The xdoc package — experimental reimplementations of features from doc, second prototype

Lars Hellström*

2003/07/07

Abstract

The xdoc package contains reimplementations of some of the features found in the standard LaTeX doc package [5] by Mittelbach *et al.* The ultimate goals for these reimplementations are that the commands should be better, easily configurable, and be easy to extend, but this is only a second prototype implementation and nothing in it is guaranteed to be the same in the third prototype.[1]

Contents

---

*E-mail: `Lars.Hellstrom@math.umu.se`

[1] But there are no guarantees there will ever be a third prototype either.

1 Usage

When I began working on this package I thought that there would be no need for a usage section (at least on the prototype stage)—either you are interested in using the new features and then you might just as well read the descriptions of the commands in the implementation part of this document (they are written as specifications of what the commands do), or else you can simply insert a `\usepackage{xdoc2}` in the preamble and see how things work a little better than when you simply use doc—but with some features it became natural to introduce incompatible changes and some new features ought to be mentioned. Hence I wrote a short section on usage after all.

It is my intention that this document will eventually evolve into the source for a package xdoc[2] which will either build on the doc package and provide better implementations of many of its features, or replace it completely, but this document is still only the source for a prototype for that package. As I believe that the need for some improvement in this area is rather large however, I have decided to release this prototype so that other people can use it in their documents or create packages that are based on it. In doing so, one must of course bear in mind that this prototype needs not be compatible with the final xdoc package, and to overcome most incompatibility problems I therefore release it under the variant name xdoc2. This way, documents based on this prototype can still be typeset using the package they were written for long after the next xdoc prototype (or final version) is released.

Thus although this document frequently speaks of xdoc, you might just as well read it as xdoc2.

---

[2] The name doc2 has also been discussed; we'll see when we get there.

1.1 Changes to old features

`dolayout` option

Whereas `doc` more or less assumes that all pages have the same layout, `xdoc` takes measures to ensure that the `doc` features support two-sided document designs. If the left margin has been widened to better accommodate long macro names however (like for example the ltxdoc document class does), then you may find that the outer margin on right (odd) pages is too narrow for printing macro names in. The remedy for this is the `dolayout` option; in two-sided mode it causes `xdoc` to recompute the `\oddsidemargin` so that the outer margin has the same size on right pages as it previously did on left pages. In documents which are not processed in two-sided mode the `dolayout` option has no effect.

`\DocInput` has been changed to not make percent a comment character upon return unless it was before the `\DocInput`. This makes `\DocInput` nestable and I recommend that `.dtx` files which input other `.dtx` files use `\DocInput` for this.

`olddocinclude` option

`fileispart` option

The `\DocInclude` command, which is defined by the ltxdoc document class rather than `doc`, is also by default redefined in an incompatible manner by `xdoc`, but you can stop `xdoc` from making incompatible changes if you pass it the option `olddocinclude`. The main incompatibility lies in that the default redefinition of `\DocInclude` behaves purely as an `\include` command which `\DocInput`s a `.dtx` file rather than merely `\input`ting a `.tex` file—you must pass the `fileispart` option to `xdoc` to get the `\part` headings etc. for each new file—but there are also minor changes in the appearance of these headings, in how page styles are set, and in how the information presented in the page footer is obtained.

Other changes are as far as I can tell minor and within the bounds of expected behaviour, but code that relies on the implementation of some feature in `doc` may of course behave differently or break completely. Note in particular that the formats of the internal `doc` variables `\saved@macroname`, `\macro@namepart`, and `\index@excludelist` have changed completely (see Section 7, Subsection 5.1, and Subsection 5.2 respectively)—hence any hack involving one of these must be revised before it is used with `xdoc`. These are however exceptions; in my experience the most noticeable changes not listed above are that the index exclude mechanism actually works for control sequences whose names consist of a single non-letter and that symbols get sorted in a different order.

1.2 Some notable new features

`\NewMacroEnvironment`

`macro`
`environment`
`option`
`switch`

The main new feature is the `\NewMacroEnvironment` command, which defines a new `macro`-like environment. The command offers complete control of the argument structure, the formatting of the marginal heading, the code for making index entries, and the change entry sorting and formatting, but the syntax is too complex to explain here. Those who are interested in using it should read Section 8. In particular, Subsections 8.3–8.4 contain several examples of how it can be used. In addition to using `\NewMacroEnvironment` for redefining the `macro` and `environment` environments, `xdoc` also defines an `option` environment (which is intended for document class and package options) and a `switch` environment (which is intended for switches defined using `\newif`; the argument should not include the `\if`).

`\NewDescribeCommand`

There is also a companion command `\NewDescribeCommand` which defines new commands similar to `\DescribeMacro` and `\DescribeEnv`. The syntax of `\NewDescribeCommand` is also too complex to explain here, so I have to refer readers who

`\describeoption`
`\describecsfamily`

want to use it to Section 9. Two more commands which are defined in that section are `\describeoption`, which is the `describe`... companion of the `option` environment, and `\describecsfamily` which is meant for describing control sequence families (see the table on page 58 for examples of what I mean). The argument of this latter command is simply the material you would put between `\csname` and `\endcsname`. Variant parts are written as `\meta{`⟨*text*⟩`}` and print as one would expect them to (but notice that the ⟨*text*⟩ is a moving argument) whereas most other characters can be written verbatim without any special quoting (but `\`, `{`, `}`, and `%` need quoting; see the comments to the definition of `\describecsfamily` for information on how to do that).

`\DoNotIndexBy`

The `\DoNotIndexBy` command tells the commands that make index entries for macros to ignore a certain character sequence when the index entries are sorted. The `\DoNotIndexBy` command takes one argument: the character sequence to ignore. If `\DoNotIndexBy` is used more than once then the indexing commands will look for, and if it finds it ignore, each of the character sequences given to it, starting with the one specified last.

`\setfileinfo`

It has already been mentioned that the `\DocInclude` command has been changed. What has not been mentioned is its companion `\setfileinfo`, which the partfiles should use for setting the date and version information presented in the page footer, but that is explained in detail in Subsection 10.2.

`\definechange`
`\usechange`
`\uintver`

Finally there is a new variant of the `\changes` command which is intended for changes that, although not limited to a single macro and thus being "general" changes in the doc terminology, affect only a few (probably widely dispersed) macros (or whatever). The basic idea is that you can define a change with a specific version, date, and text using the `\definechange` command and then recall those parameters later using the `\usechange` command. Primarily this ensures that the entry texts are identical so that makeindex will combine them into one entry, but it is also specified which macro was changed at which page. See Section 7 for more details. Another new feature concerning `\changes` is that there is now support for sorting version numbers according to mathematical order rather than ASCII order. Traditionally the version numbers `2`, `11`, and `100` would have been sorted so that `100` < `11` < `2`, but if they are entered as `\uintver{2}`, `\uintver{11}`, and `\uintver{100}` then they will be sorted as `2` < `11` < `100`. The argument of `\uintver` must be a TeX ⟨*number*⟩.

`\PrintChar`
`\MakeHarmless`

xdoc also contains several features which are of little use as direct user commands, but which can simplify the definitions of other commands. The foremost of these are the 'harmless character strings', which can be seen as a datatype for (short pieces of) verbatim text. TeX typesets a harmless character string in pretty much the same way as the corresponding string of 'other' tokens, but the harmless character string can also be written to file and read back arbitrarily many times without getting garbled, it doesn't make makeindex choke, and it survives being fed to a `\protected@edef`. The most important commands related to harmless character strings are `\PrintChar`, which is used for representing problematic characters, and `\MakeHarmless`, which converts arbitrary TeX code to the corresponding harmless character string.

`\IndexEntry`

The superfluity of indexing commands in doc has been replaced by the single command `\IndexEntry`, which has been designed with the intention that it should provide a clear interface between the user level macros and the index sorting program. It takes three arguments: the index entry specification, the name of the encapsulation scheme that should be used, and the number to put in the index. The index entry specification is a sequence of `\LevelSame` and/or `\LevelSorted` commands, which have the respective syntaxes

`\LevelSame` `\LevelSame{`⟨*text*⟩`}`
`\LevelSorted` `\LevelSorted{`⟨*sort key*⟩`}{`⟨*text*⟩`}`

Each such command specifies one level of the index entry. In the case of `\LevelSorted`, the ⟨*text*⟩ is what will be written in the sorted index at that level and ⟨*sort key*⟩ is what the index-sorting program should look at when sorting the entry (at that level). In the case of `\LevelSame`, the ⟨*text*⟩ is used both as sort key and contents of entry in the sorted index. The first command is for the topmost level and each subsequent command is for the next sublevel. The complete description appears in Subsection 4.1.

xdoc also contains support for external cross-referencing programs (see Subsection 5.3 for details) and a system for determining whether a piece of text falls on an even or an odd page (see Section 6 for details). I expect that the latter system will eventually migrate out of xdoc, either to a package of its own, or into oblivion because the LaTeX $2_{\varepsilon}*$ output routine makes it obsolete.

1.3 The **docindex** package

As of prototype version 2.2, the xdoc package has a companion package docindex [2] which provides improved formatting of the index and list of changes. xdoc works fine without docindex, however.

1.4 A note on command names

The doc package defines several commands with mixed-case names which (IMHO) should really have all-lower-case names (according to the rule of thumb spelled out in [4, Ssec. 2.4]) since people use them in the capacity of being the author of a `.dtx` file rather than in the capacity of being the writer of a class or package. The names in question are

Name in doc	**Better (?) name**
`\AlsoImplementation`	`\alsoimplementation`
`\CharacterTable`	`\charactertable`
`\CharTableChanges`	`\chartablechanges`
`\CheckModules`	`\checkmodules`
`\CheckSum`	`\checksum`
`\CodelineIndex`	`\codelineindex`
`CodelineNo` (counter)	`codelineno`
`\CodelineNumbered`	`\codelinenumbered`
`\DeleteShortVerb`	`\deleteshortverb`
`\DescribeEnv`	`\describeenv`
`\DescribeMacro`	`\describemacro`
`\DisableCrossrefs`	`\disablecrossrefs`
`\DocInput`	`\docinput`
`\DoNotIndex`	`\donotindex`
`\DontCheckModules`	`\dontcheckmodules`
`\EnableCrossrefs`	`\enablecrossrefs`

Name in doc	**Better (?) name**
`\Finale`	`\finale`
`GlossaryColumns` (counter)	`glossarycolumns`
`\GlossaryPrologue`	`\glossaryprologue`
`IndexColumns` (counter)	`indexcolumns`
`\IndexInput`	`\indexinput`
`\IndexPrologue`	`\indexprologue`
`\MakePrivateLetters`	`\makeprivateletters`
`\MakeShortVerb`	`\makeshortverb`
`\OnlyDescription`	`\onlydescription`
`\PageIndex`	`\pageindex`
`\PrintChanges`	`\printchanges`
`\PrintIndex`	`\printindex`
`\RecordChanges`	`\recordchanges`
`\SortIndex`	`\sortindex`
`\SpecialEscapechar`	`\specialescapechar`
`StandardModuleDepth` (counter)	`standardmoduledepth`
`\StopEventually`	`\stopeventually`

With the exception for `CodelineNo`,[3] I haven't changed any of the doc names in this xdoc prototype, nor introduced any of the "better names" as alternatives, but I think the matter should be given a bit of thought during the future development of doc/xdoc.

For completeness, I should also remark that there are several macros that doc gives mixed-case names which I haven't listed above. The logo command names have special capitalizing rules by tradition. Some macros and named registers—for example `\DocstyleParms`, `\IndexParms`, `\MacroFont`, `\MacroTopsep`, `\MakePercentIgnore`, and `\PrintMacroName`—are part of the package or document class writer's interface to doc, although I cannot claim it to be obvious that for example `\IndexParms` and the `IndexColumns` counter should belong to different classes here (but several of these control sequences will probably disappear from the interface in LaTeX 2$_\varepsilon$* anyway, so the problem isn't that important). The `\Special`...`Index` commands (and their even more special variants, such as `\LeftBraceIndex`) are internal commands rather than user level commands. Finally there is the `\GetFileInfo` command, which I doubt there is any point in having.

1.5 docstrip modules

The docstrip modules in `xdoc2.dtx` are:

pkg This module directive surrounds the code for the xdoc package.

driver The driver.

internals This module contains an alternative replacement text for the `\PrintVisibleChar` command that uses "LaTeX internal character representation" (i.e., as much as possible encoding-specific commands—`\text`... commands

---

[3] Where I recommend using `codelineno` instead of `CodelineNo`, `\PrintCodelineNo` instead of `\theCodelineNo`, and `\thecodelineno` instead of `\number\c@CodelineNo`; see Subsection 11.4.

and the like) rather than the primitive `\char` command for typesetting visible characters. It is provided as a separate module mainly for compability with prototype version 2.0, as this alternative definition can (as of prot. 2.1) be chosen by passing the option `notrawchar` to xdoc.

`notrawchar` option

economical There is little point in storing the harmless representations of the 161 non-visible-ASCII characters as these representations are always the same and can be formed on the fly whenever they are needed. The economical modules contain some alternative code which makes use of this fact to reduce the number of control sequences used for storing the table of harmless representations. The ⟨economical⟩ module appears inside the ⟨pkg⟩ module.

xdoc2 This module contains code for compability with previous releases of xdoc2. It will not be included in xdoc3 or xdoc (whichever is the next major version).

enccmds This module contains the code for defining two `macro`-like environments for encoding-specific commands. These are not included in the xdoc package since so few `.dtx` files define encoding-specific commands.

rsrccmd Similar to the enccmds module, but demonstrates the `\NewDescribeCommand` command instead.

example This surrounds some code which to docstrip looks like it should be copied, but isn't meant to.

2 Initial stuff

First there's the usual `\NeedsTeXFormat` and `\ProvidesPackage`.

```
⟨*pkg⟩
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{xdoc2}[2003/07/06 prot2.5 doc reimplementation package]
```

Options

The first option has to do with the page layout. Although doc itself doesn't modify any of the main layout parameters, it is well known that using it does tend to restrict one's choices in terms of document layout. In particular the `macro` and `environment` environments require a rather large left margin since they will otherwise print long macro names partially outside the paper. It is furthermore hard to decrease the `\textwidth` as it should be wide enough to contain about 70 columns of `\MacroFont` text. Thus the only solution is to do as the ltxdoc [1] document class and enlarge the left margin at the expense of the right.

The resulting layout has a left–right asymmetry with the main galley (the text rectangle) on the right and a very wide left margin (in which marginal headings and marginal notes appears). Although this layout is not uncommon in technical manuals, it is inappropriate for two-sided designs since the vertical line at which the two pages of a spread meet becomes the natural vertical symmetry axis for the entire spread and it breaks this symmetry to let the left margin be the widest on all pages. It would look better to always let the outer margin be the largest.

`dolayout` option
`\oddsidemargin`

The `dolayout` option modifies `\oddsidemargin` so that spreads are symmetric around the center in two-sided mode. As size of the outer margin is taken the size of the left margin on left (even) pages, i.e., `\evensidemargin` + 1 in.

In one-sided mode, the `dolayout` option does nothing.

```
\DeclareOption{dolayout}{%
   \if@twoside
      \setlength\oddsidemargin{\paperwidth}
      \addtolength\oddsidemargin{-\textwidth}
      \addtolength\oddsidemargin{-\evensidemargin}
      \addtolength\oddsidemargin{-2in}
   \fi
}
```

`olddocinclude` option
`fileispart` option

The `olddocinclude` and `fileispart` options are related to the `\DocInclude` command defined by the ltxdoc document class. Some of the code related to that command relies on modifying the doc internal macro `\codeline@wrindex`, but that has no effect with xdoc so in order to get the expected results one has to reimplement the `\DocInclude` command as well. The `olddocinclude` and `fileispart` options control how this should be done.

If the `olddocinclude` option is passed to xdoc then only the parts of the implementation of `\DocInclude` which must be altered to make the command work with the xdoc implementation of indexing and cross-referencing are changed. These redefinitions will furthermore only be made if the ltxdoc document class has been loaded; nothing is done if the `olddocinclude` option is passed and ltxdoc hasn't been loaded. Passing the `olddocinclude` option can be considered as requesting a "compatibility mode" for `\DocInclude`.

If the `olddocinclude` option is not passed then the `\DocInclude` command is reimplemented from scratch, regardless of whether some definition of it has already been given or not. The basis of this reimplementation is the observation that the `\DocInclude` command of ltxdoc really does two quite distinct things at once—it is an `\include` command which `\DocInput`s files rather than `\input`ting them, but it also starts a new `\part`, sets the pagestyle, and changes how the values of some counters are typeset. This latter function is by default disabled in the xdoc implementation of `\DocInclude`, but passing the `fileispart` option enables it.

There is no code for these two options here, as it is rather long; instead that code appears in Section 10. The `\PassOptionsToPackage` commands make sure that these options are registered as local options for xdoc, so that one can test for them using `\@ifpackagewith` below.

```
\DeclareOption{olddocinclude}{%
   \PassOptionsToPackage{\CurrentOption}{xdoc2}%
}
\DeclareOption{fileispart}{%
   \PassOptionsToPackage{\CurrentOption}{xdoc2}%
}
```

`notrawchar` option

The `notrawchar` option controls how the `\PrintVisibleChar` command is defined, and thereby what method is used for typesetting visible characters in e.g. macro names. The default is to use the `\char` primitive (which is better for `T1`-encoded fonts and non-italic `OT1`-encoded typewriter fonts), but the `notrawchar` option causes things to go via the "LaTeX internal character representation" instead (which is necessary for e.g. `OT1`-encoded non-typewriter fonts).

There is no code for this option here; instead that code is found in the definition of `\PrintVisibleChar`.

```
\DeclareOption{notrawchar}{%
   \PassOptionsToPackage{\CurrentOption}{xdoc2}%
}
```

Then options are processed.

```
\ProcessOptions\relax
```

And finally the doc package is loaded.

```
\RequirePackage{doc}
```

3 Character strings

A source of much of the complexity in doc is that it has to be able to deal with rather arbitrary strings of characters (mainly the names of control sequences). Once the initial problems with characters having troublesome catcodes have been overcome however, it is usually no problem to manage such things in TeX. doc does however complicate things considerably by also putting these things in the index and list of changes. Not only must they then be formatted so that the makeindex program doesn't choke on them, but they must also be wrapped up in code that allows TeX to make sense of them when they are read back. doc manages the makeindex problems mainly by allowing the user to change what characters are used as makeindex metacharacters and the reading back problem by making abundant use of `\verb`.

All this relies on that the author of a document is making sure that the metacharacters aren't used for anything else. If for example the `\verbatimchar` (by default `+`) is one of the "private letters" then names of control sequences containing that character will be typeset incorrectly because the `\verb` used to typeset it is terminated prematurely—control sequence names such as '`\lost+found`' will be typeset as '`\lost`found+'. On top of that, one also has to make sure that the font used for typesetting these `\verb` sections contains all the characters needed.

