{document}
```
⟨/driver⟩

2.2 Package options

New in v3

Starting with version 3 the `doc` package now offers a small number of package options to modify its overall behavior. These are:

`hyperref`, `nohyperref` Boolean (default `true`). Load the hyperref package and make index references to code lines and pages and other items clickable links. `nohyperref` is the complementary key.

`multicol`, `nomulticol` Boolean (default `true`). Load the multicol package for use in typesetting the index and the list of changes. `nomulticol` is the complementary key.

`debugshow` Boolean (default `false`). Provide various tracing information at the terminal and in the transcript file. In particular show which elements are indexed.

`noindex` Boolean (default `false`). If set, all automatic indexing is suppressed. This option can also be used on individual elements as described below.

`noprint` Boolean (default `false`). If set, then printing of element names in the margin will be suppressed. This option can also be used on individual elements as described below.

`reportchangedates` Boolean (default `false`). If set, then change entries list the date after the version number in the change log.

`\SetupDoc` Instead of providing options to the `doc` package you can call `\SetupDoc` and provide them there. This allows, for example, to change default values in case `doc` was already loaded earlier.

2.3 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 characters `^^A` and `^^X` are both defined as a comment character later on.[1] 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` (*env.*) 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*` (*env.*) Instead of the `macrocode` environment one can also use the `macrocode*` environment which produces the same results except that spaces are printed as ␣ characters.

2.4 Describing the usage of macros and environments

`\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.

As the argument to `\DescribeMacro` is a command name, many people got used to using the (incorrect) short form, i.e., omitting the braces around the argument as in `\DescribeMacro\foo`. This does work as long as the macro name consists only of "letters". However, if the name contains special characters that are normally not of type "letter" (such as `@`, or in case of `expl3` `_` and `:`) this will fail dramatically. `\DescribeMacro` would then receive only a partial command name (up to the first "non-letter") e.g., `\DescribeMacro\foo@bar` would be equivalent to `\DescribeMacro{\foo} @bar` and you can guess that this can resulting in both incorrect output and possibly low-level error messages.

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

New in v3 Starting with version 3 the `\Describe...` commands accept an optional ar-

[1] In version 2 it was only `^^A`, but many keyboards combine `^` and `A` and automatically turn it into "Ä"; so `^^X` was added as an alternative in version 3.

gument in which you can specify either `noindex` or `noprint` to suppress indexing or printing for that particular instance. Using both would be possible too, but pointless as then the commands wouldn't do anything any more.

2.5 Describing the definition of macros and environments

`macro` (*env.*)

To describe the definition of a (new) macro we use the `macro` environment. It has one argument: the name of the new macro.[2] 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.

New in v3

In fact it is now allowed to specify several macros in the argument, separated by commas. This is a short form for starting several `macro` environments in direct succession. Of course, you should then have also only one matching `\end{macro}`.

`\MacrocodeTopsep` (*skip*)
`\MacroTopsep` (*skip*)
`\MacroIndent` (*dimen*)
`\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}` (if you change it later you might see strange typesetting effects if you are unlucky).

`\MacroFont` does not alter the font of `\verb` or `\verb*` because it is often used to make the font size of the code displays smaller, which would look odd if used within a paragraph. If you decide to use a different font family and want to use the same family with `\verb` you need to alter the font setup for `\ttfamily` in addition to `\MacroFont`.

`environment` (*env.*)

For documenting the definition of environments one can use the environment `environment` which works like the `macro` environment, except that it expects an ⟨*env-name*⟩ (without a backslash) as its argument and internally provides different index entries suitable for environments. Nowadays you can alternatively specify a comma-separated list of environments.

New in v3

Starting with version 3 these environments accept an optional argument in which you can specify `noindex` or `noprint` or both to suppress indexing or printing for that particular instance. If any such setting is made on the environment level it overwrites whatever default was given when the `doc` element was defined or when the package was loaded.

2.6 Formatting names in the margin

`\PrintDescribeMacro`
`\PrintDescribeEnv`
`\PrintMacroName`
`\PrintEnvName`

As mentioned earlier, some macros and environment print their arguments in

[2] 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.

the margin. The actual formatting is done by four macros which are user definable.[3] They are named `\PrintDescribeMacro` and `\PrintDescribeEnv` (defining how `\DescribeMacro` and `\DescribeEnv` behave) and `\PrintMacroName` and `\PrintEnvName` (called by the `macro` and `environment` environments, respectively).

2.7 Providing further documentation items

New in v3

Out of the box the `doc` package offers the above commands and environments to document macros and environments. With version 3 this has now been extended in a generic fashion so that you can easily provide your own items, such as counters, length register, options etc.

`\NewDocElement` The general syntax for providing a new `doc` element is

`\NewDocElement[`⟨*options*⟩`]{`⟨*element-name*⟩`}{`⟨*env-name*⟩`}`

By convention the ⟨*element-name*⟩ has the first letter uppercased as in `Env` or `Macro`.

Such a declaration will define for you

- the command `\Describe`⟨*element-name*⟩ which has the syntax

  `\Describe`⟨*element-name*⟩`[`⟨*options*⟩`]{`⟨*element*⟩`}`

- the environment ⟨*env-name*⟩ which has the syntax

  `\begin{`⟨*env-name*⟩`}[`⟨*options*⟩`]{`⟨*element*⟩`}`

- the display command `\PrintDescribe`⟨*element-name*⟩ with the syntax

  `\PrintDescribe`⟨*element-name*⟩`{`⟨*element*⟩`}`

- and the `\Print`⟨*element-name*⟩`Name` display command for the environment.

If any of the commands or the environment is already defined (which especially with the ⟨*env-name*⟩ is a danger) then you will receive an error telling you so.

`\RenewDocElement` If you want to modify an existing `doc` element use `\RenewDocElement` instead.

For example, the already provided "`Env`" `doc` element could have been defined simply by making the declaration `\NewDocElement{Env}{environment}` though that's not quite what has been done, as we will see later.

`\ProvideDocElement` This declaration does nothing when the doc element is already declared, otherwise it works like `\NewDocElement`. It can be useful if you have many documentation files that you may want to process individually as well as together.

The ⟨*options*⟩ are keyword/value and define further details on how that `doc` element should behave. They are:

`macrolike` Boolean (default `false`). Does this `doc` element starts with a backslash?

`envlike` Boolean. Complementary option to `macrolike`.

[3] 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 redefine them accept their argument and do nothing with it.

`toplevel` Boolean (default `true`). Should all a top-level index entry be made? If set to `false` then either no index entries are produced or only grouped index entries (see `idxgroup` for details).

`notoplevel` Boolean. Complementary option to `toplevel`.

`idxtype` String (default ⟨*env-name*⟩). What to put (in parentheses if non-empty) at the end of a top-level index entry.

`printtype` String (default ⟨*env-name*⟩). What to put (in parentheses if non-empty) after an element name in the margin.

`idxgroup` String (default ⟨*env-name*⟩`s`). Name of the top-level index entry if entries are grouped. They are only grouped if this option is non-empty.

`noindex` Boolean (default `false`). If set this will suppress indexing for elements of this type. This setting overwrite any global setting of `noindex`.

`noprint` Boolean (default `false`). If set this will suppress printing the element name in the margin. This setting overwrite any global setting of `noprint`.

As usual giving a boolean option without a value sets it to `true`.

2.8 Displaying sample code verbatim

`verbatim` (*env.*)
`verbatim*` (*env.*)
`\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.[4] 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` (§2.5).

2.9 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 `\catcode`s. 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!).

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

2.10 Cross-referencing all macros used

\DisableCrossrefs
\EnableCrossrefs

As already mentioned, every 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[5] 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` command) is controlled by using or omitting the following declarations in the driver file preamble; if

\PageIndex
\CodelineIndex

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` and possibly further `\Describe...` commands refer to a page number but those produced by the `macro` environment (or other `doc` element environments) refer to the code lines, which will be numbered

\theCodelineNo

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.

\CodelineNumbered

When you don't wish to get an index but want your code lines numbered use `\CodelineNumbered` instead of `\CodelineIndex`. This prevents the generation of an unnecessary `.idx` file.

2.11 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).

[5] This comment was written about 30 years ago. TEX is still considerably slower but while it took minutes to process a large document (such as the LATEX kernel documentation) it takes seconds or less these days. Thus `\DisableCrossrefs` isn't really that necessary these days.

[6] Actually, `\EnableCrossrefs` changes things more drastically; any following call to `\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.

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

\actualchar The \actualchar is used to separate the 'key' and the actual index entry. The \quotechar \quotechar is used before a special index program character to suppress its special \encapchar 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 \levelchar 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.

TODO: *describe old \SpecialMainIndex and \SpecialUsageIndex*

\SpecialMainMacroIndex To produce a main index entry for a macro the \SpecialMainMacroIndex \SpecialMainEnvIndex 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]

\SpecialMacroIndex To index the usage of a macro or an environment \SpecialMacroIndex and \SpecialEnvIndex \SpecialEnvIndex may be used.

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

New in v3 If further code elements are declared with \NewDocElement{⟨*name*⟩}... then this sets up additional indexing commands, e.g., \SpecialMain⟨*name*⟩Index.

\SpecialIndex The macrocode environment is automatically indexing macros (normally by code line number). You can (with care) also do this manually by \SpecialIndex. However, note that if \CodelineIndex is used this will generate an entry referring to the last code line which is usually not what you want. It does, however, make some sense if you always refer to pages only, i.e., if you use \PageIndex.

\SpecialShortIndex For single character macros, e.g., \{, doesn't always work correctly. For this

New in v3 reason there is now also a special variant the can produce correct index entries for them.

\SortIndex Additionally a \SortIndex command is provided. It takes two arguments—the sort key and the actual index entry.

\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 \+) then that caused a problem because you ended up with an index entry containing \verb+\++ which will be typeset as '\+' and not

New in v3 as '\+'. In version 3 this is now automatically taken care of (with the help of the \SpecialShortIndex command).

* 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>Special macros for \*}
```

assuming that > is the \levelchar used by the index processor.