For xdoc, I have chosen a completely different approach. Instead of allowing the strings (after they have converted to the internal format) to contain TeX character tokens with arbitrary character codes, they may only contain TeX character tokens which are unproblematic—the normal catcode should be 11 (letter) or 12 (other), they should not be outside visible ASCII, and they may not be one of the makeindex metacharacters. All other characters are represented using a robust command which takes the character code (in decimal) as the argument. This takes care of all "moving argument" type problems that may occur.

An important observation about these character strings is that they are strings of *input* characters. This means that rather than using the characters in some special font for typesetting control sequences like `\^^M` (recall that the `^^` substitutions take place before tokenization), one should typeset them using only visible ASCII characters. (After all, that's the only way they are written in TeX code.) The default definition is to typeset invisible characters as precisely the `^^`-sequences that TeX normally uses for these characters when they are written to a file.

3.1 Typesetting problematic characters

`\PrintChar`
`\XD@threedignum`

The `\PrintChar` command has the syntax

`\PrintChar{`⟨*8-bit number*⟩`}`

where ⟨*8-bit number*⟩ is a TeX number in the range 0–255. For arguments in the range 0–31, `\PrintChar` prints ‘`^^@`’–‘`^^_`’. For an argument in the range 32–126, `\PrintChar` calls `\PrintVisibleChar` which by default simply does `\char` on that argument (but which can be redefined if the font set-up requires it); in particular, `\PrintChar{32}` should print a “visible space” character. `\PrintChar{127}` prints ‘`^^?`’. For arguments in the range 128–255, `\PrintChar` prints ‘`^^`*80*’–‘`^^`*ff*’.

`\PrintChar` is robust. `\PrintChar` also has a special behaviour when it is written to a file (when `\protect` is `\noexpand`): it makes sure that the argument consists of three decimal digits, to ensure external sorting gets it right.

```
\@ifundefined{PrintChar}{}{%
   \PackageInfo{xdoc2}{Redefining \protect\PrintChar}%
}
\def\PrintChar{%
   \ifx \protect\@typeset@protect
      \expandafter\XD@PrintChar
   \else\ifx \protect\noexpand
      \string\PrintChar
      \expandafter\expandafter \expandafter\XD@threedignum
   \else
      \noexpand\PrintChar
   \fi\fi
}
```

`\XD@threedignum` does a `\number` on its argument, possibly prepends a `0` or two, and wraps it all up in a “group” (the braces have category other, not beginning and end of group).

```
\edef\XD@threedignum#1{%
   \string{%
   \noexpand\ifnum #1<100 %
      \noexpand\ifnum #1<10 0\noexpand\fi
      0%
   \noexpand\fi
   \noexpand\number#1%
   \string}%
}
```

`\XD@PrintChar`
`\InvisibleCharPrefix`
`\InvisibleCharSuffix`

`\XD@PrintChar` manages the typesetting for `\PrintChar`. It distinguishes between visible characters (code 32–126) and invisible characters. The visible characters are typeset directly using `\PrintVisibleChar`, whereas the invisible characters are typeset as `^^`-sequences.

The macros `\InvisibleCharPrefix` and `\InvisibleCharSuffix` begin and end a `^^`-sequence. `\InvisibleCharPrefix` should print the actual `^^`, but it may also for example select a new font for the `^^`-sequence (such font changes are restored at the end of `\XD@PrintChar`).

```
\def\XD@PrintChar#1{%
   \leavevmode
   \begingroup
      \count@=#1\relax
      \ifnum \@xxxii>\count@
         \advance \count@ 64%
         \InvisibleCharPrefix
         \PrintVisibleChar\count@
         \InvisibleCharSuffix
      \else\ifnum 127>\count@
         \PrintVisibleChar\count@
      \else
         \InvisibleCharPrefix
         \ifnum 127=\count@ \PrintVisibleChar{63}\else
            \@tempcnta=\count@
            \divide \count@ \sixt@@n
            \@tempcntb=\count@
            \multiply \count@ \sixt@@n
            \advance \@tempcnta -\count@
            \advance \@tempcntb \ifnum 9<\@tempcntb 87\else 48\fi
            \advance \@tempcnta \ifnum 9<\@tempcnta 87\else 48\fi
            \char\@tempcntb \char\@tempcnta
         \fi
         \InvisibleCharSuffix
      \fi\fi
   \endgroup
}
```

```
\newcommand\InvisibleCharPrefix{%
   \/\em
   \PrintVisibleChar{`\^}\PrintVisibleChar{`\^}%
}
\newcommand\InvisibleCharSuffix{\/}
```

There are some alternative methods for making hexadecimal numbers which should perhaps be mentioned. The LaTeX kernel contains a macro `\hexnumber@` which uses `\ifcase` to produce one hexadecimal digit, but that uses upper case letters, and things like '8E' look extremely silly if the upper case letters doesn't line with the digits. Applying `\meaning` to a ⟨*chardef token*⟩ or ⟨*mathchardef token*⟩ expands to `\char"`⟨*hex*⟩ and `\mathchar"`⟨*hex*⟩ respectively, where ⟨*hex*⟩ is the corresponding number in hexadecimal, but that too has upper case A–F and leading zeros are removed.

`\PrintVisibleChar` The `\PrintVisibleChar` command should print the visible ASCII character whose character code is given in the argument. There are currently two definitions of this command: one which uses the TeX primitive `\char` and one which goes via the "LaTeX internal character representation" for the character. By default `xdoc` uses the former definition, but if `xdoc` is passed the `notrawchar` option then it will use the latter.

The reason there are two definitions is a deficiency in how the NFSS encoding attribute has been assigned to fonts; even though the encodings of Computer Modern Roman and Computer Modern Typewriter are quite different, LaTeX 2$_\varepsilon$ uses the `OT1` encoding for both. As a result of this, the LaTeX internal representation will in some important cases use characters from non-typewriter fonts despite the fact that typewriter forms are immediately available. Since the cases in which the `\char` primitive produces results as least as good as those made through the

LaTeX internal character representation includes those that the current font is `T1`-encoded or an `OT1`-encoded nonitalic typewriter font, the shorter `\char` primitive defintion has been made the default.

For compability with prototype version 2.0 of `xdoc`, the replacement text for `\PrintVisibleChar` that uses LaTeX internal character representation can alternatively be extracted by docstripping `xdoc2.dtx` with the option ⟨internals⟩.

```
\@ifpackagewith{xdoc2}{notrawchar}{%
  \newcommand\PrintVisibleChar[1]{%
⟨/pkg⟩
⟨*pkg | internals⟩
    \ifcase #1%
    \or\or\or\or\or\or\or\or \or\or\or\or\or\or\or\or
    \or\or\or\or\or\or\or\or \or\or\or\or\or\or\or\or
    % "20
      \textvisiblespace \or!\or\textquotedbl \or\#\or\textdollar
      \or\%\or\&\or\textquoteright\or(\or)\or*\or+\or,\or-\or.\or/%
    \or % "30
      0\or1\or2\or3\or4\or5\or6\or7\or8\or9\or:\or;\or
      \textless\or=\or\textgreater\or?%
    \or % "40
      @\or A\or B\or C\or D\or E\or F\or G\or
      H\or I\or J\or K\or L\or M\or N\or O%
    \or % "50
      P\or Q\or R\or S\or T\or U\or V\or W\or X\or Y\or Z\or [\or
      \textbackslash \or]\or\textasciicircum \or\textunderscore
    \or % "60
      \textquoteleft \or a\or b\or c\or d\or e\or f\or g\or h\or
      i\or j\or k\or l\or m\or n\or o%
    \or % "70
      p\or q\or r\or s\or t\or u\or v\or w\or x\or y\or z\or
      \textbraceleft \or\textbar \or\textbraceright \or
      \textasciitilde
    \fi
  }%
⟨/pkg | internals⟩
⟨*pkg⟩
}{%
  \newcommand\PrintVisibleChar[1]{\char #1\relax}%
}
```

`\Bslash` It turns out that it is very common to say `\PrintChar{92}` (backslash), so a macro which expands to that reduces typing.

```
\newcommand\Bslash{\PrintChar{92}}
```

3.2 Rendering character strings harmless

Replacing all problematic characters with `\PrintChar` calls certainly makes the strings easier to manage, but actually making those replacements is a rather complicated task. Therefore this subsection contains the macros necessary for doing these replacements.

The first problem is how to efficiently recognise the problematic characters. A first solution which gets rather far is to mainly look in the `\catcode` register for that character and keep the character as it is if the category found there is 11 or 12, but replace it with a `\PrintChar` command if the category is anything else. Two extra tests can be performed to take care of invisible ASCII, and the makeindex metacharacters can be cared for by locally changing their catcodes for when the string is processed. Unfortunately this doesn't work inside `macrocode` environments (where one would like to use it for the macro cross-referencing) since that environment changes the catcodes of several characters from being problematic to being unproblematic and vice versa.[4] As furthermore harmless character strings should be possible to move to completely different parts of the document, the test used for determining whether a character is problematic should yield the same result throughout the document.

`\XD@harmless@`⟨*code*⟩

Because of this, I have chosen a brute strength solution: build a table (indexed by character code) that gives the harmless form of every character. This table is stored in the `\XD@harmless@`⟨*code*⟩ family of control sequences, where the ⟨*code*⟩ is in the range `0`–`255`. Assignments to this table are global. In principle, the table should not change after the preamble, but there is a command `\SetHarmState` which can be used at any time for setting a single table entry. This could be useful for documents which, like for example [3], have nonstandard settings of `\catcode`s.

`\SetHarmState`

The `\SetHarmState` command takes three arguments:

`\SetHarmState{`⟨*type*⟩`}{`⟨*char*⟩`}{`⟨*harm*⟩`}`

⟨*char*⟩ is the character whose entry should be set. ⟨*type*⟩ is a flag which specifies what format ⟨*char*⟩ is given in. If ⟨*type*⟩ is `\BooleanTrue` then ⟨*char*⟩ is the TeX ⟨*number*⟩ of the table entry to set, and if ⟨*type*⟩ is `\BooleanFalse` then ⟨*char*⟩ is something which expands to a single character token whose entry should be set. The expansion is carried out by an `\edef`, so it needs not be only one level. ⟨*harm*⟩ is `\BooleanTrue` if the character is problematic and `\BooleanFalse` if it is not.

The ⟨*type*⟩ and ⟨*harm*⟩ arguments are currently not subject to any expansion. In the future they probably should be, but I don't want to make assumptions about the actual definitions of `\BooleanTrue` and `\BooleanFalse` at this point.

```
\begingroup
   \catcode\z@=12
   \@ifdefinable\SetHarmState{
      \gdef\SetHarmState#1#2#3{%
         \begingroup
            \ifx #1\BooleanTrue
               \count@=#2\relax
            \else
               \protected@edef\@tempa{#2}%
               \count@=\expandafter`\@tempa\relax
            \fi
            \ifx #3\BooleanTrue
               \edef\@tempa{\noexpand\PrintChar{\the\count@}}%
            \else
               \uccode\z@=\count@
```

---

[4] As the entire `macrocode` environment is tokenized by the expansion of `\xmacro@code` one could alternatively solve this problem by reimplementing the `macrocode` environment so that normal catcodes are in force when the contents are being typeset.

```
               \uppercase{\def\@tempa{^^@}}%
            \fi
            \global\expandafter\let
               \csname XD@harmless@\the\count@ \endcsname \@tempa
         \endgroup
      }%
   }
\endgroup
```

`\XD@harmless@`⟨*code*⟩

Initializing the `\XD@harmless@`⟨*code*⟩ table is a straightforward exercise of `\loop` . . . `\repeat`.

```
⟨*!economical⟩
\count@=\z@
\loop
   \expandafter\xdef \csname XD@harmless@\the\count@ \endcsname
      {\noexpand\PrintChar{\the\count@}}%
   \advance \count@ \@ne
\ifnum 33>\count@ \repeat
⟨/!economical⟩
⟨economical⟩\count@=\@xxxii
\begingroup
   \catcode\z@=12\relax
   \@firstofone{%
\endgroup
   \loop
      \if \ifnum 11=\catcode\count@ 1\else \ifnum 12=\catcode\count@
            1\else 0\fi\fi 1%
         \uccode\z@=\count@
         \uppercase{\def\@tempa{^^@}}%
      \else
         \edef\@tempa{\noexpand\PrintChar{\the\count@}}%
      \fi
      \global\expandafter\let
         \csname XD@harmless@\the\count@ \endcsname \@tempa
      \advance \count@ \@ne
   \ifnum 127>\count@ \repeat
}
⟨*!economical⟩
\loop
   \expandafter\xdef \csname XD@harmless@\the\count@ \endcsname
      {\noexpand\PrintChar{\the\count@}}%
\ifnum \@cclv>\count@
   \advance \count@ \@ne
\repeat
⟨/!economical⟩
```

Marking the makeindex metacharacters as harmful is deferred until `\begin {document}`, since it is not unreasonable that these are changed in the preamble.

```
\AtBeginDocument{%
   \SetHarmState\BooleanFalse\actualchar\BooleanTrue
   \SetHarmState\BooleanFalse\encapchar\BooleanTrue
   \SetHarmState\BooleanFalse\levelchar\BooleanTrue
   \SetHarmState\BooleanFalse\quotechar\BooleanTrue
}
```

doc's `\verbatimchar` is not harmful, since it isn't used at all in xdoc.

`\MakeHarmless`

To render a character string harmless, you do

`\MakeHarmless{`⟨*macro*⟩`}{`⟨*string*⟩`}`

This locally assigns to ⟨*macro*⟩ the harmless character string which corresponds to ⟨*string*⟩. During the conversion the converted part of the string is stored in `\toks@`, but that is local to `\MakeHarmless`.

```
\def\MakeHarmless#1#2{%
   \begingroup
      \toks@={}%
      \escapechar=`\\%
      \XD@harmless@#2\XD@harmless@
   \expandafter\endgroup \expandafter\def \expandafter#1%
      \expandafter{\the\toks@}%
}
```

`\XD@harmless@iii`
`\XD@harmless@iv`
`\XD@harmless@v`
`\XD@harmless@vi`

What one has to be most careful about when rendering strings harmless are the space tokens, since many of TeX's primitives gladly snatches an extra space (or more) where you don't want them to in this case. Macro parameters can be particularly dangerous, as TeX will skip any number of spaces while looking for the replacement text for an undelimited macro argument. Therefore the algorithm for rendering a character token harmless begins (`\XD@harmless@iii`) with `\string`ing the next token in the string—this preserves the character code and sets the category to 12 for all characters except the ASCII space, which gets category 10 (space)—and then `\futurelet` is used to peek at the next token. If it is a space token (`\XD@harmless@iv`) then the character code is 32 and the actual space can be gobbled (`\XD@harmless@v`), and if it isn't then the next token can be grabbed in an undelimited macro argument (`\XD@harmless@vi`). In either case, the harmless form is given by the `\XD@harmless@`⟨*code*⟩ table entry (in `\XD@harmless@v` or `\XD@harmless@vi`).

```
\def\XD@harmless@iii{%
   \expandafter\futurelet \expandafter\@let@token
   \expandafter\XD@harmless@iv \string
}
```

```
\def\XD@harmless@iv{%
   \ifx \@let@token\@sptoken
      \expandafter\XD@harmless@v
   \else
      \expandafter\XD@harmless@vi
   \fi
}
```

```
\begingroup
   \catcode`3=\catcode`a
   \catcode`2=\catcode`a
   \@firstofone{\gdef\XD@harmless@v} {%
      \toks@=\expandafter{\the \expandafter\toks@ \XD@harmless@32}%
      \XD@harmless@
   }
\endgroup
```

In the ⟨economical⟩ (with hash table space) variant implementation the `\XD@harmless@`⟨*code*⟩ table has entries only for the characters in visible ASCII. Thus the harmless forms of characters outside visible ASCII must be constructed on the fly.

```
\def\XD@harmless@vi#1{%
⟨*economical⟩
   \if \ifnum ‘#1<\@xxxii 1\else \ifnum ‘#1>126 1\else 0\fi\fi 1%
      \toks@=\expandafter{\the\expandafter\toks@
         \expandafter\PrintChar \expandafter{\number‘#1}%
      }%
   \else
⟨/economical⟩
   \toks@=\expandafter{\the\expandafter\expandafter\expandafter\toks@
      \csname XD@harmless@\number‘#1\endcsname}%
⟨economical⟩    \fi
   \XD@harmless@
}
```

`\XD@harmless@`
`\XD@harmless@i`
`\XD@harmless@ii`
`\XD@harmless\`⟨*cs-name*⟩

But that is not all `\MakeHarmless` can do. In some cases (as for example when one is describing a family of control sequences) one might want to include things in the string that are not simply characters, but more complex items—such as for example `\meta` constructions like ⟨*code*⟩. To accommodate for this, `\XD@harmless@` (which is the first step in converting a token) always begins by checking whether the next token to render harmless is a control sequence. If it is then it is checked (in `\XD@harmless@ii`) whether the control sequence `\XD@harmless\`⟨*cs-name*⟩, where ⟨*cs-name*⟩ is the name without `\` of the control sequence encountered, is defined. If it isn't then the encountered control sequence is `\string`ed and conversion continues as above, but if it is defined then the encountered control sequence begins such a more complex item.

```
\def\XD@harmless@{\futurelet\@let@token \XD@harmless@i}
```

```
\def\XD@harmless@i{%
   \ifcat \noexpand\@let@token \noexpand\XD@harmless@
      \expandafter\XD@harmless@ii
   \else
      \expandafter\XD@harmless@iii
   \fi
}
```

```
\def\XD@harmless@ii#1{%
   \@ifundefined{XD@harmless\string#1}{%
      \expandafter\XD@harmless@vi \string#1%
   }{\csname XD@harmless\string#1\endcsname}%
}
```

`\XD@harmless\XD@harmless@`

A control sequence `\XD@harmless\`⟨*cs-name*⟩ is responsible for interpreting the string item that begins with the control sequence `\`⟨*cs-name*⟩ and appending a harmless representation of it to `\toks@`. Harmless representations should only contain robust control sequences and they must not rely on changing any catcodes. Normal `\XD@harmless\`⟨*cs-name*⟩ control sequences must also end by inserting `\XD@harmless@` in front of what remains of the string after the complex string item has been removed. This sees to that the rest of the string is also rendered harmless. The only such control sequence which does not insert `\XD@harmless@` is `\XD@harmless\XD@harmless@`, but that is as it should be since `\MakeHarmless` itself appends a `\XD@harmless@` to every character string it should convert to mark the end of it.

```
\expandafter\let
   \csname XD@harmless\string\XD@harmless@\endcsname \@empty
```

`\XD@harmless\PrintChar`

It is occasionally convenient to use a `\PrintChar` command as part of a string that is to be rendered harmless instead of using the raw character. The definition is very similar to that of `\XD@harmless@vi`.

```
\@namedef{XD@harmless\string\PrintChar}#1{%
⟨*economical⟩
   \if \ifnum #1<\@xxxii 1\else \ifnum #1>126 1\else 0\fi\fi 1%
      \toks@=\expandafter{\the\expandafter\toks@
         \expandafter\PrintChar \expandafter{\number#1}%
      }%
   \else
⟨/economical⟩
   \toks@=\expandafter{\the\expandafter\expandafter\expandafter\toks@
      \csname XD@harmless@\number#1\endcsname}%
⟨economical⟩   \fi
   \XD@harmless@
}
```

3.3 Interaction with mechanisms that make characters problematic

If additional visible characters are made problematic after the initial `\XD@harmless@`⟨*code*⟩ table is formed then problems may indeed arise, because some character which is expected to be unproblematic when read from (for example) an `.ind` file will actually not be. In fortunate cases this will only lead to that characters will print strangely or not at all, but it can quite conceivably lead to errors that prevent further typesetting and it should therefore be prevented if possible.

Right now, I can think of two mechanisms that make characters problematic, and both do that by making them active. One is the shorthand mechanism of babel, but I think I'll delay implementing any interaction with that until some later prototype; I don't know it well enough and anyway I don't think it is that likely to cause any problems. The other mechanism is the short verb mechanism of doc itself, and this should be taken care of right away.

The main difficulty is that the `\XD@harmless@`⟨*code*⟩ table should be the same throughout the document body (otherwise you may get more than one index entry for the same thing, with index references arbitrarily distributed between the two) whereas short verb characters can be made and deleted at any time. It would actually be wrong to always have the `\XD@harmless@`⟨*code*⟩ table entry mirroring the current state of the character! Instead a character will be considered as problematic even if it is only made problematic temporarily (with the exception for characters that are made problematic in `verbatim` environments and the like—the index file isn't being read in while those catcodes are active). Since it is impossible to know in the beginning of a document whether a character will be made a short verb character at some later point, the modifications to the `\XD@harmless@`⟨*code*⟩ table that will be made because of short verb characters will (at least partially) be based on which characters were made short verbs on the previous run.

`\SetCharProblematic`

The `\SetCharProblematic` command should be called by commands which make a character problematic (e.g. makes it active) in the general context (commands which make some character problematic only in some very special context, such as the `verbatim` environment, need not call `\SetCharProblematic`). The syntax is

`\SetCharProblematic{`⟨*code*⟩`}`

and it sets the "harm state" of the character whose code is ⟨*code*⟩ to problematic.

When `\SetCharProblematic` is called in the preamble, it sets the harm state on the current run. When it is called in the document body however, it sets the harm state on the next run by writing a `\SetHarmState` command to the `.aux` file. This is done to ensure that the contents of the `\XD@harmless@`⟨*code*⟩ table doesn't change during the body of a document.

```
\newcommand\SetCharProblematic[1]{%
   \SetHarmState\BooleanTrue{#1}\BooleanTrue
}
\AtBeginDocument{%
   \gdef\SetCharProblematic#1{%
      \if@filesw
         \immediate\write\@auxout{\string\SetHarmState
            \string\BooleanTrue {\number#1}\string\BooleanTrue}%
      \fi
   }%
}
```

`\add@specials`

`\MakeShortVerb`'s call to `\SetCharProblematic` is put in the `\add@specials` macro, which anyway already adds the character to the `\dospecials` and `\@sanitize` lists. Only familiar definitions of `\add@special` are changed.

```
\def\@tempa#1{%
  \rem@special{#1}%
  \expandafter\gdef\expandafter\dospecials\expandafter
    {\dospecials \do #1}%
  \expandafter\gdef\expandafter\@sanitize\expandafter
    {\@sanitize \@makeother #1}}
\ifx \@tempa\add@special
   \def\add@special#1{%
      \rem@special{#1}%
      \expandafter\gdef\expandafter\dospecials\expandafter
         {\dospecials \do #1}%
      \expandafter\gdef\expandafter\@sanitize\expandafter
         {\@sanitize \@makeother #1}%
      \SetCharProblematic{`#1}%
   }
\else
   \PackageWarningNoLine{xdoc2}{Unfamiliar definition of
      \protect\add@special;\MessageBreak the macro was not patched}
\fi
```

4 Indexing

Each type of index entry `doc` produces is implemented through a different indexing command.[5] This might be manageable when there are only `macro`s and `environment`s to distinguish between, but it soon gets unmanageable if more environments of this type are added. Therefore all `xdoc` index entries are made with a single command—`\IndexEntry`.

4.1 New basic indexing commands

`\IndexEntry` `\LevelSame` `\LevelSorted` `\XD@if@index`

The `\IndexEntry` command writes one index entry to the `.idx` file. It takes three arguments:

> `\IndexEntry{`⟨*entry text*⟩`}{`⟨*encap*⟩`}{`⟨*thenumber*⟩`}`

The ⟨*entry text*⟩ contains the text for the entry. It is a nonempty sequence of commands in which each item is one of

> `\LevelSame{`⟨*text*⟩`}`
> `\LevelSorted{`⟨*sort key*⟩`}{`⟨*text*⟩`}`

Each such item specifies one level of the entry that is to be written. In the case of `\LevelSorted`, the ⟨*text*⟩ is what will be written in the sorted index at that level and ⟨*sort key*⟩ is a key which the index-sorting program should use for sorting that entry at that level. In the case of `\LevelSame`, the ⟨*text*⟩ is used both as sort key and contents of entry in the sorted index. The first item is for the topmost level and each subsequent item is for the next sublevel. The ⟨*entry text*⟩ will be fully expanded by the `\IndexEntry` command.

⟨*thenumber*⟩ is the number (if any) that the index entry refers to. It can consist of explicit characters, but it can also be a `\the`⟨*counter*⟩ control sequence or a macro containing such control sequences. ⟨*thenumber*⟩ is fully expanded by the `\IndexEntry` command, with the exception for occurrences of `\thepage`—expansion of `\thepage` will instead be delayed until the page is shipped out, so that the page numbers will be right. **Note:** ⟨*thenumber*⟩ must not contain any formatting that will upset the index-sorting program. `doc`'s default definition of `\theCodelineNo` contains such formatting, so one must instead use `\thecodelineno` as ⟨*thenumber*⟩ in that case.

⟨*encap*⟩ is the name of the encapsulation scheme that should be applied to ⟨*thenumber*⟩. All encapsulation schemes that have been implemented instruct the index sorting program to wrap up ⟨*thenumber*⟩ in some code that gives it special formatting when the sorted index is written, but one could also use the ⟨*encap*⟩ to specify 'beginning of range' and 'end of range' index entries. `none` Use `none` as ⟨*encap*⟩ if you don't want any special formatting.

Note: `\IndexEntry` uses `\@tempa` internally, so you cannot use that in argument #2 or #3. Using it in argument #1 presents no problems, though.

```
\newcommand\IndexEntry[3]{%
   \@bsphack
   \begingroup
      \def\LevelSame##1{\levelchar##1}%
```

---

[5] Sometimes there are even more than one command per entry type—the `\SpecialIndex`, `\LeftBraceIndex`, `\RightBraceIndex`, and `\PercentIndex` commands all generate entries of the same type.

```
    \def\LevelSorted##1##2{\levelchar##1\actualchar##2}%
    \protected@edef\@tempa{#1}%
    \protected@edef\@tempa{\expandafter\@gobble\@tempa\@empty}%
    \@ifundefined{XD@idxencap@#2}{%
      \PackageError{xdoc2}{Index entry encap `#2' unknown}\@eha
    }{%
      \XD@if@index{%
        \csname XD@idxencap@#2\endcsname\@tempa{#3}%
      }{}%
    }%
  \endgroup
  \@esphack
}
```

`\IndexEntry` does (like `\index`) not contribute any material to the current list if indices aren't being made.

`\XD@if@index` is `\@firstoftwo` if index entries are being written and `\@secondoftwo` if they are not.

```
\let\XD@if@index=\@secondoftwo
```

In LaTeX 2_ε\*, the `\IndexEntry` command should probably be implemented using templates, e.g. the ⟨*encap*⟩s could be names of instances.

`\levelsame`
`\levelsorted`

These names were used for `\LevelSame` and `\LevelSorted` respectively in prototype version 2.0, but the macros should belong to the same capitalization class as `\IndexEntry` so their names were changed in prototype version 2.1. The old names `\levelsame` and `\levelsorted` will continue to work in xdoc2, though.

```
⟨*xdoc2⟩
\newcommand*\levelsame{\LevelSame}
\newcommand*\levelsorted{\LevelSorted}
⟨/xdoc2⟩
```

`\XD@idxencap@`⟨*encap*⟩

Macros in the family `\XD@idxencap@`⟨*encap*⟩ takes two arguments as follows

`\XD@idxencap@`⟨*encap*⟩ `{`⟨*entry*⟩`}` `{`⟨*thenumber*⟩`}`

They should write an entry with the ⟨*encap*⟩ encapsulation of the ⟨*thenumber*⟩ to the index file. They need not check whether index generation is on or not, but they must be subject to the LaTeX kernel `@filesw` switch. They must expand both arguments fully at the time of the command, with the exception for the control sequence `\thepage`, which should not be expanded until the page on which the write appears is output. Both these conditions are met if the macro is implemented using `\protected@write`.

`\XD@idxencap@none`
`\XD@idxencap@main`
`\XD@idxencap@usage`

These macros implement the encapsulation schemes that are used in doc.

```
\def\XD@idxencap@none#1#2{%
  \protected@write\@indexfile{}{\XD@index@keyword{#1}{#2}}%
}
```

```
\def\XD@idxencap@main#1#2{%
  \protected@write\@indexfile{}%
    {\XD@index@keyword{#1\encapchar main}{#2}}%
}
```

```
\def\XD@idxencap@usage#1#2{%
   \protected@write\@indexfile{}%
      {\XD@index@keyword{#1\encapchar usage}{#2}}%
}
```

`\XD@index@keyword` The `\XD@index@keyword` is a hook for changing the index entry keyword (the text that is put in front of every index entry in the `.idx` file). It is changed by e.g. the docindex package [2].

```
\@ifundefined{XD@index@keyword}{%
   \edef\XD@index@keyword{\@backslashchar indexentry}%
}{}
```

`\CodelineIndex`
`\PageIndex`
`\TheXDIndexNumber`

The `\CodelineIndex` and `\PageIndex` commands do the same things as in doc, but work with the xdoc internals instead of the doc ones. `\TheXDIndexNumber` is used as ⟨*thenumber*⟩ argument to `\IndexEntry` by all indexing commands that would have used `\special@index` in doc.

```
\renewcommand\CodelineIndex{%
   \makeindex
   \let\XD@if@index=\@firstoftwo
   \codeline@indextrue
   \def\TheXDIndexNumber{\thecodelineno}%
}
```

```
\renewcommand\PageIndex{%
   \makeindex
   \let\XD@if@index=\@firstoftwo
   \codeline@indexfalse
   \def\TheXDIndexNumber{\thepage}%
}
```

```
\def\TheXDIndexNumber{??}
```

4.2 Making good sort keys

A common nuisance in doc indices is that many macros are sorted by parts of the name that do not carry any interesting information. In the LaTeX kernel many macro names begin with a silent `@`, whereas the names of private macros in many packages (including this one) begin with some fixed abbreviation of the package name. Since such prefixes usually are harder to remember than the rest of the macro name, it is not uncommon that the index position one thinks of first isn't the one where the macro actually is put. Hence a mechanism for removing such annoying prefixes from the macro names might be useful, and that is presicely what is defined below.

The actual mechanism is based on having a set of macros called *operators* which operate on the harmless character string that is to become the sort key. Each operator has a specific prefix string which it tries to match against the beginning of the to-be sort key, and if they match then the prefix is moved to the end of the sort key. Automatically constructed operators (see below) have names of the form `\XD@operatorA@`⟨*prefix*⟩, but operators can be given arbitrary names.

`\XD@operatorA@`⟨*prefix*⟩

`\XD@operators@list` The `\XD@operators@list` macro contains the list of all currently active operators.

```
\let\XD@operators@list\@empty
```

The operators do all their work at expand-time. When an operator macro is expanded, it is in the context

⟨*operator*⟩ ⟨*subsequent operators*⟩ `\@firstofone` ⟨*sort key text*⟩ `\@empty`

There may not be any `\@empty`s or `\@firstofone`s amongst the ⟨*subsequent operators*⟩ or in the ⟨*sort key text*⟩. This should expand to

⟨*subsequent operators*⟩ `\@firstofone` ⟨*operated-on sort key text*⟩ `\@empty`

The purpose of the `\@firstofone` after the ⟨*subsequent operators*⟩ is to remove any spaces that some operator might have put in front of the sort key. This happens if the entire sort key text has been ignored by some operator.

`\MakeSortKey` The `\MakeSortKey` command is called to make the acutal sort key. The syntax of this command is

`\MakeSortKey{`⟨*macro*⟩`}{`⟨*text*⟩`}{`⟨*extras*⟩`}`

This locally defines ⟨*macro*⟩ to be the sort key that the currently active operators manufacture from ⟨*text*⟩. The ⟨*extras*⟩ argument can contain additional assignments needed for handling macros with special harmless forms, such as `\meta`.

```
\newcommand\MakeSortKey[3]{%
   \begingroup
      \def\PrintChar{\string\PrintChar\XD@threedignum}%
      #3%
      \unrestored@protected@xdef\@gtempa{#2}%
   \endgroup
   \protected@edef#1{%
      \expandafter\XD@operators@list \expandafter\@firstofone
      \@gtempa\@empty
   }%
}
```

`\XD@make@operator` The `\XD@make@operator` macro takes a harmless character sequence as argument, constructs the corresponding operator, and returns the operator control sequence in the `\toks@` token list register.

More precisely, given a harmless character string ⟨*string*⟩, `\XD@make@operator` will construct a sequence of other tokens ⟨*text*⟩ from ⟨*string*⟩ by replacing all `\PrintChar` commands in the same way as `\MakeSortKey` does. Then it defines the macro `\XD@operatorA@`⟨*text*⟩ to be

#1 `\@firstofone` #2 `\@empty` → `\XD@operatorB@`⟨*text*⟩
`\@firstofone` #2 `\@firstofone` ⟨*text*⟩ `\@firstofone` `\relax` #1
`\@empty`

and the macro `\XD@operatorB@`⟨*text*⟩ to do

$$\#1\,\texttt{\textbackslash @firstofone}\,\langle\textit{text}\rangle\,\#2\,\texttt{\textbackslash @firstofone}\,\#3\,\texttt{\textbackslash relax}\,\#4\,\texttt{\textbackslash @empty} \to$$
$$\#4\begin{cases}\texttt{\textbackslash @firstofone}\,\#2\,\text{␣}\langle\textit{text}\rangle\,\texttt{\textbackslash @empty} & \text{if } \#1 \text{ is empty}\\ \#1\,\texttt{\textbackslash @empty} & \text{otherwise}\end{cases}$$

```
\def\XD@make@operator#1{%
   \begingroup
      \def\PrintChar{\string\PrintChar\XD@threedignum}%
      \let\protect\@gobble
      \xdef\@gtempa{#1}%
   \endgroup
   \expandafter\edef \csname XD@operatorA@\@gtempa\endcsname
      ##1\@firstofone##2\@empty{%
      \expandafter\noexpand \csname XD@operatorB@\@gtempa\endcsname
      \noexpand\@firstofone ##2\noexpand\@firstofone \@gtempa
      \noexpand\@firstofone \relax##1\noexpand\@empty
   }%
   \expandafter\edef \csname XD@operatorB@\@gtempa \expandafter\endcsname
      \expandafter##\expandafter1\expandafter\@firstofone \@gtempa
      ##2\@firstofone##3\relax##4\@empty{%
      \noexpand\ifx $##1$%
         \noexpand\expandafter \noexpand\@firstoftwo
      \noexpand\else
         \noexpand\expandafter \noexpand\@secondoftwo
      \noexpand\fi{%
         ##4\noexpand\@firstofone ##2 \@gtempa
      }{##4##1}%
      \noexpand\@empty
   }%
   \toks@=\expandafter{\csname XD@operatorA@\@gtempa\endcsname}%
}
```

`\DoNotIndexBy`

The `\DoNotIndexBy` command has the syntax

`\DoNotIndexBy{`⟨*morpheme*⟩`}`

It causes the ⟨*morpheme*⟩ to be put *last* in the index sort key for each macro name which begins by ⟨*morpheme*⟩. This can be used to ignore e.g. "silent" `@`s at the beginning of a macro name.

```
\newcommand\DoNotIndexBy[1]{%
   \MakeHarmless\@tempa{#1}%
   \XD@make@operator\@tempa
   \expandafter\def \expandafter\XD@operators@list \expandafter{%
      \the\expandafter\toks@ \XD@operators@list
   }%
}
```

4.3 Reimplementations of doc indexing commands

The doc indexing commands aren't that interesting in xdoc, since they take 'raw' control sequences as arguments rather than the harmless strings that the xdoc commands will want to put in the index. But it can be instructive to see how they would be implemented in this context.

`\SortIndex`

The `\SortIndex` takes a sort key and an entry text as argument, and writes a one-level index entry for that.

```
\renewcommand*\SortIndex[2]{%
   \IndexEntry{\LevelSorted{#1}{#2}}{none}{\thepage}%
}
```

`\SpecialIndex`
`\SpecialMainIndex`
`\SpecialUsageIndex`

The `\SpecialIndex`, `\SpecialMainIndex`, and `\SpecialUsageIndex` commands take a control sequence (or more often something which looks like a `\string`ed control sequence) as their only argument. The entry text is that item verbatim, and the initial backslash is ignored in sorting (`\SpecialIndex` always ignores the first character regardless of whether it is a backslash or not, the other two checks first). `\SpecialIndex` has `none` formatting, `\SpecialMainIndex` has `main` formatting, and `\SpecialUsageIndex` has `usage` formatting of the index number.