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

You can't use \levelchar in this situation because if \index is directly used in the document then it's argument is written out fully verbatim. However, if you define your own index commands, expansion will happen on the way to the .idx file; and in that case you can use \levelchar—this is what the doc macros do. This then allows to change the indexing syntax easily, e.g.,

```
\newcommand\ltxconcept[2]{\index{#1\levelchar#2}}
...
\ltxconcept{index entries}{Special macros for \*}
```

2.12 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 favorite 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 \MaybeStop macro, in which case a \Finale command should appear at the end of your file.

theindex (*env.*) IndexColumns (*counter*) 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 (*dimen*) 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 encap-

\code sulated by the \usage macro (which produces *italics*). \code encapsulates page

or code line numbers in entries generated by parsing the code inside `macrocode` environments. As usual these commands are user definable.

2.13 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`	= 18.63434pt
`\marginparpush`	= 0.0pt	`\MacrocodeTopsep`	= 3.0pt plus 1.2pt minus 1.0pt
`\tolerance`	= 1000		

2.14 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`.

2.15 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` `\MaybeStop` `\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

New in v3

this work you have to place the command `\MaybeStop` at a suitable point in your file. This macro[11] 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 `\MaybeStop` 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`

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

`\AlsoImplementation`

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

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

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]

`\IndexInput`

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`

```
\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 glossary entry for the

[11] For about 30 years this macro was called `\StopEventually` which was due to a "false friend" misunderstanding. In the German language the word "eventuell" mean roughly "perhaps" which isn't quite the same as "eventually". But given that this is now used for so long and all over the place we can't drop the old name. So it is still there to allow processing all the existing documentation.

[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.

⟨*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.

Note that in the history listing, the entry is shown with the page number that corresponds to its place in the source, e.g., general changes put at the very beginning of the file will show up with page number "1", change entries placed elsewhere might have different numbers (not necessarily always very useful unless you are careful).

`\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 `\MaybeStop` 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`.

`\GlossaryMin` (*dimen*) `\GlossaryPrologue` `\GlossaryParms` `GlossaryColumns` (*counter*)

The `\GlossaryMin`, `\GlossaryPrologue` and `\GlossaryParms` macros and the counter `GlossaryColumns` are analogous to the `\Index`... versions. (The LaTeX 'glossary' mechanism is used for the change entries.)

`\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 recognized 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` (*counter*)

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.

3 Examples and basic usage summary

3.1 Basic usage summary

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

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

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

3.2 Examples

The default setup includes definitions for the `doc` elements "macro" and "environment". They correspond to the following declarations:

```
\NewDocElement[macrolike = true ,
               idxtype   = ,
               idxgroup  = ,
               printtype =
              ]{Macro}{macro}

\NewDocElement[macrolike = false ,
               idxtype   = env.  ,
               idxgroup  = environments ,
               printtype = \textit{env.}
              ]{Env}{environment}
```

To showcase the new features of `doc` version 3 to some extend, the current documentation is done by redefining these declarations and also adding a few additional declarations on top.

For any internal command we document we use `Macro` and put all of them under the heading "LaTeX commands" (note the use of `\actualchar`):

```
\RenewDocElement[macrolike = true ,
```

```
                toplevel  = false,
                idxtype   = ,
                idxgroup  = LaTeX commands\actualchar\LaTeX{} commands ,
                printtype =
               ]{Macro}{macro}
```

We only have package environments so we use `Env` for those and group them as well:

```
\RenewDocElement[macrolike = false ,
                toplevel  = false,
                idxtype   = env.  ,
                idxgroup  = Package environments,
                printtype = \textit{env.}
               ]{Env}{environment}
```

All the interface commands are also grouped together under the label "Package commands", we use `InterfaceMacro` for them:

```
\NewDocElement[macrolike = true ,
                toplevel  = false,
                idxtype   = ,
                idxgroup  = Package commands,
                printtype =
               ]{InterfaceMacro}{imacro}
```

And since we also have a few obsolete interfaces we add yet another category:

```
\NewDocElement[macrolike = true ,
                toplevel  = false,
                idxtype   = ,
                idxgroup  = Package commands (obsolete),
                printtype =
               ]{ObsoleteInterfaceMacro}{omacro}
```

Another type of category are the package keys:

```
\NewDocElement[macrolike = false ,
                toplevel  = false,
                idxtype   = key  ,
                idxgroup  = Package keys ,
                printtype = \textit{key}
               ]{Key}{key}
```

Finally we have TEX counters (with a backslash in front) and LATEX counters (no backslash) and the two types of LATEX length registers:

```
\NewDocElement[macrolike = true ,
                toplevel  = false,
                idxtype   =  counter  ,
                idxgroup  = TeX counters\actualchar \protect\TeX{} counters ,
                printtype = \textit{counter}
               ]{TeXCounter}{tcounter}

\NewDocElement[macrolike = false ,
                toplevel  = false,
                idxtype   =  counter  ,
```

```
                idxgroup  = LaTeX counters\actualchar \LaTeX{} counters ,
                printtype = \textit{counter}
               ]{LaTeXCounter}{lcounter}

\NewDocElement[macrolike = true ,
                toplevel  = false,
                idxtype   =  skip  ,
                idxgroup  = LaTeX length\actualchar \LaTeX{} length (skip) ,
                printtype = \textit{skip}
               ]{LaTeXSkip}{lskip}

\NewDocElement[macrolike = true ,
                toplevel  = false,
                idxtype   =  dimen  ,
                idxgroup  = LaTeX length\actualchar \LaTeX{} length (dimen) ,
                printtype = \textit{dimen}
               ]{LaTeXDimen}{ldimen}
```

And we modify the appearance of the index: just 2 columns not 3 and all the code-line entries get prefixed with an "ℓ" (for line) so that they can easily be distinguished from page index entries.

```
\renewcommand\code[1]{\mbox{$\ell$-#1}}
\renewcommand\main[1]{\underline{\mbox{$\ell$-#1}}}
\setcounter{IndexColumns}{2}
```

4 Incompatibilities between version 2 and 3

The basic approach when developing version 3 was to provide a very high level of compatibility with version 2 so that nearly all older documents should work out of the box without the need for any adjustments.

But as with any change there are situations where that change can result in some sort of incompatibility, e.g., if a newly introduce command name was already been defined in the user document then there will be a conflict that is nearly impossible to avoid 100%.

As mentioned earlier, `doc` now supports options on several commands and environments and as a result it is necessary to use braces around the argument for `\DescribeMaro` if the "macro to be described" uses private letters such as `@` or `_` as part of its name. That was always the official syntax but in the past you could get away with leaving out the braces more often than you can now.

The old `doc` documentation also claimed that redefinitions of things like `\PrintDescribeMacro` could be done before loading the package (and not only afterwards) and `doc` would in that case not change those commands. As the setup mechanisms are now much more powerful and general such an approach is not really good. So with `doc` version 3 modifications have to be done after the `doc` package got loaded and the last modification will always win.

I'm temped to drop compatibility with LaTeX 2.09 (but so far I have left it in).

In the past it was possible to use macros declared with `\outer` in the argument of `\begin{macro}` or `\DoNotIndex` even though `\outer` is not a concept supported in LaTeX. This is no longer possible. More exactly, it is no longer possible to

prevent them from being indexed (as \DoNotIndex can't be used), but you can pass them to the macro environment as follows:

```
\begin{macro}[outer]{\foo}
```

if \foo is a macro declared with \outer. The technical reason for this change is that in the past various other commands, such as \{ or \} did not work properly in these arguments when they where passed as "strings" and not as single macro tokens. But by switching to macro tokens we can't have \outer macros because their feature is to be not allowed in arguments. So what happens above when you use [outer] is that the argument is read as a string with four character tokens so that it is not recognized as being \outer.

5 Old interfaces no longer really needed

Thirty years is a long time in the life of computer programs, so there are a good number of interfaces within doc that are really only of historical interest (or when processing equally old sources). We list them here, but in general we suggest that for new documentation they should not be used.

5.1 makeindex bugs

\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 with and without the bug. If you really still 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.

5.2 File transmission issues

In the early days of the Internet file transmission issues have been a serious problem. There was a famous gateway in Rochester, UK that handled the traffic from the European continent to the UK and that consisted of two IBM machines running with different codepages (that had non-reversible differences). As a result "strange" TEX characters got replaced with something else with the result that the files became unusable.

To guard against this problem (or rather to detect it if something got broken in transfer I added code to doc to check a static character table and also to have a very simple checksum feature (counting backslashes).

These days the \CheckSum is of little value (and a lot of pain for the developer) and character scrambling doesn't happen any more so the \CharacterTable is essentially useless. Thus neither should be used in new developments.

\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         \~}
%%
```

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 add

```
%    \CheckSum{0}
```

near the beginning of the file and use the `\MaybeStop` (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 if you put in anything other than zero but guessed wrong (this time telling you both the correct and the incorrect one). Then you go to the top of your file again and change the line to the right number, i.e., line

```
%    \CheckSum{⟨number⟩}
```

and that's all.

While `\CharacterTable` and `\CheckSum` have been important features in the early days of the public internet when `doc` was written as the mail gateways back then were rather unreliable and often mangled files they are these days more a nuisance than any help. They are therefore now fully optional and no longer recommended for use with new files.

---

6 Introduction to previous releases

Original 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

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

the features of `newdoc`.

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.[15] 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.[16] 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.

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.

Preface to version 1.7 (from around 1992)

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

[15] 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.

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

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 recognized 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 and that added after version 1.5k of `doc`. 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. 18);
- 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 `docstrip`ped `.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.

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` user will be grateful for.

7 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.

```
21 ⟨∗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.

```
22 \@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 two alternative comment signs. Because all sensible signs are already occupied, we will choose some that can only be entered indirectly:

```
23 \catcode`\^^A=14
24 \catcode`\^^X=14
```

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

```
\AtBeginDocument{\catcode`\^^A=14\relax\catcode`\^^X=14\relax}
```

7.1 Keys supported by `doc`

In the past this used kvoptions but this will be replaced by using `l3keys` at some point in the future. Right now this is only a lightweight shift—the code could and should be altered further.

TODO: *cleanup replacement of kvoptions*

Some keys are available as options for use in `\usepackage` some are for the generated item API's:

```
\DeclareKeys
 {
   noprint           .if    = {doc@noprint},
   noindex           .if    = {doc@noindex},
   hyperref          .if    = {doc@hyperref},
   nohyperref        .ifnot = {doc@hyperref},
   multicol          .if    = {doc@multicol},
   nomulticol        .ifnot = {doc@multicol},
   debugshow         .if    = {doc@debugshow},
   reportchangedates .if    = {doc@reportchangedates},
   toplevel          .if    = {doc@toplevel},
   notoplevel        .ifnot = {doc@toplevel},
   macrolike         .if    = {doc@macrolike},
   envlike           .ifnot = {doc@macrolike},
   idxtype           .store = \doc@idxtype,
   idxgroup          .store = \doc@idxgroup,
   printtype         .store = \doc@printtype,
   outer             .if    = {doc@outer},
 }
```

Setting these options to true initially.

```
\doc@hyperreftrue
\doc@multicoltrue
\doc@topleveltrue
```

7.2 Processing the package keys

```
\ProcessKeyOptions
```

`\ifscan@allowed` `\scan@allowedtrue` `\scan@allowedfalse`

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

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

`\SetupDoc` We need to save the default value for some options because `doc` elements can locally set them.

TODO: *Use 2e interface for* `\keys_set:nn` *when available*

```
\def\SetupDoc#1{%
  \csname keys_set:nn\endcsname{doc}{#1}%
    \edef\doc@noprintdefault{\ifdoc@noprint true\else false\fi}%
  \ifdoc@noindex
```

If we do not index by default then we should also turn off `\scan@allowed`.

```
    \def\doc@noindexdefault{true}%
    \scan@allowedfalse
  \else
    \def\doc@noindexdefault{false}%
  \fi
}
\SetupDoc{}              % just save the default values
```

7.3 Macros surrounding the 'definition parts'

`macrocode` (*env.*) 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[]%
```

The `\item` command sets the `\@labels` box but that box is never typeset (as `\everypar` that normally does this gets redefined later). That is normally not an issue, but produces a problem when typesetting in mixed directions, (e.g., in Japanese), so we explicitly clear it for that use case.

```
    \global\setbox\@labels\box\voidb@x
```

Additionally, everything should be set in `typewriter` font. Some people might prefer it somewhat differently; because of this the font choice is macro-driven.[17]

```
\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.

```
\obeylines
\@noligs
\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
```

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

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
                           \shapedefault
                   }%
  \else
    \def\MacroFont{\fontencoding\encodingdefault
                   \fontfamily\ttdefault
                   \fontseries\mddefault
                   \fontshape\shapedefault
                   \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 `docstrip`ped.

```
\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$_\varepsilon$ `<` 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
```

`StandardModuleDepth` (*counter*) 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` (*counter*) 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
                    \mathcode`\-="702D \mathcode`\+="702B
                    \mathcode`\:="703A \mathcode`\=="703D }
```

`\MacrocodeTopsep` (*skip*) In the code above, we have used two registers. Therefore we have to allocate them.
`\MacroIndent` (*dimen*) The default values might be overwritten with the help of the `\DocstyleParms` macro.

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

`macrocode*` (*env.*) `\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)[18] 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*}`.

```
\@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*`.

```
\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.

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

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

```
\catcode`\{=12␣\catcode`\}=12
\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.

```
|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.

```
|catcode`| =12
|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.

```
|endgroup
```

[18] 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.

7.4 Macros for the 'documentation parts'

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}
\setlength\marginparsep{\labelsep}
```

`\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` (*env.*) `verbatim*` (*env.*) 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
              \@setupverbvisiblespace
              \MacroFont \@vobeyspaces \@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\relax
  \if@minipage\else\vskip\parskip\fi
  \leftskip\@totalleftmargin\rightskip\z@
  \parindent\z@\parfillskip\@flushglue\parskip\z@
  \language\l@nohyphenation
  \@@par
  \@tempswafalse
```

`\@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
      \leavevmode \null \@@par\penalty\interlinepenalty
    \else
      \@tempswatrue
      \ifhmode\@@par\penalty\interlinepenalty\fi
    \fi
```

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 `""`).

```
  \let\do\@makeother \dospecials
  \obeylines \verbatim@font \@noligs
  \everypar \expandafter{\the\everypar \unpenalty}%
}
```

`\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}}
```

In the early versions of the package it also redefined `\verb` because that didn't include the useful test for "newline" in the verbatim text. This is nowadays part of LATEX so we do not redefine it any longer (the original code is still kept in the file after `\endinput` to keep the long history intact).

`\macro@cnt` (*counter*) 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
```

`\MacroTopsep` (*skip*) Here is the default value for the `\MacroTopsep` parameter used above.

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

7.5 Formatting the margin

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

7.6 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 `\specialMainMacroIndex`, 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 \catcodes 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 \catcodes).

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\\}
```

7.7 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 5.2 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}
```

\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 use `\string` to normalize it the same way it is done in `\DoNotIndex`, i.e. everything ends up catcode 12 except for the space character.

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

Any indexing is then delegated to `\maybe@index@short@macro`. Depending on the actual character seen, this macro has to do different things, which is why we keep it separate from `\maybe@index@macro` to avoid the special tests in the more common case of a multi-letter macro name.

```
    \maybe@index@short@macro\macro@namepart
```

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 }
```

`\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}
```

`\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.[19]

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

7.8 The index exclude list[20]

The following part of the code is a new implementation using the LaTeX3 programming layer as the constructs and types provided therein are making programming much easier. Over time I will probably replace the rest of that doc code too.