Although these definitions will (or at least are supposed to) yield the same typeset results as the `doc` definitions in the mainstream cases, I doubt that they will do so in all cases. At any rate, they shouldn't perform worse.

```
\renewcommand\SpecialIndex[1]{%
   \expandafter\MakeHarmless \expandafter\@tempa
      \expandafter{\string#1}%
   \IndexEntry{%
      \LevelSorted{%
         \expandafter\XD@unbackslash \@tempa\@empty
      }{\texttt{\@tempa}}%
   }{none}{\TheXDIndexNumber}%
}
```

```
\renewcommand\SpecialMainIndex[1]{%
   \expandafter\MakeHarmless \expandafter\@tempa
      \expandafter{\string#1}%
   \IndexEntry{%
      \LevelSorted{%
         \expandafter\XD@unbackslash \@tempa\@empty
      }{\texttt{\@tempa}}%
   }{main}{\TheXDIndexNumber}%
}
```

```
\renewcommand\SpecialUsageIndex[1]{%
   \expandafter\MakeHarmless \expandafter\@tempa
      \expandafter{\string#1}%
   \IndexEntry{%
      \LevelSorted{%
         \expandafter\XD@unbackslash \@tempa\@empty
      }{\texttt{\@tempa}}%
   }{usage}{\thepage}%
}
```

`\XD@unbackslash`
`\XD@unbackslash@`

`\XD@unbackslash` is a utility macro which removes the first character from a harmless character string if that character is a backslash (i.e., if it is `\PrintChar{92}`). The `doc` commands have traditionally used `\@gobble` for doing this, but the `\@SpecialIndexHelper@` macro that was comparatively recently added tries to do better.

```
\def\XD@unbackslash#1{%
   \ifx \PrintChar#1%
      \expandafter\XD@unbackslash@
   \else
      \expandafter#1%
   \fi
}
\def\XD@unbackslash@#1{\ifnum #1=92 \else \PrintChar{#1}\fi}
```

`\SpecialMainEnvIndex`
`\SpecialEnvIndex`

These are similar to the above, but `doc` thinks that the arguments don't need any special care, and it produces two index entries per command. `\SpecialEnvIndex` should really have been called `\SpecialUsageEnvIndex`.

```
\renewcommand\SpecialMainEnvIndex[1]{%
   \IndexEntry{\LevelSorted{#1}{\texttt{#1} (environment)}}{main}%
      {\TheXDIndexNumber}%
   \IndexEntry{\LevelSame{environments:}\LevelSorted{#1}{\texttt{#1}}}%
      {main}{\TheXDIndexNumber}%
}
```

```
\renewcommand\SpecialEnvIndex[1]{%
   \IndexEntry{\LevelSorted{#1}{\texttt{#1} (environment)}}{usage}%
      {\thepage}%
   \IndexEntry{\LevelSame{environments:}\LevelSorted{#1}{\texttt{#1}}}%
      {usage}{\thepage}%
}
```

`\it@is@a`
`\XD@special@index`

The `\it@is@a` macro is a specialized version of `\SpecialIndex`, but the format of its argument is quite different. After full expansion the argument will become a single category 12 token ($\langle t\rangle$, say), and the control sequence for which an entry should be made is `\`$\langle t\rangle$. doc uses `\it@is@a` for control sequences with one-character names. Note: The following definition should really have special code for the ⟨economical⟩ docstrip module, but I don't think that is necessary since the doc macros which used `\it@is@a` will be redefined so that they don't.

`\XD@special@index` does the same thing as `\SpecialIndex`, but it does it with xdoc datatypes—the argument must be a harmless character string that does not include the initial escape (backslash).

```
\def\it@is@a#1{%
   \edef\@tempa{#1}%
   \XD@special@index{\csname XD@harmless@\number
      \expandafter`\@tempa\endcsname}%
}
```

```
\def\XD@special@index#1{%
   \MakeSortKey\@tempa{#1}{}%
   \IndexEntry{\LevelSorted{\@tempa}{\texttt{\Bslash#1}}}{none}%
      {\TheXDIndexNumber}%
}
```

`\LeftBraceIndex`
`\RightBraceIndex`
`\PercentIndex`
`\OldMakeIndex`

More specialised forms of `\SpecialIndex`. The `\OldMakeIndex` command can safely be made a no-op.

```
\renewcommand\LeftBraceIndex{\XD@special@index{\PrintChar{123}}}
\renewcommand\RightBraceIndex{\XD@special@index{\PrintChar{125}}}
\renewcommand\PercentIndex{\XD@special@index{\PrintChar{37}}}
\let\OldMakeIndex\relax
```

`\@wrindex`

Finally, while we're at redefining indexing commands, let's redefine `\@wrindex` as well to ensure that the index entry keyword is the same for all indexing commands.

```
\def\@wrindex#1{%
   \protected@write\@indexfile{}{\XD@index@keyword{#1}{\thepage}}%
   \endgroup
   \@esphack
}
```

5 Cross-referencing

5.1 Scanning `macrocode` for TEX control sequences

The cross-referencing mechanism in doc isn't problematic in the same way as the indexing mechanism is, so one could pretty much leave it as it is, but there are things that are better done differently when the basic indexing commands are based on harmless character strings. Rather than storing control sequence names (without escape character) as sequences of category 11 tokens, they will be stored as the equivalent harmless character strings.

`\macro@switch` As in doc, `\macro@switch` determines whether the control sequence name that follows consists of letters (call `\macro@name`) or a single non-letter (call `\short@macro`). Unlike doc, xdoc accumulates the characters from a multiple-letter control sequence name in a token register (`\@toks`), which is why that is cleared here.

```
\def\macro@switch{%
   \ifcat\noexpand\next a%
      \toks@={}%
      \expandafter\macro@name
   \else
      \expandafter\short@macro
   \fi
}
```

`\scan@macro` Since `\macro@namepart` isn't used as in doc, I might as well remove the command that cleared it from `\scan@macro`.

```
\def\scan@macro{%
   \special@escape@char
   \step@checksum
   \ifscan@allowed
      \def\next{\futurelet\next\macro@switch}%
   \else \let\next\@empty \fi
   \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. It will produce an index entry for that macro, unless that macro has been excluded from indexing, and it will also typeset the character that constitutes the name of the macro.

```
\def\short@macro#1{%
   \protected@edef\macro@namepart{%
⟨*economical⟩
      \ifnum `#1<\@xxxii
         \noexpand\PrintChar{\number`#1}%
      \else\ifnum `#1>126
         \noexpand\PrintChar{\number`#1}%
      \else
⟨/economical⟩
         \csname XD@harmless@\number`#1\endcsname
⟨economical⟩        \fi\fi
   }%
   \ifnot@excluded \XD@special@index{\macro@namepart}\fi
```

The cross-referencing mechanism is disabled for when the actual character is printed, as it could be the escape character. The index entry must be generated before the character is printed to ensure that no page break intervenes (recall that a `^^M` will start a new line).

```
   \scan@allowedfalse #1\scan@allowedtrue
}
```

There is one mechanism in `\TeX`'s control sequence tokenization that `\short@macro` doesn't cover, and that is the `^^` sequence substitution—`\^^M` is (with default catcodes) seen as the three tokens `\^`, `^`, and `M`, not as the single control sequence token that TeX will make out of it. But this is the way it is done in doc.

`\macro@name`
`\more@macroname`
`\macro@finish`

Then there's the macros for assembling a control sequence name which consists of one or more letters (category 11 tokens). (This includes both the characters which are normally letters in the document and those that are made letters by `\MakePrivateLetters`.) They're pretty straightforward.

```
\def\macro@name#1{%
⟨*economical⟩
   \if \ifnum `#1<\@xxxii 1\else \ifnum `#1>126 1\else 0\fi\fi 1%
      \toks@=\expandafter{\the\expandafter\toks@
         \expandafter\PrintChar \expandafter{\number`#1}%
      }%
   \else
⟨/economical⟩
   \toks@=\expandafter{\the\expandafter\expandafter\expandafter\toks@
      \csname XD@harmless@\number`#1\endcsname}%
⟨economical⟩   \fi
   \futurelet\next\more@macroname}
```

```
\def\more@macroname{%
   \ifcat\noexpand\next a%
      \expandafter\macro@name
   \else
      \macro@finish
   \fi
}
```

```
\def\macro@finish{%
   \edef\macro@namepart{\the\toks@}%
   \ifnot@excluded \XD@special@index{\macro@namepart}\fi
   \macro@namepart
}
```

5.2 The index exclude list

The index exclude list mechanisms are not quite as simple to convert for use with harmless character strings as the construction of macro names are. This is because the trick used for searching the exclude list for a certain string doesn't work if the string one is looking for contains tokens with category 1 or 2 (beginning and end of group), as the ⟨*parameter text*⟩ of a `\def` cannot contain such tokens. On the other hand the only groups that can appear in the harmless character strings one will be looking for are the ones around the argument of some `\PrintChar`, and these can easily be converted to something else. Therefore an item in the index exclude list of xdoc will have the format

`\do` ⟨*string*⟩

where the ⟨*string*⟩ is different from a harmless character string only in that all `\PrintChar{`⟨*num*⟩`}` have been replaced by `\PrintChar(`⟨*num*⟩`)`. The ⟨*string*⟩ does not include an escape character. The `\do` serves only to separate the item from the one before, but it could in principle be used for other purposes as well (such as in typesetting the entire exclude list).

`\XD@paren@PrintChar`

`\XD@paren@PrintChar` is a definition of `\PrintChar` which, when it is used in an `\edef`, merely replaces the group around the argument by a parenthesis and normalizes the number in the argument.

```
\def\XD@paren@PrintChar#1{\noexpand\PrintChar(\number#1)}
```

`\DoNotIndex`
`\do@not@index`
`\XD@do@not@index`

These are the macros which add elements to the index exclude list. `\DoNotIndex` is pretty much as in `doc`, but I have added resetting of the catcodes of ‘`,`’ (since `\XD@do@not@index` relies on it) and ‘`#`’ (since it can otherwise mess things up for the `\def\@tempa` in `\do@not@index`).

```
\renewcommand\DoNotIndex{%
   \begingroup
      \MakePrivateLetters
      \catcode`\#=12\catcode`\\=12\catcode`,=12\catcode`\%=12
   \expandafter\endgroup \do@not@index
}
```

`\do@not@index`, on the other hand, is quite different, as it more or less has to convert the argument from the format used in `doc` to that of `xdoc`. The bulk of the work is done by `\XD@do@not@index`, which grabs one of the elements in the argument of `\do@not@index` and converts it (minus the initial backslash) to a harmless character string. That harmless character string is then converted by `\XD@paren@PrintChar`, so that the string can be searched for using `\expanded@notin`.

The reason for using a special loop structure here, as opposed to using for example `\@for`, is that one cannot use either of `\` or `,` alone as item separators, as they may both be part of control sequence names (consider for example `\DoNotIndex{\a,\\,\b,\,,\c}`), but they should be sufficient when combined.

The reason for storing new elements in `\toks@` until the end of the loop and only then inserting them into the index exclude list is speed; the index exclude list can get rather large, so you don't want to expand it more often than you have to. I don't know if the difference is noticeable, though.

```
\begingroup
   \catcode`\|=0
   \catcode`\,=12
   \catcode`\\=12

   |gdef|do@not@index#1{%
      |def|@tempa{#1}%
      |ifx |@empty|@tempa |else
         |toks@={}%
         |expandafter|XD@do@not@index |@gobble #1,\|XD@do@not@index,\%
      |fi
   }
   |gdef|XD@do@not@index#1,\{%
      |ifx |XD@do@not@index#1%
```

```
         |index@excludelist=|expandafter{%
            |the|expandafter|index@excludelist |the|toks@
         }%
         |expandafter|@gobble
      |else
         |MakeHarmless|@tempa{#1}%
         |begingroup
            |let|PrintChar|XD@paren@PrintChar
            |unrestored@protected@xdef|@gtempa{|noexpand|do|@tempa}%
         |endgroup
         |toks@=|expandafter{|the|expandafter|toks@ |@gtempa}%
      |fi
      |XD@do@not@index
   }
|endgroup
```

`\DoNotIndexHarmless` The `\DoNotIndexHarmless` command takes a harmless character string as argument and locally adds the control sequence whose name is that character string to the index exclude list.

```
\newcommand\DoNotIndexHarmless[1]{%
   \begingroup
      \let\PrintChar\XD@paren@PrintChar
      \unrestored@protected@xdef\@gtempa{\noexpand\do#1}%
   \endgroup
   \index@excludelist=\expandafter{%
      \the\expandafter\index@excludelist \@gtempa
   }%
}
```

`\index@excludelist` In case the index exclude list is not empty, its contents are converted to `xdoc` format.

```
\edef\@tempa{\the\index@excludelist}
\index@excludelist{}
\ifx \@tempa\@empty \else
   \def\@tempb#1,\@nil{\do@not@index{#1}}
   \expandafter\@tempb \@tempa \@nil
   \let\@tempa\@empty
   \let\@tempb\@empty
\fi
```

The fact that the `\XD@harmless@`⟨*code*⟩ table has not yet reached its final form means that some of these control sequences listed in the exclude list might get a different form here than they actually should, but there isn't much that can be done about that. It is furthermore unusual that control sequence are given such names that they would be affected by this.

`\ifnot@excluded` The `\ifnot@excluded` macro ultimately boils down to an `if`, which evaluates to true if and only if the string in `\macro@namepart` is not one of the items in the index exclude list. Before `\expanded@notin` gets to carry out the actual test, the `\PrintChar` calls in `\macro@namepart` are converted by `\XD@paren@PrintChar` (it's OK to use an unprotected `\edef` for this, since `\PrintChar` is the only control sequence that can appear in `\macro@namepart`) so that `\expanded@notin` can be used to test for its presence.

```
\def\ifnot@excluded{%
   \begingroup
      \let\PrintChar\XD@paren@PrintChar
      \edef\@tempa{\macro@namepart}%
   \expandafter\endgroup \expandafter\expanded@notin
      \expandafter{\expandafter\do \@tempa\do}%
      {\the\index@excludelist}%
}
```

5.3 External cross-referencing

(This subsection is a bit speculatory, but I think the structures it describes may come in handy.)

It's rather easy to write macros for scanning TEX code for the names of control sequences—just look for the escape (category 0) character, and whatever follows is the name of a control sequence. Doing the same thing for other languages may lay anywhere between "a tricky exercise in advanced TEX programming" and "possible in theory",[6] but in most cases the available solutions turn out to be too complicated and/or slow to be of practical use. When that happens, one might instead want to use some external piece of software for doing the cross-referencing.

The commands in this subsection implement basic support for such an external cross-referencing program (or XXR,[7] for short). The idea is that an XXR should communicate with LATEX like BIBTEX does—scan the `.aux` file (or files, if we're `\include`ing things) for certain "commands" and use them to locate the files to cross-reference, get parameter settings (like for example entries for the index exclude list), and so on. It should then cross-reference the file(s) and write the index entries in a suitable format to some file (appending them to the `.idx` file is probably the easiest solution). This way, it is (almost) as simple to use as the built-in cross-referencing and the extra work for supporting it is (in comparison to not supporting it) negligible.

`xternalXRefMsg` XXR-command
`\SendExternalXRefMsg`

It's hardly possible to predict all kinds of information that one might want to give to an XXR, and neither can one assume that there is only one XXR program that will read the `.aux` file. A complicated project might involve code in several languages, and each language might have its own XXR. Therefore the general XXR-command (text in an `.aux` file which is used for communicating information to an XXR) simply has the syntax

`%%ExternalXRefMsg␣{`⟨*who*⟩`}␣{`⟨*what*⟩`}`

⟨*who*⟩ identifies the XXR this message is meant for. It must be balanced text to TEX and may not contain any whitespace, but can otherwise be rather arbitrary. ⟨*what*⟩ is the actual message. It too must be balanced text to TEX and it may not contain any newlines, but it is otherwise arbitrary. The reason for these restrictions on the contents of ⟨*who*⟩ and ⟨*what*⟩ is that many (maybe even most) scripting languages (which is what at least the `.aux`-scanning part of an XXR will probably be written in) are much better at recognising words on a line than they are at

---

[6] I.e., you know it can be implemented as a computer program (in some language), you know that any computer program can be translated to a Turing machine (or if you prefer that, expressed in lambda calculus), and you know that a Turing machine can be emulated by TEX, but that's the closest thing to a solution you've managed to come up with.

[7] Maybe not the most logical name, but it looks much cooler than ECR.

recognising a brace-delimited group. By accepting these restrictions, one can make sure that all XXRs can correctly determine whether a message is for them, even if they see the `.aux` file as a sequence of lines composed of whitespace-delimited words.

`\SendExternalXRefMsg` is the basic command for writing `ExternalXRefMsg`s to the `.aux` file, but it might be recommendable that XXR writers provide users with a set of commands that have more specific purposes. The syntax of the `\SendExternalXRefMsg` command is (hardly surprising)

`\SendExternalXRefMsg{`⟨*who*⟩`}{`⟨*what*⟩`}`

`\SendExternalXRefMsg` does a protected full expansion (like `\protected@edef`) of its arguments at the time it is called.

```
\newcommand\SendExternalXRefMsg[2]{%
   \begingroup
      \if@filesw
         \let\protect\@unexpandable@protect
         \immediate\write\@auxout{\@percentchar\@percentchar
            ExternalXRefMsg {#1} {#2}}%
      \fi
   \endgroup
}
```

The remaining commands in this subsection address complications that exist because of how `.dtx` files are generally written, and thus constitutes difficulties that all XXRs will have to face.

`ternalXRefFile` XXR-command

The usual way to write `.dtx` files is to include a driver—a short piece of uncommented LaTeX code which contains the necessary preamble material and a document body which mainly contains a `\DocInput` for the `.dtx` file itself—but it is also usually understood that this driver may be copied to another file if necessary and larger projects usually have a completely separate driver file. Therefore an XXR cannot be expected to be able to find the file(s) to cross-reference simply by changing suffix on the name of the `.aux` file it reads its commands from. A more intricate method must be used.

To tell the XXR that “here I input the file ...”, one includes an `ExternalXRefFile` XXR-command in the `.aux` file. Its syntax is

`%%ExternalXRefFile␣{`⟨*cmd*⟩`}␣{`⟨*file*⟩`}␣{`⟨*what*⟩`}`

⟨*file*⟩ is the name (as given to `\input` or the like) of the file to input. ⟨*cmd*⟩ is either `begin` (begin of ⟨*file*⟩) or `end` (end of ⟨*file*⟩). ⟨*what*⟩ is a declaration of what is in the file; XXRs should use it to determine whether they should process this file or not. ⟨*what*⟩ is empty if all XXRs should process the file, but for example `\IndexInput` will put `TeX` here to declare that the contents of this file are TeX code and only XXRs that cross-reference TeX code need to process this file.

In connection to this, it should be mentioned that XXRs must also look for (and act on) `\@input{`⟨*auxfile*⟩`}` commands that `\include` or `\DocInclude` has written to the `.aux` file, since these ⟨*auxfile*⟩s can also contain commands for the XXR that should result in output to the same `.idx` file. In particular, the `ExternalXRefFile` XXR-commands that are written because of a `\DocInclude` will be written to such an ⟨*auxfile*⟩.

`ternalXRefSync` XXR-command

Most XXRs will probably find it an unreasonable task to keep exact track of all codelines in all documents, i.e., they will sometimes think that a piece of code contains more or fewer numbered codelines than it actually does. If for example a document contains code such as

```
% \iffalse
%    \begin{macrocode}
Etaoin Shrldu
%    \end{macrocode}
% \fi
```

then all reasonable XXRs will probably be fooled into thinking that the `Etaoin Shrldu` line is a numbered codeline. This would of course be very bad if an XXR thought it should cross-reference the contents of this line, but that shouldn't usually be a problem since the specifications[8] of what code should be cross-referenced will probably make it clear that the above line should not be cross-referenced. Code such as the above will still be problematic however, as it will cause the XXR to believe that the `codelineno` counter has another value on any following line that is indexed than it actually has in the typeset document. This will cause index entries to refer to another line than it actually should.

To overcome this, the `ExternalXRefSync` XXR-command can be used to tell the XXR what the corresponding values of `\inputlineno` and `codelineno` are. Its syntax is

`%%ExternalXRefSync␣{`⟨*inputlineno*⟩`}␣{`⟨*codelineno*⟩`}`

where ⟨*inputlineno*⟩ is the expansion of `\the\inputlineno` and ⟨*codelineno*⟩ is the expansion of `\thecodelineno`, both expanded at the same point in the program. Note here that the first line of a file is line number 1, that line number 0 is used to denote "just before the first line", and that `codelineno` gets increased immediately before the number is typeset (i.e., `codelineno` contains the number of the last numbered codeline).

This doesn't support external cross-referencing by pages, since doing that requires that the document outputs a lot more information to the `.aux` file. In principle, one could put a `\mark{\thecodelineno}` in `\PrintCodelineNo` and a `\write` in the page header which outputs to the `.aux` file which range of codelines correspond to a given page, but the LaTeX 2ε sectioning commands' use of marks tends to interfere with this. The LaTeX 2ε∗ package xmarks will probably solve that problem, though.

`\syncexternalxref`

The `\syncexternalxref` command writes an `ExternalXRefSync` XXR-command for the current line number and value of the `codelineno` counter to `.aux` file. It is used for synchronizing the numbered codeline counter that an XXR maintains with the `codelineno` counter that is used for numbering codelines in the typeset document after a piece of code in the document that some XXR is likely to misinterpret. `\syncexternalxref` shouldn't be used inside `macrocode` environments (or the like) as they tend to read ahead in the file—instead it is best placed shortly after such an environment. `\syncexternalxref` has no arguments.