```
\ExplSyntaxOn
```

\l__doc_donotindex_seq

Local sequence that holds names (as strings) of commands that should not be indexed. Within a doc element environment that element is placed into the sequence so that it isn't unnecessarily index within that part of the code. As the sequence is local it will revert this setting at the end of the environment so that the command is indexed elsewhere (unless it is generally disabled from indexing).

\g__doc_idxtype_prop

Global property list that holds for all commands that are special doc elements the type of the element. The key is the command name without backslash and the value is doc element type identifier, e.g., \texttt{Length} for length registers if that type has been set up. doc only indexes commands, that is things starting with an escape character, i.e., a backslash (by default). doc elements that do not start with an escape character, e.g., environments are not identified when parsing code so that aren't indexed automatically inside. Thus for them there is no point in keeping them in that property list.

\doc_dont_index:n
\DoNotIndex

Takes a clist of commands (with backslash) as input and exclude all of them from the index. User facing we make this available as \DoNotIndex.

\ShowIndexingState

Displays the current list of the exclude index list in a fairly low-level form.

\RecordIndexType

This command takes two arguments: a command (with escape char) and its type (i.e., first mandatory argument of a \NewDocElement declaration). If #1 should not be included from the index, then the data is used to record that this command is of this type. The information is then used to generate appropriate index entries. Obviously, index entries generated earlier will be listing the wrong type. Therefore this information is also placed into the .aux file so that it will be available at the beginning of further runs.

This command is internally executed as part of any doc element environment so it only needs to be explicitly given if for some reason a command with a special type has no corresponding environment.

\l__doc_donotindex_seq
\g__doc_idxtype_prop

Declarations.

```
\seq_new:N  \l__doc_donotindex_seq
\prop_new:N \g__doc_idxtype_prop
```

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

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

`\__doc_trace:x` A helper for tracing...

```
\cs_new:Npn\__doc_trace:x {
  \legacy_if:nTF{ doc@debugshow }{ \iow_term:x } { \use_none:n }
}
```

`\doc_dont_index:n` `\__doc_dont_index:n` `\__doc_dont_index_aux:n` Parses the argument a clist of commands with `\MakePrivateLetters` in force (so that special characters are recognized as being part of command names) and puts each command without is backslash as a string into the sequence `\l__doc_donotindex_seq`.

```
\cs_new:Npn \doc_dont_index:n {
  \group_begin:
    \MakePrivateLetters
    \__doc_dont_index:n
}
\cs_new:Npn \__doc_dont_index:n #1 {
  \group_end:
  \__doc_trace:x{Disable~ indexing~ for~ '\tl_to_str:n{#1}' }
```

Adding the commands to the `\l__doc_donotindex_seq` sequence is done by mapping the function `\__doc_dont_index_aux:n` on each element in the clist.

```
  \clist_map_function:nN {#1} \__doc_dont_index_aux:n
}
```

We record each command by using its name as a string. This means more tokens in the sequence but it allows to compare names not "action" which is important as different commands may have the same meaning (e.g., they may not be defined at all),

```
\cs_new:Npn \__doc_dont_index_aux:n #1 {
  \seq_put_right:Nx \l__doc_donotindex_seq {\expandafter\@gobble \string#1}
}
```

`\DoNotIndex` The document-level interface

```
\cs_set_eq:NN \DoNotIndex \doc_dont_index:n
```

`\ShowIndexingState` Some tracing information that may be helpful.

```
\def \ShowIndexingState {
  \__doc_trace:x{Show~ doc~ indexing~ state:}
  \seq_show:N  \l__doc_donotindex_seq
  \prop_show:N \g__doc_idxtype_prop
}
```

`\__doc_idxtype_put:Nn` **TODO:** *Change name of interface command!*

`\RecordIndexType` `\RecordIndexTypeAux` This is the internal form for `\RecordIndexType`. The first argument is turned into a string and the rest of the processing is then done by `\__doc_idxtype_put:nn`

```
\cs_new:Npn \__doc_idxtype_put:Nn #1#2 {
  \exp_args:Nx \__doc_idxtype_put:nn { \cs_to_str:N #1 }{#2}
```

We also make an entry in the `.aux` file so that this declaration becomes immediately available in the next run. However, for this we aren't reusing `\__doc_idxtype_put:N` (a.k.a. `\RecordIndexType`) since that would result in doubling such lines each time the document is run. Instead we use `\RecordIndexTypeAux` which is only updating the data structures without writing to the `.aux` file.

```
  \protected@write\@auxout{}
     {\string\RecordIndexTypeAux {\string#1 }{#2} }
}
```

When we execute this code from the `.aux` we better not generate a new line in the `.aux`. Otherwise those would cumulate over time.

```
\cs_new:Npn \RecordIndexTypeAux #1#2 {
  \exp_args:Nx \__doc_idxtype_put:nn { \cs_to_str:N #1 }{#2}
}
```

Similarly, when the `.aux` is read at the end of the run we should disable that command to avoid unnecessary processing.

```
\AtEndDocument{
  \cs_set_eq:NN \RecordIndexTypeAux \use_none:nn
}
```

Finally, we provide the user-level interface

```
\cs_set_eq:NN \RecordIndexType \__doc_idxtype_put:Nn
```

`\__doc_idxtype_put_scan:nn` When we want to record an index type for a scanned name we can't turn that into a csname and then call `\__doc_idxtype_put:Nn` because turning it into a csname may change the meaning of the name from "undefined" to `\relax`. Got bitten by that when processing the kernel sources containing `\@undefined` within the code: suddenly that wasn't undefined any longer. So here is another version that works only on characters as input. As we don't know whether or not they are proper strings already we first make sure that this is the case.

```
\cs_new:Npn \__doc_idxtype_put_scan:nn #1#2 {
  \exp_args:Nf \__doc_idxtype_put:nn { \tl_to_str:n {#1} }{#2}
```

In this case we also have to append a backslash when writing to the `.aux` file.

```
  \protected@write\@auxout{}
     {\string\RecordIndexTypeAux {\bslash #1 }{#2} }
}
```

`\__doc_idxtype_put_scan:on` And here is the one we really need since the characters are stored in some macro.

```
\cs_generate_variant:Nn \__doc_idxtype_put_scan:nn {o}
```

`\record@index@type@save` And here is the interface to the rest of the code:

```
\cs_set_eq:NN \record@index@type@save \__doc_idxtype_put_scan:on
```

`\__doc_idxtype_put:nn` This internal command takes two arguments: a command name as string (no backslash) and its type (i.e., first mandatory argument of a `\NewDocElement` declaration). If `#1` is not in `\l__doc_donotindex_seq` it will add this data to the property list `\g__doc_idxtype_prop` using `#1` as key and `#2` as its value. If the key already exist its value will be overwritten. If the command is later marked for exclusion from the index the property list setting remains unchanged but as long as no index is produced for the command it will not be consulted.

Note: the command assumes that `#1` is already in string form

```
\cs_new:Npn \__doc_idxtype_put:nn #1#2 {
```

No mystery here: if the command is not marked for exclusion from the index add the property. The extra `\tl_to_str:n` is a safety measure in case the input wasn't already in that form (should only be the case with broken input but ...)

```
  \exp_args:NNf
  \seq_if_in:NnTF \l__doc_donotindex_seq {\tl_to_str:n{#1}}
```

Some tracing info ...

```
    {
      \__doc_trace:x{Not~ recording~ index~ type~ for~ '\bslash #1' }
    }
    {
      \__doc_trace:x{Recording~ index~ type~ for~ '\bslash #1' ~ as~ #2 }
```

Stick the data into the property list:

```
      \prop_gput:Nnn \g__doc_idxtype_prop {#1}{#2}
    }
}
```

`\exp_args:co` A helper: construct a function and call it with its first argument expanded once:

```
\cs_new:Npn \exp_args:co #1#2
   { \cs:w #1 \exp_after:wN \cs_end:\exp_after:wN {#2} }
```

`\tl_to_str:o` Another helper: take some token list variable, expand it and turn it into a string.

```
\cs_generate_variant:Nn \tl_to_str:n {o}
```

`\maybe@index@macro`

This takes a macro name (without backslash) as parsed within a `macrocode` environment and checks if it should get indexed (i.e., is not on the exclude list) and if so how (i.e., gets it index type property and makes the right choice depending on that.

`\maybe@index@macro` `\__doc_maybe_index:o` We first make sure that the argument is really a string (so that we have a defined situation) and then pass it on to `\__doc_maybe_index_aux:nN` to do the work. The second argument defines the indexing operation `\SpecialIndex` for multi-letter macros and below `\SpecialShortIndex` for single character macros.

```
\cs_new:Npn \__doc_maybe_index:o #1 {
  \exp_args:Nf \__doc_maybe_index_aux:nN { \tl_to_str:o {#1} }
                                         \SpecialIndex
}
```

And here is what we call it in the older non-expl3 code:

```
\cs_set_eq:NN \maybe@index@macro \__doc_maybe_index:o
```

`\maybe@index@short@macro` `\__doc_maybe_index_short:o` Single character macros are handled similarly but there the indexing is done by `\SpecialShortIndex` and it is simpler because we know that the argument contains a string token not letters.

```
\cs_new:Npn \__doc_maybe_index_short:o #1 {
  \exp_args:No \__doc_maybe_index_aux:nN #1
                                         \SpecialShortIndex
}
\cs_set_eq:NN \maybe@index@short@macro \__doc_maybe_index_short:o
```

`\__doc_maybe_index_aux:nN` Take a string (representing a macro without backslash) and make the right choices with respect to indexing.