```
\newcommand\syncexternalxref{%
```

---

[8] I imagine these specifications will consist of a list of docstrip options (modules), possibly used in combination with restrictions on the names of surrounding environments.

```
   \if@filesw
      \immediate\write\@auxout{\@percentchar\@percentchar
         ExternalXRefSync {\the\inputlineno} {\thecodelineno}%
      }%
   \fi
}
```

`ternalXRefWrap` XXR-command

The `\DocInclude` command complicates matters for XXRs by redefining things so that the `codelineno` counter only makes up a part of the line numbers appearing in the index. The purpose of the `ExternalXRefWrap` XXR-command is to inform XXRs about such changes. The command

`%%ExternalXRefWrap␣{`⟨*prefix*⟩`}␣{`⟨*suffix*⟩`}`

means that codeline numbers written to the index should have the form

⟨*prefix*⟩⟨*codelineno*⟩⟨*suffix*⟩

This setting takes effect from the next `ExternalXRefSync` and stays in effect until the end of the document or until another `ExternalXRefWrap` overrides it. The state at the beginning of the document is to have both ⟨*prefix*⟩ and ⟨*suffix*⟩ empty.

`\XD@input`

The `\XD@input` command is a version of `\input` which takes care to inform XXRs that another file is being `\input`ted. Its syntax is

`\XD@input{`⟨*file*⟩`}{`⟨*what*⟩`}`

where ⟨*file*⟩ is the name of the file to `\input` and ⟨*what*⟩ is the contents of the file, as specified in `ExternalXRefFile` commands.

```
\def\XD@input#1#2{%
   \if@filesw
      \immediate\write\@auxout{\@percentchar\@percentchar
         ExternalXRefFile {begin} {#1} {#2}%
      }%
      \immediate\write\@auxout{\@percentchar\@percentchar
         ExternalXRefSync {0} {\thecodelineno}%
      }%
   \fi
   \input{#1}%
   \if@filesw
      \immediate\write\@auxout{\@percentchar\@percentchar
         ExternalXRefFile {end} {#1} {#2}%
      }%
      \immediate\write\@auxout{\@percentchar\@percentchar
         ExternalXRefSync {\the\inputlineno} {\thecodelineno}%
      }%
   \fi
}
```

`\DocInput`

The `\DocInput` command is redefined so that it writes `ExternalXRefFile` and `ExternalXRefSync` XXR-commands to the `.aux` file. Furthermore, with `xdoc` one should always use the `\DocInput` command (or some command based on it, like `\DocInclude`) for inputting a file where percent is an ‘ignore’ character—even when one such file inputs another. (Doing that didn’t work with the `doc` definition, as it always called `\MakePercentComment` upon return, but the xdoc definition contains code for dealing with that.)

```
\renewcommand\DocInput[1]{%
   \relax
   \ifnum \catcode`\%=14
      \expandafter\@firstoftwo
   \else
      \expandafter\@secondoftwo
   \fi{%
      \MakePercentIgnore\XD@input{#1}{}\MakePercentComment
   }{\XD@input{#1}{}}%
}
```

`\IndexInput` The `\IndexInput` command also needs to be redefined to write XXR-commands to the `.aux` file. It would probably be enough here to write an `ExternalXRefSync` after the file has been `\input` since no external cross-referencing of `\IndexInput`ted files is needed, but I do the more verbose variant here just to exemplify how these things would look for other languages.

```
\renewcommand\IndexInput[1]{%
   \begingroup
      \macro@code
      \frenchspacing
      \@vobeyspaces
      \XD@input{#1}{TeX}%
      \endmacrocode
   \endgroup
}
```

6 Two-sided printing

The main problem one faces when reimplementing doc so that the marginal material always appears in the outer margin in two-sided documents is that the justification of doc's marginal material is asymmetric; it always extends outwards. This means that the justification to use when typesetting the marginal material must depend on whether it is to be put on a left or a right page—something which cannot be determined for sure when the material is typeset! This is a minor difficulty if the marginal material is put in place using LaTeX's `\marginpar` command, as that allows the user to supply different versions of the marginal paragraph for left and right margin placements. It is however a major difficulty if the marginal material is displaced out into the margin from within the main galley (like the `macro` environment of doc does), since the output routine is never involved.

Even though this difficulty provides arguments for using a `\marginpar` mechanism for all text that is put in the margin, that will not be done in xdoc (but maybe it will in some successor). Instead xdoc contains a general mechanism which uses data written to the `.aux` file for determining whether a piece of text was put on an odd or even numbered page the *last* time the document was typeset. By the usual convergence of page breaks in a LaTeX document, this will eventually produce a typeset document with the marginal material consistently in the outer margin.

The mechanism works as follows. The places in the document (the document source) at which it is necessary to determine whether something is going to appear on an even (left) or an odd (right) page are called "page situations"[9] or just "situations". In each situation, a relatively simple test (is the `page` counter currently even or odd?) which is right more often than not is used as a first guess, and both the guess, the placement actually used, and the correct answer (determined from the value of `page` when the piece of text is shipped out) are recorded in the `.aux` file. If the guess (for the current situation) coincided with the correct answer the last time the document was typeset then the guess determined now is used, otherwise the opposite of the guess determined now is used. Finally, when at `\end{document}` the `.aux` file is inputted to check for changed labels, the placements used are also checked and the user is given a suitable warning if there was an incorrect one.

`\IfOddPageSituation` The `\IfOddPageSituation` macro is the user level test for whether the current page situation appears on an odd or an even page. It has the syntax

`\IfOddPageSituation{`⟨*odd*⟩`}{`⟨*even*⟩`}`

and this will expand to ⟨*odd*⟩ if the current situation is expected to end up on an odd page (based on how correct it was to look at the value of `page` last time) and to ⟨*even*⟩ otherwise. In single-sided mode, it always expands to ⟨*even*⟩. In two-sided mode, `\IfOddPageSituation` is redefined for the new situation each time `\StepPageSituation` is called.

```
\let\IfOddPageSituation=\@secondoftwo
```

`\StepPageSituation`
`\macro@cnt`
`\XD@next@wrong`
`\XD@wrongs@list`

The `\StepPageSituation` command is called to inform the page situation mechanism that a new situation has begun. The rule for when you need to use `\StepPageSituation` is simple: if you use `\IfOddPageSituation` in two places which may end up on different pages, then there must be a `\StepPageSituation` between them. There is no code which automatically calls `\StepPageSituation`—not even `\clearpage` or other macros which force page breaks do this—hence macros which use the page situation mechanism must always call `\StepPageSituation` explicitly when a new situation begins.

Since the `\macro@cnt` count register isn't used for stacking marginal headings ("macro" names) anymore (see below), it is employed for enumerating page situation. `\XD@next@wrong` is a macro which contains the number of the next situation in which the guess was wrong last time. Unless `\XD@next@wrong` = `\macro@cnt`, the guess was right last time. All assignments to `\macro@cnt` and `\XD@next@wrong` are global.

`\XD@wrongs@list` is a list of all the wrong guesses. It has the syntax

`\@elt{`⟨*guess no.*⟩`}\@elt{`⟨*guess no.*⟩`}`...`\@elt{`⟨*guess no.*⟩`}`

where the ⟨*guess no.*⟩s are the numbers of the wrong guesses, in increasing order. The contents of `\XD@wrongs@list` are collected when the `.aux` file is inputted at `\begin{document}`, and they are removed again as TeX passes the situation in the document that they apply to. All assignments to `\XD@wrong@list` are global.

Calling `\StepPageSituation` increases `\macro@cnt` by one, updates `\XD@next@wrong` and `\XD@wrong@list` appropriately, and sets `\IfOddPageSituation` to `\@firstoftwo` or `\@secondoftwo` (whichever is correct for this situation). `\@next` is a list management macro from the LaTeX kernel.

---

[9] I know it's not a particularly good name. Suggestions for better names are gracefully accepted.

```
\if@twoside
   \def\StepPageSituation{%
      \global\advance \macro@cnt \@ne
      \ifnum \XD@next@wrong<\macro@cnt
         \global\@next\XD@next@wrong\XD@wrongs@list{}{%
            \let\XD@next@wrong\maxdimen
         }%
      \fi
      \ifnum \ifodd\c@page -\fi \@ne=%
         \ifnum \XD@next@wrong=\macro@cnt -\fi \@ne
         \global\let\IfOddPageSituation\@secondoftwo
      \else
         \global\let\IfOddPageSituation\@firstoftwo
      \fi
   }
   \def\XD@next@wrong{-\maxdimen}
   \let\XD@wrongs@list\@empty
\else
   \let\StepPageSituation=\relax
\fi
```

`\RecordPageSituation`

The `\RecordPageSituation` command generates a `\write` whatsit node which records the outcome of the current page situation. It is the location of this whatsit node that determines on which page a certain situation is considered to occur. If you don't execute this macro for a certain page situation, the first guess will always be used for that situation and no warnings will be given if that guess is incorrect. In single-sided mode, this is a no-op (thus you should better place it somewhere where it doesn't affect spacing). Furthermore you must make sure that TeX does not change the value of the `page` counter between a `\StepPageSituation` and its corresponding `\RecordPageSituation`, since the `\ifodd` test must yield the same result in both cases.

```
\if@twoside
   \def\RecordPageSituation{%
      \if@filesw
         \edef\@tempa{%
            \string\XD@situation{\the\macro@cnt}{%
               \ifodd\c@page 1\else 0\fi
            }{\IfOddPageSituation{1}{0}}%
         }%
         \write\@auxout\expandafter{\@tempa{\ifodd\c@page 1\else 0\fi}}%
      \fi
   }%
\else
   \let\RecordPageSituation=\relax
\fi
```

`\XD@situation`
`\XD@check@situation`

`\XD@situation` is the command that will be written to the `.aux` file with the data about how the situation turned out. Its syntax is

`\XD@situation{`⟨*number*⟩`}{`⟨*guess*⟩`}{`⟨*did*⟩`}{`⟨*correct*⟩`}`

where ⟨*number*⟩ is the number of the situation, and ⟨*guess*⟩, ⟨*did*⟩, and ⟨*correct*⟩ describe what the guess, the actual action done, and what the correct action to do respectively was. ⟨*guess*⟩, ⟨*did*⟩, and ⟨*correct*⟩ are either `0` (denoting even page) or `1` (denoting odd page).

The definition for `\XD@situation` set here is the one which will be in force when the `.aux` file is inputted at `\begin{document}`; its purpose is to build the `\XD@wrongs@list`. `\XD@check@situation` is the definition for `\XD@situation` which will be in force when the `.aux` file is inputted at `\end{document}`; its purpose is to check if anything was incorrectly placed.

The main problem `\XD@situation` has to face is that text in the `.dvi` file needs not appear in exactly the same order as it was typeset, and it is therefore possible that `\XD@situation`s in the `.aux` file do not appear in increasing ⟨*number*⟩ order. Because of this, `\XD@situation` must sort the `\XD@wrongs@list` while constructing it. The only reasonable algorithm for this seems to be insertion sort, but as the items to insert are almost surely almost sorted, a special check is done in the beginning to see if that is the case. `\XD@next@wrong` is used in this to store the number of the last item so far inserted into the `\XD@wrongs@list`. By only assigning `\XD@next@wrong` locally here, one is relieved of having to reset it in `\AtBeginDocument` code.

When sorting is actually applied, a new item `\@elt{`⟨*insert*⟩`}` is inserted through expanding the list. When doing that, the `\@elt` macro has the syntax

> `\@elt` ⟨*flag*⟩ `{`⟨*number*⟩`}` ⟨*next*⟩

where ⟨*flag*⟩ is `\BooleanTrue` or `\BooleanFalse`, ⟨*number*⟩ is the item that the `\@elt` belong to, and ⟨*next*⟩ is either the next `\@elt` or `\@gobble` (if this is the last). The ⟨*flag*⟩ specifies whether the item has been inserted; `\BooleanTrue` means that it has. The above `\@elt`-sequence will expand to

> `\noexpand\@elt{`⟨*number*⟩`}` ⟨*next*⟩ `\BooleanTrue`

if ⟨*flag*⟩ is `\BooleanTrue`, or ⟨*flag*⟩ is `\BooleanFalse` and ⟨*number*⟩ is equal to ⟨*insert*⟩. It will expand to

> `\noexpand\@elt{`⟨*number*⟩`}` ⟨*next*⟩ `\BooleanFalse`

if ⟨*flag*⟩ is `\BooleanFalse` and ⟨*number*⟩ is less than ⟨*insert*⟩. It expands to

> `\noexpand\@elt{`⟨*insert*⟩`}` `\noexpand\@elt{`⟨*number*⟩`}`
> ⟨*next*⟩ `\BooleanTrue`

if ⟨*flag*⟩ is `\BooleanFalse` and ⟨*number*⟩ is greater than ⟨*insert*⟩.

```
\if@twoside
   \def\XD@situation#1#2#3#4{%
      \if #2#4\else
         \ifnum #1<\XD@next@wrong
            \begingroup
               \def\@elt##1##2##3{%
                  \noexpand\@elt
                  \ifcase
                     \ifx ##1\BooleanTrue 0%
                     \else\ifnum ##2<#1 1%
                     \else\ifnum ##2>#1 2%
                     \else 0%
                     \fi\fi\fi
                  \space
                     {##2}\expandafter\@secondoftwo
                  \or
                     {##2}\expandafter\@firstoftwo
                  \else
                     {#1}\noexpand\@elt{##2}\expandafter\@secondoftwo
                  \fi{##3\BooleanFalse}{##3\BooleanTrue}%
               }%
               \xdef\XD@wrongs@list{%
                  \expandafter\expandafter \expandafter\@elt
                  \expandafter\@firstoftwo \expandafter\BooleanFalse
                  \XD@wrongs@list \@gobble
               }%
            \endgroup
         \else\ifnum #1>\XD@next@wrong
            \def\XD@next@wrong{#1}%
            \expandafter\gdef \expandafter\XD@wrongs@list
               \expandafter{\XD@wrongs@list \@elt{#1}}%
         \fi\fi
      \fi
   }
   \def\XD@check@situation#1#2#3#4{%
      \if #3#4\else
         \PackageWarningNoLine{xdoc2}{Page breaks may have changed.%
            \MessageBreak Rerun to get marginal material right}%
         \let\XD@situation\@gobblefour
      \fi
   }
   \AtBeginDocument{\global\let\XD@situation\XD@check@situation}
\else
   \let\XD@situation\@gobblefour
\fi
```

`\XD@set@situation`
`\XD@write@situation@ckpt`
`\cl@@ckpt`

The page situation counter `\macro@cnt` is closely related to the `page` counter and it needs to be among the counters whose values are recorded in `\include` checkpoints, since the enumeration of situations will otherwise change when files are added to or removed from the `\@partlist`. It is not sufficient to simply set the value of `\macro@cnt` however; one must also advance to the correct position in the `\XD@wrongs@list` list and set `\XD@next@wrong` accordingly. The `\XD@set@situation` command has the syntax

> `\XD@set@situation{`⟨*number*⟩`}`

It sets `\macro@cnt` to ⟨*number*⟩ and updates `\XD@wrongs@list` and `\XD@next@wrong` accordingly.

```
\if@twoside
   \def\XD@set@situation#1{%
      \global\macro@cnt=#1\relax
      \loop \ifnum \XD@next@wrong<\macro@cnt
         \global\@next\XD@next@wrong\XD@wrongs@list{}{%
            \let\XD@next@wrong\maxdimen
         }%
      \repeat
   }
```

```
\else \let\XD@set@situation=\@gobble \fi
```

The `\XD@write@situation@ckpt` macro writes an `\XD@set@situation` command to the `.aux` file in the way that `\@wckptelt` writes `\setcounter` commands for normal counters. A problem for `\XD@write@situation@ckpt` is that it will have to appear in a macro which is regularly subjected to the `\xdef` in `\@cons`. For that reason, it will simply expand to itself whenever `\@elt` isn't `\@wckptelt`.

```
\if@twoside
   \def\XD@write@situation@ckpt{%
      \ifx \@elt\@wckptelt
         \immediate\write\@partaux{%
            \string\XD@set@situation{\the\macro@cnt}%
         }%
      \else
         \noexpand\XD@write@situation@ckpt
      \fi
   }
   \expandafter\def \expandafter\cl@@ckpt
      \expandafter{\cl@@ckpt \XD@write@situation@ckpt}
\fi
```

7 The list of changes

Reimplementations elsewhere have required a few modifications related to the `\changes` command. There are a lot of other things that could and perhaps should be done with these mechanisms, though.

`\saved@macroname` The contents of the `\saved@macroname` macro now have the syntax

`{`⟨*sort key*⟩`}{`⟨*text*⟩`}`

i.e., exactly like the argument sequence of `\LevelSorted`. It's not fed to that macro right now, but it is not unlikely that it will in the future. The default definition corresponds to the default definition in doc.

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

Unlike the case in doc, the formatting of the text in `\saved@macroname` must be included.

`\if@version@key@` The `@version@key@` switch is used for supporting intelligent sorting of version numbers. It is normally false, but at times where the version number argument of `\changes` is being expanded because it will be used as a sort key then it is true. This is used by the `\uintver` macro. Assignments to this switch are as a rule global, since it is never true for any longer time.

```
\newif\if@version@key@
\@version@key@false
```

`\uintver` The `\uintver` command can be used in the ⟨*version*⟩ argument of `\changes` to ensure that (unsigned) integers are sorted in mathematical rather than ASCII order by makeindex. Thus if for example version `1.10` is later than version `1.9` then one should write this as

`\changes{1.\uintver{10}}{`...

The general syntax is

> `\uintver{`⟨*number*⟩`}`

and this expands completely in TeX's mouth.

The idea is that 0–9 are compared as 0–9, whereas 10–99 are compared as A10–A99, 100–999 are compared as B100-B999, and so on. The comparisons are correct up to 99999, but it could easily be extended further.

```
\newcommand*\uintver[1]{%
   \if@version@key@
      \ifnum #1>9
         \ifnum #1<100
            A%
         \else\ifnum #1<\@m
            B%
         \else\ifnum #1<\@M
            C%
         \else
            D%
         \fi\fi\fi
      \fi
   \fi
   \expandafter\@firstofone \expandafter{\number#1}%
}
```

`\changes@` This `\changes@` is a simple redefinition of the doc macro with the same name. The main difference is that all formatting of the second entry level has been taken out—it is supposed to be provided in `\saved@macroname`—but in addition to that the date is being used as a third level sort key and `\uintver` may be used in the version number to correct the data.

The former makes more sense for projects where the date is increased faster than the version number and it doesn't change anything relevant in the remaining cases. The latter is necessary if version numbers are assigned for example by CVS.

```
\def\changes@#1#2#3{%
   \global\@version@key@true
   \protected@edef\@tempa{#1}%
   \global\@version@key@false
   \protected@edef\@tempa{%
      \noexpand\glossary{%
         \@tempa\actualchar#1\levelchar
         \expandafter\@firstoftwo\saved@macroname\actualchar
            \expandafter\@secondoftwo\saved@macroname:\levelchar
         #2\actualchar#3%
      }%
   }%
   \@tempa
   \endgroup
   \@esphack
}
```

`\@wrglossary`
`\XD@glossary@keyword`
The `\@wrglossary` macro is the one which actually writes entries to the `.glo` file. It is redefined by xdoc to put the contents of `\XD@glossary@keyword`, rather than a

hardwired `\glossaryentry`, in front of the glossary entry. `\XD@glossary@keyword` is redefined by the docindex package [2].

```
\def\@wrglossary#1{%
   \protected@write\@glossaryfile{}%
      {\XD@glossary@keyword{#1}{\thepage}}%
   \endgroup
   \@esphack
}
\@ifundefined{XD@glossary@keyword}{%
   \edef\XD@glossary@keyword{\@backslashchar glossaryentry}%
}{}
```

`\definechange`
`\XD@definechange`

The `\definechange` command has the syntax

> `\definechange{`⟨*name*⟩`}{`⟨*version*⟩`}{`⟨*date*⟩`}{`⟨*text*⟩`}`

The three last arguments are precisely like the arguments of `\changes`, but `\definechange` doesn't write the change to the `.glo` file; instead it stores them away as the "named change" ⟨*name*⟩, for later use in the `\usechange` command.

```
\newcommand\definechange{%
   \begingroup\@sanitize
   \catcode`\\\z@ \catcode`\ 10 \MakePercentIgnore
   \expandafter\endgroup \XD@definechange
}
\def\XD@definechange#1#2#3#4{\@namedef{XD@ch-#1}{{#2}{#3}{#4}}}
```

`\XD@ch-`⟨*name*⟩

The named changes are stored in the `\XD@ch-`⟨*name*⟩ family of control sequences. These are parameterless macros with replacement texts of the form

> `{`⟨*version*⟩`}{`⟨*date*⟩`}{`⟨*text*⟩`}`

`\usechange`
`\XD@usechange`

To use a named change defined earlier, one of course uses the command `\usechange`, which has the syntax

> `\usechange{`⟨*name*⟩`}`

The effect of this is similar to that of a general `\changes` (i.e., it appears outside all `macro`-like environments) with the arguments specified in the `\definechange`, but this also includes the macro (or whatever) name with the page number, using the encapsulation mechanism in makeindex.

```
\newcommand*\usechange[1]{%
   \@ifundefined{XD@ch-#1}{%
      \PackageError{xdoc2}{Named change `#1' undefined}\@eha
   }{%
      \expandafter\expandafter \expandafter\XD@usechange
         \csname XD@ch-#1\endcsname
   }%
}
\def\XD@usechange#1#2#3{%
   \def\@tempa{{ }{\generalname}}%
   \ifx \@tempa\saved@macroname
      \let\@tempa\@empty
   \else
      \protected@edef\@tempa{%
```

```
         \encapchar labelednumber%
         {\expandafter\@secondoftwo\saved@macroname}%
      }
   \fi
   \global\@version@key@true
   \protected@edef\@tempb{#1}%
   \global\@version@key@false
   \glossary{%
      \@tempb\actualchar #1\levelchar
      \space\actualchar\generalname:\levelchar
      #2\actualchar#3\@tempa
   }%
}
```

`\labelednumber`

The `\labelednumber` macro belongs to the same category as the `\main` and `\usage` macros, but it takes an extra argument. The syntax is

`\labelednumber{`⟨*extra*⟩`}{`⟨*number*⟩`}`

which typesets as

⟨*number*⟩ (⟨*extra*⟩)

```
\newcommand*\labelednumber[2]{#2\nolinebreak[2] (#1)}
```

8 `macro`-like environments

There are several reasons one might want to improve the `macro` and `environment` environments.

- The code in them cannot be reused if you want to define other things than TeX macros or LaTeX environments. (During the last year or so, I have defined `macro`-like environments for over a dozen different things.)
- They always put the macro/environment name to the left of the current column. This is inappropriate for two-sided printing, as there should be a symmetry over an entire spread in that case.
- The vertical extent of a macro/environment name must not exceed that of the `\strut`, since they will otherwise overprint each other when stacked. In particular this makes it impossible to make line breaks in macro names—something which would otherwise be of interest in projects (such as for example [3]) where some names are very long and obvious breakpoints are available.

(I'm quite sure there are more things that have annoyed me, but I can't remember which they are right now.) The redefinitions below take care of the all these problems.

8.1 Grabbing arguments

A special feature of the `macro`-like environments is that (at least some) of their arguments must be given rather special treatment. This special treatment usually consists of making temporary `\catcode` changes for the time these arguments are

Grabber	**Arg. type**	**Catcodes**[a]	**Post-processing**
`\XD@grab@marg`	Mandatory	—	None
`\XD@grab@oarg`	Optional	—	None
`\XD@grab@sarg{`⟨*char*⟩`}`	1-char optional	—	Returns `\BooleanTrue` if the character was present and `\BooleanFalse` otherwise.
`\XD@grab@withprivate`	Mandatory	PL	None
`\XD@grab@asmacro`[b]	Mandatory	OB+PL	None
`\XD@grab@harmless`⟨*proc*⟩	Mandatory	—	`\MakeHarmless` followed by ⟨*proc*⟩
`\XD@grab@harmless@oarg`	Optional	—	`\MakeHarmless`
`\XD@grab@harmless@asmacro`	Mandatory	OB+PL	`\MakeHarmless` followed by `\XD@unbackslash`
`\XD@grab@harmless@cs`	Mandatory[c]	PL	`\string` whilst `\escapechar` is set to −1, followed by `\MakeHarmless`
`\XD@grab@harmless@withprivate{`⟨*proc*⟩`}`	Mandatory	PL	`\MakeHarmless` followed by ⟨*proc*⟩

---

[a] Catcode settings key: — = no change, PL = changes made by `\MakePrivateLetters`, OB = set the catcode of backslash to ordinary.

[b] This grabber is probably obsolete; it is included because it grabs the argument in precisely the way that the `macro` environment of `doc` does.

[c] The argument is normally precisely one control sequence.

Table 2: Grabbers currently defined by `xdoc`

tokenized—since the standard `\catcode`s for some important characters tend to be unsatisfactory in these cases—but there are other possibilities as well. For that reason, the `xdoc` package employs a mechanism that is very similar to that used in the Mittelbach–Rowley–Carlisle `xparse` package [6], although it does not share any code with that. I call this mechanism the argument grabber.

The heart of the argument grabber is the macro `\XD@grab@arguments`, which has the following syntax:

`\XD@grab@arguments{`⟨*call*⟩`}{`⟨*grabber sequence*⟩`}`⟨*arguments to grab*⟩

⟨*call*⟩ is something which will eventually be placed in front of all the arguments grabbed. It can simply be a single macro, but it can also contain some arguments for that macro. ⟨*grabber sequence*⟩ is a sequence of grabbers. A *grabber* is typically a macro which grabs the next argument and stores it in a token list together with the arguments that were grabbed before. A grabber could however be some more complex piece of code that performs a similar action.

When arguments are being grabbed, the ⟨*call*⟩ is stored in `\toks@` and the arguments are appended to `\toks@` as they are grabbed. For that reason, a grabber may not itself call `\XD@grab@arguments`, nor may it use a command defined through `xparse`'s `\DeclareDocumentCommand` or anything else which uses this token register in a bad way.

When a grabber is expanded, it is in the context

⟨*grabber*⟩ ⟨*following grabbers*⟩ `\XD@endgrab` ⟨*ungrabbed arguments*⟩

After it has grabbed its argument, everything of the above should be put back except for the ⟨*grabber*⟩ and the argument it grabbed. The argument itself should be wrapped in a group and appended to `\toks@`.

Note: In prototype 2 the format in which the argument grabber returns the grabbed arguments was changed so that it can now be unified with argument grabbing mechanisms of xparse. I think this should be done some time in the future, but for the moment it seems best not to rely on $\LaTeX\,2_{\varepsilon}$* packages like xparse.

`\XD@grab@arguments`
`\XD@endgrab`

The `\XD@grab@arguments` and `\XD@endgrab` macros set up and finish off argument grabbing.

```
\def\XD@grab@arguments#1#2{%
   \toks@={#1}%
   #2\XD@endgrab
}
```

```
\def\XD@endgrab{\the\toks@}
```

`\XD@grab@marg`

A grabber for ordinary arguments, like the `m` arguments of xparse.

```
\long\def\XD@grab@marg#1\XD@endgrab#2{%
   \addto@hook\toks@{{#2}}%
   #1\XD@endgrab
}
```

`\XD@grab@oarg`
`\XD@grab@oarg@`

A grabber for optional arguments (`o` arguments in xparse). It looks ahead for an optional argument and grabs that argument if there was one. If it doesn't find anything which looks like an optional argument (i.e., if the next character isn't a `[`), then the grabber will not grab anything (although it may have tokenized the next argument), but it will still append `\NoValue` to `\toks@`.

```
\def\XD@grab@oarg#1\XD@endgrab{%
   \@ifnextchar[{\XD@grab@oarg@{#1}}{%
      \addto@hook\toks@\NoValue
      #1\XD@endgrab
   }%
}
```

`\XD@grab@oarg@` is a helper to remove the brackets around the optional argument.

```
\long\def\XD@grab@oarg@#1[#2]{%
   \addto@hook\toks@{{#2}}%
   #1\XD@endgrab
}
```

`\XD@grab@sarg`

A grabber for 'star'-type arguments (`s` arguments in xparse). The syntax is

`\XD@grab@sarg{`⟨*char*⟩`}`

It looks ahead to see if the next character is the ⟨*char*⟩. In that case it gobbles it and adds a `\BooleanTrue` to the grabbed arguments, otherwise it adds a `\BooleanFalse` to the grabbed arguments.

```
\def\XD@grab@sarg#1#2\XD@endgrab{%
   \@ifnextchar#1{%
      \addto@hook\toks@\BooleanTrue
      \@firstoftwo{#2\XD@endgrab}%
   }{%
      \addto@hook\toks@\BooleanFalse
      #2\XD@endgrab
   }%
}
```

`\XD@grab@withprivate`

`\XD@grab@withprivate` is like `\XD@grab@marg` but grabs the argument when the catcodes are as set by `\MakePrivateLetters`.

```
\def\XD@grab@withprivate{%
   \begingroup\MakePrivateLetters\relax\expandafter\endgroup
   \XD@grab@marg
}
```

To think about: Perhaps things like `\XD@grab@withprivate` should rather be considered a modifier for a grabber? Instead of having `\XD@grab@withprivate` be the entire grabber, one could let the grabber be something like

```
\XD@grab@withprivate\XD@grab@marg
```

where the `\XD@grab@withprivate` should only expand to

```
\begingroup\MakePrivateLetters\relax\expandafter\endgroup
```

`\XD@grab@asmacro`

`\XD@grab@asmacro` is very similar to `\XD@grab@withprivate`, but it sees to that the catcode settings are exactly those used by `doc`'s `macro` environment.

```
\def\XD@grab@asmacro{%
   \begingroup
      \catcode`\\=12 \MakePrivateLetters\relax
   \expandafter\endgroup
   \XD@grab@marg
}
```

`\XD@grab@harmless`
`\XD@grab@harmless@oarg`
`\XD@grab@harmless@oarg@`

The `\XD@grab@harmless` grabber grabs one mandatory argument and converts it to a harmless character string, which it contributes to the list of arguments. The syntax is

`\XD@grab@harmless{`⟨*post-processing*⟩`}`

where ⟨*post-processing*⟩ are commands that will be performed after the grabbed argument has been made harmless, but before it is contributed to the list of arguments. Thus the ⟨*post-processing*⟩ can modify the argument some more, but ⟨*post-processing*⟩ can just as well be empty.

```
\def\XD@grab@harmless#1#2\XD@endgrab#3{%
   \MakeHarmless\@tempa{#3}%
   #1%
   \toks@=\expandafter{\the\expandafter\toks@ \expandafter{\@tempa}}%
   #2\XD@endgrab
}
```

The `\XD@grab@harmless@oarg` grabber grabs one optional argument and converts it to a harmless character string. This string is contributed to the list of arguments if the optional argument, or else the token `\NoValue` is contributed instead.

```
\def\XD@grab@harmless@oarg#1\XD@endgrab{%
```

```
   \@ifnextchar[{\XD@grab@harmless@oarg@{#1}}{%
      \addto@hook\toks@\NoValue
      #1\XD@endgrab
   }%
}
```

`\XD@grab@harmless@oarg@` is a helper to remove the brackets around the optional argument.

```
\long\def\XD@grab@harmless@oarg@#1[#2]{%
   \MakeHarmless\@tempa{#2}%
   \toks@=\expandafter{\the\expandafter\toks@ \expandafter{\@tempa}}%
   #1\XD@endgrab
}
```

`\XD@grab@harmless@asmacro`
`\XD@grab@harmless@cs`
`\XD@grab@harmless@cs@`

The `\XD@grab@harmless@asmacro` grabber combines the features of `\XD@grab@asmacro` and `\XD@grab@harmless`, since when the argument to grab is tokenized the catcode of \ is set to 12 and the catcode assignments in `\MakePrivateLetters` are made. Then the grabbed argument is converted to a harmless character sequence, and finally the first character is removed if it is a backslash.

```
\def\XD@grab@harmless@asmacro{%
   \begingroup
      \catcode`\\=12 \MakePrivateLetters\relax
   \expandafter\endgroup
   \XD@grab@harmless{%
      \protected@edef\@tempa{%
         \expandafter\XD@unbackslash\@tempa\@empty
      }%
   }%
}
```

The `\XD@grab@harmless@cs` grabber is for use with commands like `doc`'s `\DescribeMacro`, which take an actual control sequence as the argument. It grabs one argument while having catcodes changed as indicated by `\MakePrivateLetters`, `\string`s the argument while `\escapechar` is `-1` (so that there is no escape character inserted), and continues as `\XD@grab@harmless`.

```
\def\XD@grab@harmless@cs{%
   \begingroup
      \MakePrivateLetters\relax
   \expandafter\endgroup \XD@grab@harmless@cs@
}
```

```
\long\def\XD@grab@harmless@cs@#1\XD@endgrab#2{%
   \begingroup
      \escapechar=\m@ne
   \expandafter\endgroup
   \expandafter\MakeHarmless \expandafter\@tempa
      \expandafter{\string#2}%
   \toks@=\expandafter{\the\expandafter\toks@ \expandafter{\@tempa}}%
   #1\XD@endgrab
}
```

`XD@grab@harmless@withprivate`

`\XD@grab@harmless@withprivate` is like `\XD@grab@harmless` but grabs the argument when the catcodes are as set by `\MakePrivateLetters`. Like `\XD@grab@harmless`, `\XD@grab@harmless@withprivate` takes an argument which can contain code that modifies the harmless character string after it has been formed.

```
\def\XD@grab@harmless@withprivate{%
  \begingroup\MakePrivateLetters\relax\expandafter\endgroup
  \XD@grab@harmless
}
```

8.2 The `\XD@m@cro` and `\NewMacroEnvironment` commands

In doc the macro that contains most of the code for the `macro` and `environment` environments is called `\m@cro@`. In xdoc the corresponding macro is `\XD@m@cro`.

At this point, it is helpful to recall what `\m@cro@` actually does. It can be summarized in the following four points:

- It starts a `\trivlist`.[10]
- It prints the name of the macro/environment that is about to be defined in the margin.
- It writes an index entry (and inhibits cross-referencing of the macro inside the environment).
- It sets `\saved@macroname` to the name of the macro/environment (for use by `\changes`).

The first and fourth points are simple, and commands for the third were defined in Section 4, but the second point needs a few helper macros.

`\XDStackItemLabels`
`\XD@macro@dimen`

The `\XDStackItemLabels` macro is a definition of `\makelabel` which is used in the `macro`-like environments for stacking the names printed by subsequent environments under each other. It makes a box which has zero height and depth (it should have zero width as well, but that is left as a restriction on the argument) and the printed names will be stacked if the reference points of the subsequent boxes generated by `\XDStackItemLabels` coincide.

`\XD@macro@dimen` (always assigned globally) stores the vertical distance from the reference point of the box that `\XDStackItemLabels` makes to the (bottommost) baseline of the previous printed name. `\XD@macro@dimen` is updated by each new `\XDStackItemLabels`. The baseline of the next printed name will be put one `\baselineskip` lower than that of the previous printed name, except for when `\XD@macro@dimen` is `-\maxdimen` (see below). To avoid that printed names clash into each other, this additional `\baselineskip` is generated as normal interline glue where the upper box has the same depth as a strut and the new value of `\XD@macro@dimen` is measured in such a way that the printed name's depth below the nominal baseline will not exceed the depth of a strut (that's what the `\boxmaxdepth` assignment is for). When `\XD@macro@dimen` is `-\maxdimen` the (topmost) baseline of the printed name will instead go through the reference point of the box. This case is intended for the first item label in a stack.

The reason `\everypar` is cleared is that that is where the list environments put the commands which actually insert the item label into the paragraph. If that

---

[10] Seriously, can someone explain to me why it seems just about every non-math LaTeX environment that doesn't start a `\list` starts a `\trivlist`? What good does all these `\trivlists` do? Is it (a) that people just like the basic design, (b) that there's some deep technical reason, or (c) that people in general doesn't have a clue but all other environments do that so it's best to include it just in case?

code gets executed inside `\makelabel`, the list environments get seriously confused with not at all nice consequences.