```
\cs_new:Npn \__doc_maybe_index_aux:nN #1#2 {
```

A bit of tracing:

```
  \__doc_trace:x{Searching~ for~ '\bslash #1'}
```

If the name is on the exclude list do nothing.

```
\seq_if_in:NnTF \l__doc_donotindex_seq {#1}
  {
   \__doc_trace:x{Not~ indexing~ '\bslash #1' }
  }
```

Otherwise check if this name has an index type property attached to it.

```
  {
   \prop_get:NnNTF \g__doc_idxtype_prop {#1} \l__doc_idxtype_tl
```

If so construct and execute `\Code`⟨*idxtype*⟩`Index`[21] which is done inside `\__doc_maybe_index_aux`

```
     {
      \exp_args:Ncno \__doc_maybe_index_aux:Nnn
            { Code \tl_use:N \l__doc_idxtype_tl Index }
            {code} {\bslash #1}
      }
```

Otherwise execute `\SpecialIndex` which is a short form for `\CodeMacroIndex{code}` or execute `\SpecialShortIndex` which deals with some special cases for single letter macros.

```
      {
        \__doc_trace:x{Indexing~ '\bslash #1'\space (\string #2)}
        \exp_args:No #2 {\bslash #1}
      }
  }
}
```

`\SpecialShortIndex` **TODO:** *to be documented; also needs cleaning up as it is a mix of old and new right now*

```
\cs_new:Npn \SpecialShortIndex #1 {
   \@SpecialIndexHelper@ #1\@nil
 \@bsphack
 \ifdoc@noindex \else
   \str_case_e:nnF {\@gtempa }
       {
         {\cs_to_str:N \^^M } {\def\reserved@a{ \string \space \actualchar }
                               \def\reserved@b { \space }
                               \let\reserved@c \@empty                          }
```

With the fix for `\␣` we now have to look for a real space to handle that command sequence.

```
         { ~ }                {\def\reserved@a{ \string \space \actualchar }
                               \def\reserved@b { \space }
                               \let\reserved@c \@empty                          }
         {\c_left_brace_str} { \def\reserved@a{ \bgroup \actualchar }
                               \def\reserved@b { \c_left_brace_str }
                               \def\reserved@c { \noexpand\iffalse
                                                 \c_right_brace_str
                                                 \noexpand\fi }                 }
         {\c_right_brace_str} { \def\reserved@a{ \egroup \actualchar
                                                 \noexpand\iffalse
                                                   \c_left_brace_str
```

[21] I guess this should really be an internal name not a user-level one.

```
                                                \noexpand\fi }
                                    \def\reserved@b { \c_right_brace_str }
                                    \let\reserved@c \@empty                          }
```

The case of `\verbatimchar` is tricky. We can't stick it into the normal `\verb` because we would then get something like `\verb+\++` which would comes out as "\+" instead of `\+`. So we use the `\verb` to only generate the backslash and then use `\texttt` on the `\verbatimchar` itself. However, that is not enough if we are unlucky and somebody (like Will :-)) has used something like `&` with a special catcode for the `\verbatimchar`. We therefore also apply `\string` to it when we read it back.

```
          {\verbatimchar}  { \def\reserved@a{ \quotechar\verbatimchar
                                             \actualchar }
                             \let\reserved@b \@empty
                             \def\reserved@c
                                 { \string\texttt{\string\string\verbatimchar} } }
        }
        { \def\reserved@a {\quotechar \@gtempa \actualchar }
          \def\reserved@b {\quotechar \@gtempa  }
          \let\reserved@c \@empty                              }
    \special@index {
    \reserved@a
    \string\verb
    \quotechar *\verbatimchar \quotechar \bslash
    \reserved@b
    \verbatimchar
    \reserved@c
    \encapchar code}
  \fi
  \@esphack
}
```

`\__doc_maybe_index_aux:Nnn` Execute the function passed on as first argument taking argument 2 and 3 as input.

```
\cs_new:Npn \__doc_maybe_index_aux:Nnn #1#2#3 {
```

We have to be a little careful: as that function name is constructed it may not actually exist (as constructions generate `\relax` in TEX in that case). In that case we raise an error, otherwise we execute (with a little bit of tracing info):

```
    \cs_if_exist:NTF #1
      {
        \__doc_trace:x{Indexing~ '#3'\space as~
                       \tl_use:N \l__doc_idxtype_tl }
        #1{#2}{#3}
      }
      {
        \PackageError{doc}{Doc~ element~
           '\tl_use:N \l__doc_idxtype_tl'~ unknown}%

          {When~ using~ '\string\RecordIndexType'~ the~ type~ must~
           be~ known~\MessageBreak
           to~ the~ system,~ i.e.,~ declared~ via~
           '\string\NewDocElement'\MessageBreak
           before~ it~ can~ be~ used~ in~ indexing.}
```

```
    }
}
```

Back to old style coding ...

```
\ExplSyntaxOff
```

7.9 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 favorite such program.

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{+}}{}
```

\@SpecialIndexHelper@ **TODO:** *doc or drop*

```
\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
```

\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{%
  \ifdoc@noindex\else
    \index{#1\actualchar#2}%
  \fi
}
```

\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% % to fool emacs highlighting
                  \quotechar*\verbatimchar
                  \quotechar\bslash{\verbatimchar\string\iffalse}\string\fi}}}{}

\@ifundefined{RightBraceIndex}{\def\RightBraceIndex{%
 \special@index{\egroup\actualchar\string\iffalse{\string\fi
           \string\verb% % to fool emacs highlighting
           \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% % to fool emacs highlighting
           \quotechar*\verbatimchar\quotechar\bslash
           \percentchar\percentchar\verbatimchar}}}
{\catcode`\%=12 \gdef\percentchar{%}}
```

\it@is@a

This macro is supposed to produce a correct \SortIndex entry for a given character. Since this character might be recognized 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% % to fool emacs highlighting
                           \quotechar*\verbatimchar
                           \quotechar\bslash\quotechar#1\verbatimchar}}
```

7.10 Redefining the index environment

\IndexMin (*dimen*)
IndexColumns (*counter*)

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 (*env.*)

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.

```
\ifdoc@multicol
  \RequirePackage{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
  \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 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\pfill{\unskip~%
           \leaders\hbox to.6em{\hss .\hss}\hfill
           \penalty500\strut\nobreak
           \leaders\hbox to.6em{\hss .\hss}\hfil
           ~\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[22] 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}}}{}
```

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

\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}}}{}
```

\code The \code macro is used to indicate index entries to code lines that aren't main entries. By default we do nothing special with them the usage of a macro.

```
\@ifundefined{code}{\def\code#1{#1}}{}
```

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

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

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⟩
```

7.11 Dealing with the change history[23]

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).

[23] 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.

`\changes` The output of the `\changes` command goes into the ⟨*Glossary_ File*⟩ and therefore uses the normal `\glossaryentry` commands.[24] 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%
```

If asked for we also show the date of in the change log (after the version).

```
                    \ifdoc@reportchangedates
                      \space -- #2\fi
                    \levelchar
```

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 precede it by `\generalname`.

```
                    \ifx\saved@macroname\@empty
```

Putting a `!` at the beginning of the entry hopefully moves this entry to the very beginning during sorting.

```
                       \quotechar!%
                       \actualchar
                       \generalname
                    \else

                       \saved@indexname
                       \actualchar
                       \string\verb% % to fool emacs highlighting
                       \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{}
```

[24] 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.

`\saved@indexname` The macroname being document without a backslash for the index (or the environment name which doesn't have one in the first place).

```
\def\saved@indexname{}
```

`\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` (*dimen*)
`GlossaryColumns` (*counter*) 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` (*env.*) The environment `theglossary` is defined in the same manner as the `theindex` environment.

```
\ifdoc@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 except that we make it sort of ragged right as it contains text that often doesn't break nicely in small columns.

```
\@ifundefined{GlossaryParms}{\let\GlossaryParms\IndexParms
  \expandafter\def\expandafter\GlossaryParms\expandafter{\GlossaryParms
     \rightskip 15pt plus 1fil
     \parfillskip -15pt plus -1fil\relax}
}{}
```

\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 \MaybeStop 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}
```

7.12 Bells and whistles

\MaybeStop
\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 \MaybeStop 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\MaybeStop##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: \MaybeStop 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.

```
661                \init@checksum
662                \@esphack}%
663          }
```

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

```
664 \AlsoImplementation
```

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

```
665 \def\OnlyDescription{\@bsphack\long\def\MaybeStop##1{%
```

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

```
666             ##1\endinput}\@esphack}
```

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

```
667 \let\Finale\relax
```

\StopEventually The old wrong name for \MaybeStop. We need to use \def (i.e., expansion) as \MaybeStop gets redefined once in a while.

```
668 \def\StopEventually{\MaybeStop}
```

\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 \hboxes. 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 \protected. The new implementation fixes this problem by defining \meta in a radically different way: we prevent hyphenation by defining a \language which has no patterns associated with it and use this to typeset the words within the angle brackets.

```
669 \ifx\l@nohyphenation\undefined
670   \newlanguage\l@nohyphenation
671 \fi
672 \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.

```
673      \ensuremath\langle
674      \ifmmode \expandafter \nfss@text \fi
675      {%
676       \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.

```
      \edef\meta@hyphen@restore
        {\hyphenchar\the\font\the\hyphenchar\font}%
      \hyphenchar\font\m@ne
      \language\l@nohyphenation
      #1\/%
      \meta@hyphen@restore
     }\ensuremath\rangle
}
```

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

```
\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.

```
\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*⟩.[25] 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*⟩.

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

`\@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{%
```

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

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% % to fool emacs highlighting
                          (*)}%
  \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% % to fool emacs highlighting
                          (*)}%
  \else
    \@shortvrbinfo{Deleted }{#1 as}%
                           {\@empty\verb% % to fool emacs
                                          % highlighting
                           (*)}%
    \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⟩
```

7.13 Providing a checksum and character table[26]

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

```
\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>\m@ne
```

We do nothing if the checksum in the file is negative (or not given as it is initialized with -1).

```
    \ifnum\check@sum=\z@
      \typeout{**********************************}%
```

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

```
      \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
  \fi
  \global\check@sum\m@ne}
```

`\check@sum` (*counter*)
`\bslash@cnt` (*counter*)

We need to define counters, `\bslash@cnt` for the number of backslashes counted and `\check@sum` for the value advertised by the file if any. A negative value means there is no checksum checking which is the default.

```
\newcount\check@sum           \check@sum  = \m@ne
\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.}
```

7.14 Attaching line numbers to code lines[27]

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
```

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

`\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
     \begingroup
        \set@display@protect
        \immediate\write\@indexfile
            {\string\indexentry{#1}%
             {\number\c@CodelineNo}}%
      \endgroup
    \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}
```

`CodelineNo` (*counter*) 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}
  {}
```

7.15 Layout Parameters for documenting package files

`\tolerance` People documenting package files would probably rather have things "sticking out" in overfull `\hbox`es 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;[28] particularly for something like

[28] 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 `\mathcode`s to work with any family assignments. For an example see the article about the new font selection scheme.

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

Clear out `\DocstyleParms` after use (or non-use).

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

`\PlainTeX` `\Web` There's even a PLAIN TEX and a WEB.

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

7.16 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}
```

7.17 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}
```

8 Integrating hypdoc

If the option hyperref is selected (which is the default), then we load the hypdoc package. We do that as late as possible so that we don't generate option clashes if it is also loaded in the preamble. That package currently changes more commands than it should (not knowing about their new definitions defined below) so we have to save and restore a few.

Midterm all this code in hypdoc should be directly included in doc. For now, while they are separate we have to do this juggling.

```
\AddToHook{begindocument/before}[doc/hyperref]{%
 \ifdoc@hyperref
```

Annoying to code around issue #22

```
  \expandafter\let\expandafter\doc@eoph@@k\csname doc.sty-h@@k\endcsname
```

We require the package without any option so if it was already loaded there is no option clash.

```
  \RequirePackage{hypdoc}
  \expandafter\let\csname doc.sty-h@@k\endcsname\doc@eoph@@k
```

The package adds new definitions for \special@index into \CodelineIndex and \PageIndex but since we might be loading it very late we are already past them (in the preamble). So we test the final state and do it here, if necessary.

```
  \ifx\special@index\@gobble  % do we write index entries at all?
  \else
    \ifcodeline@index
      \let\special@index\HD@codeline@wrindex
    \else
      \let\special@index\HD@page@wrindex
    \fi
  \fi
```

The amsmath documentation uses \env in headings and with hyperref enabled this causes trouble in bookmarks.

TODO: *fix elsewhere eventually*

```
  \AddToHook{class/amsdtx/after}{%
     \pdfstringdefDisableCommands{\let\env\@empty }}%
```

That package also adds extra code into \index entries but it doesn't know about all the stuff that doc does (now). So we need to provide us with two helpers that handle the \encapchar case in some entries.

```
  \def\doc@providetarget{\HD@target}%
  \def\doc@handleencap#1{\encapchar hdclindex{\the\c@HD@hypercount}{#1}}%
```

If that package is not loaded these helpers do little to nothing.

```
 \else
  \let\doc@providetarget\@empty
  \def\doc@handleencap#1{\encapchar #1}%
```

We define the next commands just in case the user changed the option `hyperref` from `true` to `false` without removing the auxiliary files.

```
  \def\hdclindex#1#2{\ifx\@nil#2\@nil\else\csname #2\expandafter\endcsname\fi}%
  \def\hdpindex   #1{\ifx\@nil#1\@nil\else\csname #1\expandafter\endcsname\fi}%
\fi
}
```

9 Integrating the DoX package code

The code in this section is largely taken over from the `DoX` package by Didier with only minor modifications (so far). This means it is a bit back and forth and both the code and the documentation need further updates.

9.1 DoX environments

`\@doc@env` `\@doc@env@`

TODO: *original doc – fix*

`{`⟨*are-we-macrolike*⟩`}{`⟨*item*⟩`}{`⟨*indextype*⟩`}{`⟨*name*⟩`}`

In `doc.sty`, the `macro` and `environment` environments go through the `\m@cro@` macro which implements specific parts by testing a boolean condition as its first argument. This mechanism is not extensible, so I have to hack away a more generic version that would work for any new `dox` item, only which looks pretty much like the original one (with the addition of options management).

First step is to see if we have a comma-separated list of names in `#3` and if so we call the macro doing the work individually for each

```
\ExplSyntaxOn
\long\def\@doc@env#1#2#3{
```

The `\endgroup` here closes the scanning of names (using special catcodes).

```
  \endgroup
  \clist_map_inline:nn {#3} { \@doc@env@{#1}{#2}{##1} }
}

\ExplSyntaxOff
```

And here is the payload for each name from the given list:

```
\long\def\@doc@env@#1#2#3{%
  \topsep\MacroTopsep
  \trivlist
    \edef\saved@macroname{\string#3}%
```

Since version 2.1g, `doc` creates a `\saved@indexname` command which in used by `\changes`. We now support that as well. The expansion of this command depends on whether the documented item is macrolike or not, which we don't know here (it's only known by `\NewDocElement`). That's why we need one specific command generating `\saved@indexname` the right way for every single item. These

commands are named`\@Save`⟨*item*⟩`IndexName`; they are technically part of the generated API, only not meant for public use.

TODO: *above docu is no longer right (but code probably needs further changes anyway)*

`#1` is either `TT` (for true = macrolike) or `TF`. If true then we drop the first char from `\saved@macroname` and store the result in `\saved@indexname` and use the latter for sorting in the index.

```
    \if #1%
      \edef\saved@indexname{\expandafter\@gobble\saved@macroname}%
%
```

If the `doc` element described is macrolike but not a normal "macro" then its type should be recorded and this is the places where this happens. For macros (which should make up the bulk of these items) we don't do this and for anything else that looks from an indexing perspective like a macro we don't do that either to keep the list of exceptions small. That would be the case if the indexing command `\Code`⟨*doc-element*⟩`Index` is equivalent to `\CodeMacroIndex`.