```
\def\XDStackItemLabels#1{%
   \setbox\z@=\vbox{%
      \ifdim \XD@macro@dimen=-\maxdimen
         \setbox\z@=\vtop{%
            \color@begingroup
            \everypar={}%
            #1%
            \color@endgroup
         }%
         \kern-\ht\z@
         \unvbox\z@
      \else
         \color@begingroup
         \everypar={}%
         \kern\XD@macro@dimen
         \setbox\z@=\copy\strutbox \ht\z@=\z@ \box\z@
         #1%
         \color@endgroup
      \fi
      \boxmaxdepth=\dp\strutbox
   }%
   \global\XD@macro@dimen=\ht\z@
   \vtop to\z@{\unvbox\z@ \vss}%
}
```

```
\newdimen\XD@macro@dimen
```

`\XDToMargin`

The `\XDToMargin` macro takes one argument, which is assumed to be some horizontal material, and puts that material in a `\hbox` of width zero, horizontally shifted out into the the outer margin, in such a way that longer arguments extend further out. `\marginparsep` is used as the distance between the argument and the main galley. All these placements assume that the `\hbox` will be put `\labelsep` to the left of the beginning of a nonindented paragraph, since that is where it will be put by the `\item` of a `\trivlist`.

A question is where the margin should be considered to start if the `\@totalleftmargin` isn't zero. The corresponding `doc` action would be to consider the margin as everything outside the `\linewidth` width, but I don't think that would be appropriate here (especially not since `doc` always puts the codeline numbers at the edge of the `\textwidth` width).

```
\newcommand\XDToMargin[1]{%
   \hb@xt@\z@{%
      \IfOddPageSituation{%
         \dimen@=-\@totalleftmargin
         \advance \dimen@ \labelsep
         \advance \dimen@ \textwidth
         \advance \dimen@ \marginparsep
         \kern\dimen@
      }\hss
      #1%
      \IfOddPageSituation\hss{%
         \dimen@=\@totalleftmargin
         \advance \dimen@ -\labelsep
         \advance \dimen@ \marginparsep
         \kern\dimen@
      }%
   }%
}
```

`\XDParToMargin`

The `\XDParToMargin` command is in syntax and use similar to the `\XDToMargin` command, but it will try to linebreak an argument that is too long rather than letting it extend outside the paper.

The implementation first tries to break the argument without considering justification or positioning, but with a rather high `\linepenalty`. If the result of that try is a single line paragraph then `\XDToMargin` will be called to actually typeset the argument. Otherwise the argument is typeset as a paragraph which gets displaced out into the outer margin by giving `\leftskip` and `\rightskip` nonzero natural widths. The practical line width in the paragraph is the `\marginparwidth`, but the hboxes containing the individual lines will have width zero. The first line of the paragraph will be set flush outwards, the last line of the paragraph will be set flush inwards, and the remaining lines will be centered.

```
\newcommand\XDParToMargin[1]{%
   \parindent=\z@
   \setbox\z@=\vbox{%
      \leftskip=\z@skip
      \rightskip=\z@\@plus 1fil%
      \parfillskip=\z@skip
      \hsize=\marginparwidth
      \linepenalty=1000%
      \color@begingroup
      \noindent\ignorespaces #1\@@par
      \color@endgroup
   \expandafter}%
   \expandafter\ifnum \the\prevgraf<\tw@
      \XDToMargin{#1}%
   \else
      \hsize=\z@
      \leftskip=\z@ \@plus \marginparwidth
      \rightskip=\leftskip
      \IfOddPageSituation{%
         \dimen@=-\@totalleftmargin
         \advance \dimen@ \labelsep
         \advance \dimen@ \textwidth
         \advance \dimen@ \marginparsep
         \advance \leftskip \dimen@
         \advance \rightskip -\dimen@ \@minus \p@
         \advance \rightskip -\marginparwidth
         \parfillskip=\z@ \@plus 1fil%
      }{%
         \dimen@=\@totalleftmargin
         \advance \dimen@ -\labelsep
         \advance \dimen@ \marginparsep
         \advance \leftskip -\dimen@ \@minus \p@
         \advance \leftskip -\marginparwidth
         \advance \rightskip \dimen@
         \parfillskip=\z@ \@plus -\marginparwidth%
      }
      \noindent\nobreak\hskip\parfillskip
      \ignorespaces #1\@@par
    \fi
}
```

In the following I exploit the implementation of the `\item` command in a slightly hackish way. Instead of starting a new paragraph with the item label (which is what one at first would believe `\item` does), `\item` actually puts the label in the box `\@labels` register, and stores code in `\everypar` that inserts that box into the new paragraph. Therefore I can make sure that various `\write` whatsits that need to be as the same page as an `\item` label will be there by adding them to the contents of the `\@labels` box. This seems more reliable to me than putting them on the vertical list followed by a `\nobreak` as `doc` does, but that would probably work as well.

[A funny thing in that which confused me a while was the question of whether the `\box` command that inserts the box into the paragraph and simultaneously clears the register acted globally or locally. It turns out that the question was ill-posed, as the distinction between local and global assignments is determined by what restore items they put on TEX's save stack. The `\box` command doesn't put anything there, so the assignment it makes will essentially appear at the same grouping level as the `\setbox` command that set the contents of the box register. As all `\setbox`es for the `\@labels` box register are global, the box register will be globally void after `\box\@labels`.]

`\XD@m@cro` This is the workhorse of all the `macro`-like environments. It calls `\trivlist` and sets related parameters, prints the “macro” name in the proper place, updates the representation of the “macro” name that `\changes` will use, and writes appropriate index entries (possibly making temporary changes in cross-referencing). Exactly what these tasks consist of can vary quite a lot between different `macro`-like environments, and therefore the `\XD@m@cro` macro has the following syntax:

> `\XD@m@cro{`⟨*print*⟩`}{`⟨*index*⟩`}{`⟨*changes*⟩`}{`⟨*assign*⟩`}`

⟨*print*⟩, ⟨*index*⟩, and ⟨*assign*⟩ are simply the commands for printing the “macro” name as it should appear in the margin, generating the index entries for this `macro`-like environment, and making whatever additional local assignments that are needed for this environment (usually a couple of `\DoNotIndexHarmless` commands, if anything at all) respectively. At the time ⟨*index*⟩ is executed, `codelineno` holds the number of the *next* codeline. ⟨*changes*⟩, finally, is code that will be put in the context

> `\protected@edef\saved@macroname{`⟨*changes*⟩`}`

to set the `\saved@macroname` macro (for `\changes`).

```
\def\XD@m@cro#1#2#3#4{%
  \topsep\MacroTopsep
  \trivlist
  \global\setbox\@labels=\hbox{%
    \unhbox\@labels
    \if@inlabel \else
      \global\XD@macro@dimen=-\maxdimen
      \StepPageSituation
      \RecordPageSituation
    \fi
    \advance \c@codelineno \@ne
    #2%
  }%
  \let\makelabel\XDStackItemLabels
  \item[#1]%
  \protected@edef\saved@macroname{#3}%
  #4%
  \ignorespaces
}
```

In the first `xdoc` prototype, the `macro`-like environments were implemented so that each new environment only used two control sequences (`\`⟨*env*⟩ and `\end`⟨*env*⟩), which is the absolute minimum. This implementation worked fine for single argument environments, but the number of helper macros that would have to be introduced to deal with multiple argument environments exceeded what could be considered reasonable. Therefore the second prototype claims a third control sequence for the implementation of a `macro`-like environment ⟨*env*⟩, namely `\\`⟨*env*⟩, which is also used by normal LaTeX 2$_\varepsilon$ environments which take an optional argument.

It should also be mentioned that the implementation in the first prototype required that most of the code in `\`⟨*env*⟩ had to be written in a very special way. Instead of using the `#`⟨*digit*⟩ notation for the arguments and write straightforward LaTeX code, one had to express everything using macros which operate on arguments "up ahead" (immediately after the code you can specify). This curious coding model made it out of the question to create a class designer interface for defining new `macro`-like environments, but in the second `xdoc` prototype it is quite simple to do something of that sort: the command name is `\NewMacroEnvironment`.

`\NewMacroEnvironment`
`\XD@NewMacroEnvironment`
`\XD@NewMacroEnvironment@`

The `\NewMacroEnvironment` command is used for defining new `macro`-like environments. It has the syntaxes

```
\NewMacroEnvironment{⟨name⟩}{⟨grabbers⟩}{⟨numargs⟩}
                 {⟨unjust-print⟩}{⟨index⟩}{⟨changes⟩}{⟨assign⟩}
\NewMacroEnvironment*{⟨name⟩}{⟨grabbers⟩}{⟨numargs⟩}
                       {⟨print⟩}{⟨index⟩}{⟨changes⟩}{⟨assign⟩}
```

where ⟨*name*⟩ is the name of the environment to define, ⟨*grabbers*⟩ is a sequence of argument grabbers, ⟨*numargs*⟩ is the number of arguments that the grabbers will grab, and ⟨*print*⟩, ⟨*index*⟩, ⟨*changes*⟩, and ⟨*assign*⟩ are code that will be put in the respective arguments of `\XD@m@cro`. In the four last arguments, argument specifiers `#1` to `#`⟨*numargs*⟩ inclusive can be used do mean the arguments that were grabbed by the sequence of grabbers.

The argument grabbers that are currently made available by the `xdoc` package are listed in Table 2 on page 43.

The ⟨*print*⟩ code will be executed while TeX is in internal vertical mode and it should put one or several hboxes of width zero onto the vertical list. The contents of these boxes should be some amount of text which will appear displaced out into the outer margin on the page when the reference point of the box appears

`\labelsep` to the left of the left edge of the line. The easiest way of achieveing this is to use a ⟨*print*⟩ of the form

`\XDToMargin{`⟨*unjust-print*⟩`}`

and this is exactly what the non-star form of `\NewMacroEnvironment` does by default.

```
\newcommand\NewMacroEnvironment{%
   \@ifstar\XD@NewMacroEnvironment\XD@NewMacroEnvironment@
}
\def\XD@NewMacroEnvironment@#1#2#3#4{%
   \XD@NewMacroEnvironment{#1}{#2}{#3}{\XDToMargin{#4}}%
}
\def\XD@NewMacroEnvironment#1#2#3#4#5#6#7{%
   \expandafter\@ifdefinable\csname#1\endcsname{%
      \expandafter\def \csname#1\expandafter\endcsname
         \expandafter{\expandafter\XD@grab@arguments
         \csname\@backslashchar#1\endcsname{#2}}%
      \let\l@ngrel@x\relax
      \expandafter\@yargdef \csname\@backslashchar#1\endcsname \@ne
         {#3}{\XD@m@cro{#4}{#5}{#6}{#7}}%
      \expandafter\let \csname end#1\endcsname \endtrivlist
   }%
}
```

The ⟨*grabbers*⟩ argument—in which one specifies a list of internal macros—is not how the interface should really look, but I think it will have to do for now. The final interface will probably use something like the argument specifications of `\DeclareDocumentCommand`, but there is little point in implementing that before xparse has gotten its final form.

The macro `\@yargdef` used above should perhaps be checked so that its syntax hasn't changed, but since `\@yargdef` quite recently (`ltdefn.dtx` v 1.3c, 1999/01/18) was completely reimplemented without any change in the syntax (despite the fact that the syntax is afterwards rather peculiar), I think it can be assumed that the syntax will not change in LaTeX 2_ε.

8.3 Reimplementing `macro` and `environment`

Well, then how does one reimplement the `macro` and `environment` environments using `\XD@m@cro`? We shall soon see, but first it is convenient to define a utility macro.

`\XDMainIndex` The `\XDMainIndex` macro is an abbreviation to save a couple of tokens in a very frequent call to `\IndexEntry`. It has the syntax

`\XDMainIndex{`⟨*argument*⟩`}`

and that expands to

`\IndexEntry{`⟨*argument*⟩`}{main}{\TheXDIndexNumber}`

```
\newcommand\XDMainIndex[1]{\IndexEntry{#1}{main}{\TheXDIndexNumber}}
```

`macro`
`environment`

It is very easy to implement `macro` and `environment` environments which behave pretty much as in doc using the `\NewMacroEnvironment` command. The important difference is that in doc everything that distinguished the two environments was to be found in various helper macros, but here all that code is in the `\\macro` and `\\environment` macros. Thus to define one new `macro`-like environment, one doesn't have to define six or so new macros—everything can be handled in one definition.

The reason for the `\let` commands below is of course that `macro` and `environment` are already defined, and there is no `\RenewMacroEnvironment` command. It could perhaps have been better if `\NewMacroEnvironment` had behaved like `\DeclareRobustCommand`, but I don't think that is an important problem for the moment.

```
\let\macro=\relax
\let\endmacro=\relax
\NewMacroEnvironment{macro}{\XD@grab@harmless@asmacro}{1}
   {\MacroFont\Bslash#1}
   {\MakeSortKey\@tempa{#1}{}%
    \XDMainIndex{\LevelSorted{\@tempa}{\texttt{\Bslash#1}}}}
   {{#1}{\texttt{\Bslash#1}}}
   {\DoNotIndexHarmless{#1}}
```

```
\let\environment=\relax
\let\endenvironment=\relax
\NewMacroEnvironment{environment}{\XD@grab@harmless@asmacro}{1}
   {\MacroFont#1}
   {\XDMainIndex{\LevelSorted{#1}{\texttt{#1} (environment)}}%
    \XDMainIndex{%
       \LevelSame{environments:}\LevelSorted{#1}{\texttt{#1}}%
   }}%
   {{#1}{\texttt{#1}}}
   {}%
```

8.4 Further examples of `macro`-like environments

`option`

The `option` environment is for class/package options. IMHO, something like this environment should have been added to doc years ago!

```
\NewMacroEnvironment{option}{\XD@grab@harmless\relax}{1}
   {\MacroFont#1 \normalfont option}
   {\XDMainIndex{\LevelSorted{#1}{\texttt{#1} option}}%
    \XDMainIndex{%
       \LevelSame{options:}\LevelSorted{#1}{\texttt{#1}}%
   }}%
   {{#1 option}{\texttt{#1} option}}
   {}%
```

`switch`

The `switch` environment is for switches created by `\newif` (PLAIN TEX style).

```
\NewMacroEnvironment{switch}{\XD@grab@harmless\relax}{1}
   {\MacroFont#1 \normalfont switch}%
```

What makes switches different from the other `macro`-like environments defined here is the large number of index entries it makes. For a switch ⟨*sw*⟩ it first makes one under the 'switches:' heading:

```
   {%
```

```
      \MakeSortKey\XD@last@key{#1}{}%
      \XDMainIndex{%
         \LevelSame{switches:}\LevelSorted{\XD@last@key}{\texttt{#1}}%
      }%
```

Second it makes a '⟨*sw*⟩ switch' entry:

```
      \XDMainIndex{\LevelSorted{\XD@last@key}{\texttt{#1} switch}}%
```

Third it makes an entry for the macro `\if`⟨*sw*⟩. The sort key for this entry is *not* subjected to `\MakeSortKey` because no reasonable operator will act on the `if` prefix (an operator which acts on `if` could do rather strange things to e.g. `\ifnum`).

```
      \XDMainIndex{\LevelSorted{if#1}{\texttt{\Bslash if#1}}}%
```

Fourth it makes an entry for the macro `\`⟨*sw*⟩`false`:

```
      \MakeSortKey\@tempa{#1false}{}%
      \XDMainIndex{\LevelSorted{\@tempa}{\texttt{\Bslash#1false}}}%
```

Finally it makes an entry for the macro `\`⟨*sw*⟩`true`:

```
      \MakeSortKey\@tempa{#1true}{}%
      \XDMainIndex{\LevelSorted{\@tempa}{\texttt{\Bslash#1true}}}%
   }%
```

The `\changes` heading, on the other hand, is trivial.

```
   {{#1}{\texttt{#1} switch}}
```

Finally, `switch` should turn off indexing of the three macros it makes `main` entries for, since makeindex will otherwise complain.

```
   {\DoNotIndexHarmless{if#1}%
    \DoNotIndexHarmless{#1false}%
    \DoNotIndexHarmless{#1true}}%
⟨/pkg⟩
```

To end this section, there now follows two examples which are not part of the package as they are very specific, but which have been included here because they illustrate that `macro`-like environments may have several arguments.

`enccommand`
`enccomposite`

The `enccommand` and `enccomposite` environments can be used for marking up sources for encoding definition files and the like. `enccommand` is for encoding-specific commands and has the syntax

> `\begin{enccommand}{`⟨*command*⟩`}[`⟨*encoding*⟩`]`

where ⟨*command*⟩ is the encoding-specific command and ⟨*encoding*⟩ is the encoding that this definition is for. If the ⟨*encoding*⟩ is omitted then the `enccommand` is assumed to be for the default definition of the command.

`enccomposite` is for composites of encoding-specific commands (defined for example using `\DeclareTextComposite`). It has the syntax

> `\begin{enccomposite}{`⟨*command*⟩`}{`⟨*encoding*⟩`}{`⟨*argument*⟩`}`

where ⟨*command*⟩ and ⟨*encoding*⟩ are as for `enccommand` and ⟨*argument*⟩ is the argument with which the command is being composed.

The marginal headings these commands print are the actual control sequences in which the definitions are stored.