```
      \expandafter\ifx
                  \csname Code#2Index\endcsname
                  \CodeMacroIndex
      \else
        \record@index@type@save
          {\saved@indexname}{#2}%
      \fi
    \else
      \let\saved@indexname\saved@macroname
    \fi
%
    \def\makelabel##1{\llap{##1}}%
    \if@inlabel
      \let\@tempa\@empty
      \count@\macro@cnt
      \loop\ifnum\count@>\z@
        \edef\@tempa{\@tempa\hbox{\strut}}\advance\count@\m@ne
      \repeat
      \edef\makelabel##1{\llap{\vtop to\baselineskip{\@tempa\hbox{##1}\vss}}}%
      \advance\macro@cnt\@ne
    \else
      \macro@cnt\@ne
    \fi
    \ifdoc@noprint
      \item
    \else
      \edef\@tempa{%
        \noexpand\item[%
```

The second notable modification to the original macro involves dynamically constructing the name of the print macro:

```
        \noexpand\doc@providetarget
        \noexpand\strut
        \noexpand\@nameuse{Print#2Name}{\saved@macroname}]}%
      \@tempa
    \fi
    \ifdoc@noindex\else
```

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

and the third one involves dynamically constructing the name of the index macro:

```
      \csname SpecialMain#2Index\expandafter\endcsname
        \expandafter{\saved@macroname}\nobreak
      \global\advance\c@CodelineNo\m@ne
    \fi
```

Suppress further `\index` entries when we are within a `macrolike` environment. There is no point doing that for non-`macrolike` environments are index entries are only generated for items starting with a backslash anyway.

TODO: *fix*

```
    \if#1\expandafter\DoNotIndex \expandafter {\saved@macroname}\fi
    \ignorespaces}
```

`\doc@env` {⟨*true-value*⟩}{⟨*item*⟩}[⟨*options*⟩]

Handle optional arguments and call `\@doc@env`. Because environments can be nested, we can't rely on grouping for getting options default values. Hence, we need to reset the options at every call.

TODO: *Use 2e interface for* `\keys_set:nn` *when available*

```
\def\doc@env#1#2[#3]{%
  \@nameuse{doc@noprint\doc@noprintdefault}%
  \@nameuse{doc@noindex\doc@noindexdefault}%
  \csname keys_set:nn\endcsname{doc}{#3}%
  \begingroup
    \ifdoc@outer
      \catcode`\\12
    \fi
    \MakePrivateLetters
    \@doc@env{#1}{#2}%
}
```

9.2 `doc` descriptions

`\@doc@describe` {⟨*item*⟩}{⟨*name*⟩}

```
\def\@doc@describe#1#2{%
    \ifdoc@noprint\else
      \marginpar{\raggedleft
```

The hyperref target has to be in horizontal mode (which is the case if it is after the `\strut`).

```
                \strut
                \doc@providetarget
                \@nameuse{PrintDescribe#1}{#2}}%
    \fi
    \ifdoc@noindex\else
      \@nameuse{Special#1Index}{#2}%
    \fi
  \@esphack
  \endgroup
  \ignorespaces}
```

\doc@describe {⟨*item*⟩}[⟨*options*⟩]

Handle optional arguments and call \@doc@describe.

TODO: *Use 2e interface for* \keys_set:nn *when available*

```
\def\doc@describe#1[#2]{%
  \leavevmode\@bsphack
  \csname keys_set:nn\endcsname{doc}{#2}%
  \@doc@describe{#1}}
```

9.3 API construction

\@temptokenb A scratch register (which may have been defined elsewhere)

```
\@ifundefined{temptokenb}{\newtoks\@temptokenb}{}
```

\doc@createspecialmainindex {⟨*item*⟩}{⟨*idxtype*⟩}{⟨*idxcat*⟩}

reatespecialmainmacrolikeindex {⟨*item*⟩}{⟨*idxtype*⟩}{⟨*idxcat*⟩}

TODO: *original doc – fix*

The "macrolike" version does something similar to doc's \SpecialIndex@ macro, but simplified. Let's just hope nobody will ever define \␣ or nonletter macros as macrolike doc elements...

```
\def\doc@createspecialindexes#1#2#3{%
  \@temptokena{\space (#2)}%
  \@temptokenb{#3:}%
  \@nameedef{SpecialMain#1Index}##1{%
    \noexpand\@bsphack
  \ifdoc@toplevel
    \noexpand\special@index{##1\noexpand\actualchar
    {\string\ttfamily\space##1}%
    \ifx\@nil#2\@nil\else \the\@temptokena \fi
    \noexpand\encapchar main}%
  \fi
  \ifx\@nil#3\@nil\else
    \noexpand\special@index{\the\@temptokenb\noexpand\levelchar
      ##1\noexpand\actualchar{\string\ttfamily\space##1}%
      \noexpand\encapchar main}%
  \fi
    \noexpand\@esphack}%
  \@nameedef{Special#1Index}##1{%
    \noexpand\@bsphack
  \ifdoc@toplevel
    \noexpand\doc@providetarget
    \noexpand\index{##1\noexpand\actualchar{\string\ttfamily\space##1}%
    \ifx\@nil#2\@nil\else \the\@temptokena \fi
          \noexpand\doc@handleencap{usage}}%
  \fi
  \ifx\@nil#3\@nil\else
    \noexpand\index{\the\@temptokenb\noexpand\levelchar
       ##1\noexpand\actualchar{\string\ttfamily\space##1}%
          \noexpand\doc@handleencap{usage}}%
  \fi
    \noexpand\@esphack}}
```

```
\def\doc@createspecialmacrolikeindexes#1#2#3{%
  \@temptokena{\space (#2)}%
  \@temptokenb{#3:}%
  \@nameedef{Code#1Index}##1##2{%
    \noexpand\@SpecialIndexHelper@##2\noexpand\@nil
    \noexpand\@bsphack
  \noexpand\ifdoc@noindex\noexpand\else
    \ifdoc@toplevel
      \noexpand\special@index{\noexpand\@gtempa\noexpand\actualchar
 \string\verb% % to fool emacs highlighting
 \noexpand\quotechar*\noexpand\verbatimchar
 \noexpand\bslash\noexpand\@gtempa\noexpand\verbatimchar
 \ifx\@nil#2\@nil\else \the\@temptokena \fi
 \noexpand\encapchar ##1}%
    \fi
    \ifx\@nil#3\@nil\else
      \noexpand\special@index{\the\@temptokenb\noexpand\levelchar
 \noexpand\@gtempa\noexpand\actualchar
 \string\verb% % to fool emacs highlighting
 \noexpand\quotechar*\noexpand\verbatimchar
 \noexpand\bslash\noexpand\@gtempa\noexpand\verbatimchar
 \noexpand\encapchar ##1}%
    \fi
  \noexpand\fi
    \noexpand\@esphack}%
  \@nameedef{SpecialMain#1Index}##1{%
    \expandafter\noexpand\csname Code#1Index\endcsname
        {main}{##1}}%
  \@nameedef{Special#1Index}##1{%
    \noexpand\@SpecialIndexHelper@##1\noexpand\@nil
    \noexpand\@bsphack
  \noexpand\ifdoc@noindex\noexpand\else
    \ifdoc@toplevel
      \noexpand\doc@providetarget
      \noexpand\index{\noexpand\@gtempa\noexpand\actualchar
 \string\verb% % to fool emacs highlighting
 \noexpand\quotechar*\noexpand\verbatimchar
 \noexpand\bslash\noexpand\@gtempa\noexpand\verbatimchar
 \ifx\@nil#2\@nil\else \the\@temptokena \fi
 \noexpand\doc@handleencap{usage}}%
    \fi
    \ifx\@nil#3\@nil\else
      \noexpand\index{\the\@temptokenb\noexpand\levelchar
 \noexpand\@gtempa\noexpand\actualchar
 \string\verb% % to fool emacs highlighting
 \noexpand\quotechar*\noexpand\verbatimchar
 \noexpand\bslash\noexpand\@gtempa\noexpand\verbatimchar
 \noexpand\doc@handleencap{usage}}%
    \fi
  \noexpand\fi
    \noexpand\@esphack}}
```

`\doc@createdescribe` {⟨*item*⟩}

```
\def\doc@createdescribe#1{%
  \@namedef{Describe#1}{%
```

Because of the optional argument we have to set `\MakePrivateLetters` already before parsing that (fingers crossed). Otherwise incorrect but quite common usage, such as `\DescribeMacro\foo@bar` will break because the scan for the optional argument will tokenize the following input (i.e., `\foo` in that case) before the `@` sign becomes a letter. As a result `DescribeMacro` would receive only `\foo` as its argument.

```
    \begingroup
      \MakePrivateLetters
      \@ifnextchar[%]
      {\doc@describe{#1}}{\doc@describe{#1}[]}}}
```

`\doc@createenv` `{`⟨*item*⟩`}{`⟨*envname*⟩`}`

```
\def\doc@createenv#1#2#3{%
  \@namedef{#3}{%
    \@ifnextchar[%]
    {\doc@env{#1}{#2}}{\doc@env{#1}{#2}[]}}%
```

Instead of `\let`ting the end of the environment to `\endtrivlist` we use one level of expansion. This way any possible change in that environment (if that ever happens) is properly reflected.

```
  \@namedef{end#3}{\endtrivlist}%
%  \expandafter\let\csname end#3\endcsname\endtrivlist
}
```

`\@nameedef`

```
\def\@nameedef#1{\expandafter\edef\csname #1\endcsname}
```

9.4 API creation

The whole user interface is created in one macro call.

```
defaults:

  idxtype   = #3
  idxgroup  = #3s
  printtype =
```

`\doc@declareerror`

```
\def\doc@declareerror#1#2{%
   \PackageError{doc}{Doc element '#1/#2' already defined?\@gobble}%
      {There is already a definition for
       '\string\Print#1Name',\MessageBreak
       '\string\PrintDescribe#1'
       or the environment '#2'.\MessageBreak
       Maybe you are overwriting something by mistake!\MessageBreak
       Otherwise use '\string\RenewDocElement' instead.}%
}
```

`\doc@notdeclarederror`

```
\def\doc@notdeclarederror#1#2{%
   \PackageError{doc}{Doc element '#1/#2' unknown}%
```

```
      {I expected an existing definition for
       '\string\Print#1Name',\MessageBreak
       '\string\PrintDescribe#1' and
       the environment '#2' but\MessageBreak
       not all of them are defined.\MessageBreak
       Maybe you wanted to use
       '\string\NewDocElement'?}%
}
```

\doc@ignoredinfo

```
\def\doc@ignoredinfo#1#2{%
  \PackageInfo{doc}{Doc element '#1/#2' declaration
    ignored}%
}
```

\NewDocElement [⟨*options*⟩]{⟨*name*⟩}{⟨*envname*⟩}

```
\newcommand\NewDocElement[3][]{%
  \@ifundefined{Print#2Name}%
      {\@ifundefined{PrintDescribe#2}%
           {\@ifundefined{#3}%
               {\@ifundefined{end#3}%
                    {\@NewDocElement{#1}}%
                    \doc@declareerror
               }\doc@declareerror
           }\doc@declareerror
      }\doc@declareerror
  {#2}{#3}%
}
```

\ProvideDocElement [⟨*options*⟩]{⟨*name*⟩}{⟨*envname*⟩} This does nothing unless the doc element could be declared with \NewDocElement.

```
\newcommand\ProvideDocElement[3][]{%
  \@ifundefined{Print#2Name}%
      {\@ifundefined{PrintDescribe#2}%
           {\@ifundefined{#3}%
               {\@ifundefined{end#3}%
                    {\@NewDocElement{#1}}%
                    \doc@ignoredinfo
               }\doc@ignoredinfo
           }\doc@ignoredinfo
      }\doc@ignoredinfo
  {#2}{#3}%
}
```

\RenewDocElement [⟨*options*⟩]{⟨*name*⟩}{⟨*envname*⟩}

```
\newcommand\RenewDocElement[3][]{%
  \@ifundefined{Print#2Name}\doc@notdeclarederror
      {\@ifundefined{PrintDescribe#2}\doc@notdeclarederror
           {\@ifundefined{#3}\doc@notdeclarederror
```

```
                {\@ifundefined{end#3}\doc@notdeclarederror
                     {\@NewDocElement{#1}}%
                }%
            }%
       }%
  {#2}{#3}%
}
```

`\@NewDocElement` {⟨*options*⟩}{⟨*name*⟩}{⟨*envname*⟩}

```
\def\@NewDocElement#1#2#3{%
  \doc@macrolikefalse
  \doc@topleveltrue
```

TODO: *Use 2e interface for* `\keys_set:nn` *when available*

```
  \def\doc@idxtype{#3}%
  \def\doc@idxgroup{#3s}%
  \let\doc@printtype\@empty
  \csname keys_set:nn\endcsname{doc}{#1}%
```

`\Print...Name` {⟨*name*⟩}

TODO: *extremely messy this with so many* `\expandafter`*s ... should reimplement in expl3*

```
  \ifx\doc@printtype\@empty
    \@temptokena{}%
  \else
    \@temptokena\expandafter{\expandafter
         \textnormal\expandafter{\expandafter
         \space\expandafter
         (\doc@printtype)}}%
  \fi
  \@nameedef{Print#2Name}##1{%
     {\noexpand\MacroFont
      \ifdoc@macrolike
         \noexpand\string
      \fi
      ##1%
      \the\@temptokena
     }}%
```

`\PrintDescribe...` {⟨*name*⟩}

```
  \expandafter\let\csname PrintDescribe#2\expandafter\endcsname
                  \csname Print#2Name\endcsname
```

`\SpecialMain...Index` {⟨*name*⟩}

`\Special...Index` {⟨*name*⟩}

```
  \edef\doc@expr{%
     \ifdoc@macrolike
       \noexpand\doc@createspecialmacrolikeindexes
     \else
       \noexpand\doc@createspecialindexes
```

```
      \fi
      {#2}%
     }%
    \expandafter\expandafter\expandafter
    \doc@expr
    \expandafter\expandafter\expandafter
      {\expandafter\doc@idxtype\expandafter}\expandafter
      {\doc@idxgroup}%
```

`\Describe...` `[`⟨*options*⟩`]{`⟨*name*⟩`}`

```
  \doc@createdescribe{#2}%
```

`\metaDocElement` (*env.*) **TODO:** *can't have formatting in argument – fix*
`[`⟨*options*⟩`]{`⟨*name*⟩`}`

```
    \ifdoc@macrolike
      \doc@createenv{TT}{#2}{#3}%
    \else
      \doc@createenv{TF}{#2}{#3}%
    \fi
}
```

9.5 Setting up the default `doc` elements

9.5.1 Macro facilities

Macros get only a single index entry (no index group, no index type) and they do not get any label either when printing in the margin.

```
\NewDocElement[macrolike = true ,
                idxtype   = ,
                idxgroup  = ,
                printtype =
               ]{Macro}{macro}
```

`\SpecialMainIndex` In `doc` v2 we had `\SpecialMainIndex` and `\SpecialMainEnvIndex` but now with additional `doc` elements we always add the element name after "`Main`" so this would be `\SpecialMainMacroIndex`. We use `\def` not `\let` so any redefinition of `\SpecialMainMacroIndex` will be transparent.

```
\def\SpecialMainIndex{\SpecialMainMacroIndex}
```

`\SpecialUsageIndex` `doc` v2 also had `\SpecialUsageIndex` which is now called `\SpecialMacroIndex` generating the "usage" index entry for a macro. Again we provide that as an alias via `\def`.

In fact the documentation of `doc` v2 claimed that one can use this for both macros and environments but that was never true as for environments the result was that the first character was dropped in sorting of the index. The correct way is to use `\SpecialEnvIndex` for this.

```
\def\SpecialUsageIndex{\SpecialMacroIndex}
```

`\SpecialIndex`

```
\def\SpecialIndex     {\CodeMacroIndex{code}}
```

9.5.2 Environment facilities

Providing documentation support for environments. Here we differ from doc V2 by marking the environments with "(*env.*)" when printing the name in the margin.

```
\NewDocElement[macrolike = false ,
                idxtype   = env.  ,
                idxgroup  = environments ,
                printtype = \textit{env.}
               ]{Env}{environment}
```

To be able to restore the definition after hypdoc is loaded we better save them beforehand. We only load the package at the end of the preamble, but the user might do this earlier and then chaos is ensured. Thus, to support this generally we save them directly before the package is loaded. In this way the user can still alter the definition for `\PrintDescribeMacro` and friends in the preamble.

```
\AddToHook{package/hypdoc/before}{%
  \let\@@PrintDescribeMacro \PrintDescribeMacro
  \let\@@PrintDescribeEnv \PrintDescribeEnv
  \let\@@PrintMacroName \PrintMacroName
  \let\@@PrintEnvName \PrintEnvName
  \let\@@SpecialUsageIndex \SpecialUsageIndex
  \let\@@SpecialEnvIndex \SpecialEnvIndex
  \let\@@SortIndex \SortIndex
  \let\@@DescribeMacro \DescribeMacro
  \let\@@DescribeEnv \DescribeEnv
}
```

After hypdoc got loaded we need to reset those macros again. This is done in the generic hook `package/hypdoc/after`.

```
\AddToHook{package/hypdoc/after}{%
  \let\PrintDescribeMacro \@@PrintDescribeMacro
  \let\PrintDescribeEnv \@@PrintDescribeEnv
  \let\PrintMacroName \@@PrintMacroName
  \let\PrintEnvName \@@PrintEnvName
  \let\SpecialUsageIndex \@@SpecialUsageIndex
  \let\SpecialEnvIndex \@@SpecialEnvIndex
  \let\SortIndex \@@SortIndex
  \let\DescribeMacro \@@DescribeMacro
  \let\DescribeEnv \@@DescribeEnv
}
```

10 Misc additions

`\cs`

```
\DeclareRobustCommand\cs[1]{\texttt{\bslash #1}}
```

amsdtx has its own definition for `\cs` but that now gets overwritten because the class loads doc afterwards. So for now we reinstall it here.

TODO: *fix elsewhere*

```
\AddToHook{class/amsdtx/after}{%
  \DeclareRobustCommand\cs[1]{%
    \@boxorbreak{%
        \ntt
```

```
1234          \addbslash#1\@empty
1235          \@xp\@xp\@xp\@indexcs\@xp\@nobslash\string#1\@nil
1236      }%
1237    }%
1238    \def\cn{\cs}%
1239 }
```

We can now finish the `docstrip` main module.

```
1240 ⟨/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`. The file is part of core LaTeX.

[7] R. E. Raspe (*1737, †1797). Baron Münchhausens narrative of his marvelous 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.

P

T

Change History