```
⟨*enccmds⟩
```

```
\NewMacroEnvironment{enccommand}{%
   \XD@grab@harmless@asmacro \XD@grab@oarg
}{2}{\MacroFont\Bslash \ifx\NoValue#2?\else#2\fi \Bslash #1}{%
   \XDMainIndex{%
      \LevelSorted{#1}{\texttt{\Bslash#1}}%
      \ifx \NoValue#2%
         \LevelSame{default}%
      \else
         \LevelSorted{#2}{\texttt{#2} encoding}%
      \fi
   }%
}{{#1}{\texttt{\Bslash#1}}}{\DoNotIndexHarmless{#1}}
```

```
\NewMacroEnvironment{enccomposite}{%
   \XD@grab@harmless@asmacro \XD@grab@marg \XD@grab@harmless\relax
}{3}{\MacroFont\Bslash#2\Bslash#1-#3}{%
   \XDMainIndex{%
      \LevelSorted{#1}{\texttt{\Bslash#1}}%
      \LevelSorted{#2}{\texttt{#2} encoding}%
      \LevelSorted{\XD@unbackslash#3\@empty}{\texttt{#3} composite}%
   }%
}{{#1}{\texttt{\Bslash#1}}}{\DoNotIndexHarmless{#1}}
⟨/enccmds⟩
```

In the file `cyoutenc.dtx` the definitions of many encoding-specific commands are written so that the same line of code can work is all four files `t2aenc.def`, `t2benc.def`, `t2cenc.def`, and `x2enc.def`. Therefore the ⟨*encoding*⟩ argument of the `enccommand` and `enccomposite` environments should perhaps rather be a comma-separated list of encodings than a single encoding, but that would make this example unnecessarily complicated.

9 Describing macros and the like

`\if@mparswitch`
`\if@reversemargin`

In two-sided mode, marginal notes should appear in the outer margin. The following code takes care of that.

```
⟨*pkg⟩
\if@twoside
   \@mparswitchtrue
   \normalmarginpar
\fi
```

`\GenericDescribePrint`

The `\GenericDescribePrint` macro is a utility macro for use in commands like `\DescribeMacro`. Its syntax is

> `\GenericDescribePrint{`⟨*text*⟩`}`

and it puts ⟨*text*⟩ in a marginal paragraph, giving it the appropriate justification for appearing in that margin.

The first part simply tests whether the argument fits on a single line.

```
\newcommand\GenericDescribePrint[1]{%
   \setbox\z@=\vbox{%
      \parindent=\z@
      \leftskip=\z@skip
      \rightskip=\z@\@plus 1fil%
      \parfillskip=\z@skip
      \hsize=\marginparwidth
      \linepenalty=\@m
      \color@begingroup
      \noindent\ignorespaces #1\@@par
      \color@endgroup
   \expandafter}%
   \expandafter\ifnum \the\prevgraf<\tw@
```

Then comes the actual typesetting. First the single-line format. The braces in the optional argument are there to prevent trouble in case `#1` contains a right brace; they will be stripped off when the argument is grabbed.

```
      \if@twoside
         \marginpar[{\raggedleft\strut #1}]{\raggedright\strut #1}%
      \else
         \marginpar{\raggedleft\strut#1}%
      \fi
   \else
      \if@twoside
         \marginpar[{%
            \leftskip=\z@ \@plus \marginparwidth
            \rightskip=\leftskip
            \parfillskip=\z@ \@plus -\marginparwidth
            \noindent\nobreak\hskip\parfillskip
            \ignorespaces #1%
         }]{%
            \leftskip=\z@ \@plus \marginparwidth
            \rightskip=\leftskip
            \parfillskip=\z@ \@plus 1fil%
            \noindent\nobreak\hskip\parfillskip
            \ignorespaces #1%
         }%
      \else
         \marginpar{%
            \leftskip=\z@ \@plus \marginparwidth
            \rightskip=\leftskip
            \parfillskip=\z@ \@plus -\marginparwidth
            \noindent\nobreak\hskip\parfillskip
            \ignorespaces #1%
         }%
      \fi
   \fi
}
```

The `describe`-commands are supposed to be invisible—only leave a single space even when there are spaces both before and after them—but there are problems with the mechanisms for this. I get the impression that they have never worked perfectly, but that seems to be mainly due to that certain macros in the LaTeX kernel never did either, and I suspect that the general problem has been thrashed over many times before.

`doc`'s `\DescribeMacro` and `\DescribeEnv` are wrapped up in a `\@bsphack` … `\@esphack` "group" to become invisible, but the `\marginpar` and various index commands they are built on are themselves already invisible, so one would suspect that there is no need for additional invisibility. There are however two factors which create this need. One is that it doesn't do the right thing at beginning of lines; here it seems like what the `describe`-commands would need is the `\@vbsphack` macro (whose definition appears in `ltspace.dtx`, but which has been commented out) since they should start a new paragraph and leave no following space if they are used in vertical mode. The other factor is that the standard `\@bsphack`–`\@esphack` can only suppress every second intermediate space if several invisible commands appear in sequence, as is quite common for the `describe`-commands.[11]

Instead the doc implementations of `\DescribeMacro` and `\DescribeEnv` begin with `\leavevmode` and end with `\ignorespaces`, which means that they are only "invisible" if they appear on on the left of visible material, but that's how it has been for over a decade now.

`\NewDescribeCommand` The `\NewDescribeCommand` command is a relative to the `\NewMacroEnvironment` command which defines commands analogous to `\DescribeMacro` rather than `macro`-like environments. Its syntax is

`\NewDescribeCommand{`⟨*command*⟩`}{`⟨*grabbers*⟩`}{`⟨*numargs*⟩`}{`⟨*definition*⟩`}`

⟨*command*⟩ is the control sequence to define. ⟨*grabbers*⟩ and ⟨*numargs*⟩ are as for the `\NewMacroEnvironment` command. ⟨*definition*⟩ is the command definition. In addition to the definition given in the ⟨*definition*⟩ argument and the code for grabbing the arguments, the command actually defined by `\NewDescribeCommand` will contain a `\leavevmode` at the start and an `\ignorespaces` at the end.

The `\NewDescribeCommand` command should really just be a call to xparse's `\DeclareDocumentCommand`, but that will have to wait until xdoc becomes based on the xparse package.

```
\newcommand\NewDescribeCommand[4]{%
   \@ifdefinable#1{%
      \expandafter\def \expandafter#1\expandafter{%
         \expandafter\XD@grab@arguments \csname\string#1\endcsname{#2}%
      }%
      \let\l@ngrel@x\relax
      \expandafter\@yargdef \csname\string#1\endcsname \@ne {#3}%
         {\leavevmode#4\ignorespaces}%
   }%
}
```

`\DescribeMacro` `\DescribeEnv` The `\DescribeMacro` and `\DescribeEnv` commands are as in doc. The argument of `\DescribeMacro` is supposed to be the actual control sequence to describe (not as with the `macro` environment something which looks like the control sequence after being `\stringed`).

```
\let\DescribeMacro=\relax
\NewDescribeCommand\DescribeMacro{\XD@grab@harmless@cs}{1}{%
   \GenericDescribePrint{\MacroFont\Bslash#1}%
   \MakeSortKey\@tempa{#1}{}%
   \IndexEntry{%
      \LevelSorted{\@tempa}{\texttt{\Bslash#1}}%
```

---

[11] It would seem that a simple fix for this is to have `\@esphack` insert `\nobreak \hskip-\@savsk \hskip\@savsk` before it executes `\ignorespaces`, but since that fix hasn't been incorporated into the kernel or the fixltx2e package there probably is some problem with it.

```
  }{usage}{\thepage}%
}
```

The argument of `\DescribeEnv`, on the other hand, is treated like that of the `environment` environment, but backslash isn't given catcode 12—only the catcode assignments in `\MakePrivateLetters` are made.

```
\let\DescribeEnv=\relax
\NewDescribeCommand\DescribeEnv{%
  \XD@grab@harmless@withprivate\relax
}{1}{%
  \GenericDescribePrint{\MacroFont#1}%
  \IndexEntry{%
     \LevelSame{environments:}\LevelSorted{#1}{\texttt{#1}}%
  }{usage}{\thepage}%
  \IndexEntry{%
     \LevelSorted{#1}{\texttt{#1} (environment)}%
  }{usage}{\thepage}%
}
```

`\describeoption` The `\describeoption` command is the `describe`-companion to the `option` environment.

```
\NewDescribeCommand\describeoption{\XD@grab@harmless\relax}{1}{%
  \GenericDescribePrint{\MacroFont#1 \normalfont option}%
  \IndexEntry{%
     \LevelSame{options:}\LevelSorted{#1}{\texttt{#1}}%
  }{usage}{\thepage}%
  \IndexEntry{%
     \LevelSorted{#1}{\texttt{#1} option}%
  }{usage}{\thepage}%
}
```

`\describecsfamily` The `\describecsfamily` command is for marking out sections in text where a particular family of control sequences is described—just like `\DescribeMacro` does for individual commands. To clarify what I mean by a control sequence family, here are a couple of examples:

`\c@`⟨*counter*⟩	countdef token for the `\count` register storing the LaTeX counter ⟨*counter*⟩
`\ps@`⟨*pagestyle*⟩	macro storing settings for the pagestyle ⟨*pagestyle*⟩
`\`⟨*enc*⟩`/`⟨*fam*⟩`/`⟨*ser*⟩`/`⟨*sh*⟩`/`⟨*sz*⟩	the fontdef token for the font which has encoding ⟨*enc*⟩, family ⟨*fam*⟩, series ⟨*ser*⟩, shape ⟨*sh*⟩, and size ⟨*sz*⟩ under NFSS
`\`⟨*enc*⟩`\`⟨*cmd*⟩	the macro containing the definition for encoding ⟨*enc*⟩ of the encoding-specific LaTeX command `\`⟨*cmd*⟩
`\fps@`⟨*type*⟩	the default placement specifier for LaTeX floats of type ⟨*type*⟩
`\l@`⟨*name*⟩	a macro which formats table of contents entries for items of type ⟨*name*⟩ (`chapter`, `section`, etc.)

`\l@`⟨*language*⟩	the `\language` number babel has allocated for the language ⟨*language*⟩ (`english`, `french`, etc.)
`\i-`⟨*int*⟩	the control sequence (either a mathchardef token or a macro) which stores the value of the fontinst integer ⟨*int*⟩

The syntax for `\describecsfamily` is

`\describecsfamily{`⟨*cs-fam specification*⟩`}`

The ⟨*cs-fam specification*⟩ includes only what would be put between `\csname` and `\endcsname`; the `\describecsfamily` command will add a backslash when printing the name. No special catcodes will be in force in the argument, but the `#`, `$`, `&`, `_`, `^`, and `~` characters present no problems even if they have their ordinary catcodes. All spaces are seen as ASCII space and TeX is skipping spaces as usual. Characters with catcode 0, 1, 2, 5, 9, 14, or 15 may however be problematic. If you need to specify such a problematic character then you can do so by writing `\PrintChar{`⟨*code*⟩`}`, where ⟨*code*⟩ is the ASCII code for the character, as a valid TeX number in the range 0–255. In case you do not remember the ASCII code for some character ⟨*c*⟩, there is no harm in specifying it as `` `\ ``⟨*c*⟩, e.g. `` \PrintChar{`\}} `` for a right brace. It is even possible to write `\PrintChar` commands for characters outside visible ASCII (but those are typeset as `^^`-sequences).

The variant parts in the control sequence names are specified as

`\meta{`⟨*text*⟩`}`

and these will be typeset exactly as in normal text. The arguments of `\meta`s appearing in a ⟨*cs-fam specification*⟩ are moving. All control sequences other than `\PrintChar` and `\meta` in a ⟨*cs-fam specification*⟩ (and which do not appear in the argument of a `\PrintChar` or `\meta`) are essentially treated as if they had been `\string`ed.

Apart from the above differences in treatment of the argument, the `\describecsfamily` command is similar to `\DescribeMacro`—it prints the control sequence name in the margin and makes a `usage` index entry.

```
\NewDescribeCommand\describecsfamily{\XD@grab@harmless{}}{1}{%
   \GenericDescribePrint{%
      \MetaNormalfont\MacroFont\Bslash#1%
   }%
   \MakeSortKey\@tempa{#1}{\def\meta##1{(##1)}}%
   \IndexEntry{%
      \LevelSorted{\@tempa}{\texttt{\protect\MetaNormalfont\Bslash#1}}%
   }{usage}{\thepage}%
}
⟨/pkg⟩
```

As for `\NewMacroEnvironment`, I also give an example of an application of `\NewDescribeCommand` which is much too special for including in xdoc in general and therefore the code is placed in a special module. I had originally written the code as part of another package, but I removed it because I thought it was a bit too special even for that context. The commentry below is kept unchanged.

I believe this feature is primarily of interest for MacOS programs, but there might be sufficiently similar structures in other operating systems to make it useful even in other contexts. Be as it may, what the feature described here does is that it allows the user to put an entry in the index for each resource in the code. This gives an easy way of checking that no two resources are assigned the same id, even though there is no mechanism for especially warning for such collisions.

`\DescribeResource`

The main command available is

`\DescribeResource{`⟨*type*⟩`}{`⟨*id*⟩`}{`⟨*text*⟩`}`

⟨*type*⟩ is a four-character string. Most special characters are treated as ordinary ones (very useful for `#`s), but the visible ASCII characters `%`, `{`, `\`, and `}` retain their usual meaning. To use such a troublesome character ⟨*c*⟩ in a resource type, write it as `\PrintChar{`\`\`⟨*c*⟩`}`. ⟨*id*⟩ is a TEX number; it will be used as the number of the resource. ⟨*text*⟩ is normal text that will be put in the index entry to describe the resource; it seems a good idea to use the name of the resource for this. ⟨*id*⟩ and ⟨*text*⟩ are read with normal LATEX catcodes. Note that ⟨*text*⟩ is a moving argument.

`\DescribeResource` does two things—it prints the ⟨*type*⟩ and ⟨*id*⟩ of the resource in the margin, and it writes an entry

⟨*type*⟩ resources:
    ⟨*id*⟩
        ⟨*text*⟩

(plus a lot of formatting not shown here) to the `.idx` file. The reference is for the page.

The idea with advancing `\count@` like that when constructing the index entry is to get a sort key for which lexicographic order equals the wanted order. This would not be the case if the number was simply written down. The current code maps numbers to six-digit positive integers, but five-digits integers would be sufficient (a resource ⟨*id*⟩ is a signed 16-bits integer). The construction chosen here furthermore puts the negative numbers after the positive ones.

```
⟨*rsrccmd⟩
\NewDescribeCommand\DescribeResource{%
   \XD@grab@harmless\relax \XD@grab@marg \XD@grab@marg
}{3}{%
   \GenericDescribePrint{#1%
      \textnormal{:\ifnum#2<\z@ \textminus\number-\else\number\fi#2}%
   }%
   \count@=#2\relax
   \advance \count@ 100000\ifnum \count@<\z@ 0\fi \relax
   \protected@edef\@tempa{%
      \noexpand\LevelSorted{\the\count@}{%
         \ifnum #2<\z@ \string\textminus \number-\else\number\fi#2%
      }%
   }%
   \IndexEntry{%
      \LevelSorted{#1 resources:}{\texttt{#1} resources:}%
      \@tempa
```

```
            \LevelSame{#3}%
         }{usage}{\thepage}%
      }
      ⟨/rsrccmd⟩
```

10 The `\DocInclude` command

The code in this section is based on code from the ltxdoc document class [1] and it implements a command called `\DocInclude`. Two implementations of this command are given: one which is essentially that of ltxdoc (preserving all its peculiarities), and one which is a reimplementation from scratch. The default is to use the latter, but passing the `olddocinclude` option to xdoc selects the former.

10.1 Old implementation

It should be observed that this is not a complete implementation of the `\DocInclude` command—it only redefines the ltxdoc macros that need to be changed if the `\DocInclude` command is to work with xdoc (it doesn't for example change the definition of `\DocInclude` itself). Furthermore it doesn't define anything if the ltxdoc document class hasn't been loaded, since then the details of the definition of `\DocInclude` (even if it would be defined) are unknown.

`\CodelineIndex`
`\filesep`
`\@docinclude`

ltxdoc redefines `\codeline@wrindex` so that `\filesep` is prepended to each codeline number that is written to the index file. That redefinition has no effect unless the `\CodelineIndex` command is executed afterwards however, so there is no harm in having `\CodelineIndex` itself apply the corresponding change.

```
⟨*pkg⟩
\@ifpackagewith{xdoc2}{olddocinclude}{%
   \@ifclassloaded{ltxdoc}{%
      \renewcommand\CodelineIndex{%
         \makeindex
         \let\XD@if@index=\@firstoftwo
         \codeline@indextrue
         \def\TheXDIndexNumber{\filesep\thecodelineno}%
      }%
```

The `\filesep` macro is redefined so that the docindex package [2] can use a `page_compositor` string different from the default - simply by redefining `\XD@page@compositor`. This redefinition has to be put in `\docincludeaux` since that macro redefines `\filesep` too.

```
      \expandafter\def \expandafter\docincludeaux \expandafter{%
         \docincludeaux
         \gdef\filesep{\thepart\XD@page@compositor}%
      }
```

The change to `\@docinclude` merely consists of inserting code for writing an `ExternalXRefWrap` to the `.aux` file to record the new value of the `part` counter.

```
      \def\@docinclude#1 {%
         \clearpage
         \if@filesw
            \immediate\write\@mainaux{\string\@input{#1.aux}}%
         \fi
         \@tempswatrue
         \if@partsw
            \@tempswafalse
            \edef\@tempb{#1}%
            \@for\@tempa:=\@partlist\do{%
               \ifx\@tempa\@tempb\@tempswatrue\fi
            }%
         \fi
         \if@tempswa
            \let\@auxout\@partaux
            \if@filesw
               \immediate\openout\@partaux #1.aux
               \immediate\write\@partaux{\relax}%
            \fi
            \part{#1.dtx}%
            \if@filesw
               \immediate\write\@partaux{\@percentchar\@percentchar
                  ExternalXRefWrap {\filesep} {}%
               }%
            \fi
            {%
               \let\ttfamily\relax
               \xdef\filekey{%
                  \filekey, \thepart={\ttfamily\currentfile}%
               }%
            }%
            \DocInput{#1.dtx}%
            \clearpage
            \@writeckpt{#1}%
            \if@filesw \immediate\closeout\@partaux \fi
         \else
            \@nameuse{cp@#1}%
         \fi
         \let\@auxout\@mainaux
      }
   }{}
}{}
```

10.2 New implementation

The default action of the second implementation is to be precisely an `\include` variant of `\DocInput`, but in addition to that it also has a (one-argument) hook called `\docincludeaux` which is executed before a file is actually `\DocInput`ted, but after it has been determined that it should be included, and this hook is only executed for the files which should be `\include`d. This hook is normally `\@gobble`, but passing the `fileispart` option to `xdoc` redefines it to start a new part and set the pagestyle.

`\DocInclude`
`\@docinclude`

Most of the code for the `\DocInclude` command is put in the `\@docinclude` macro; `\DocInclude` simply checks that it hasn't been nested. The main difference to `\include` is that a nested `\DocInclude` becomes an error plus the corresponding `\DocInput`, whereas a nested `\include` simply becomes an error. The rationale for this is that it is probably closer to what was intended.

The argument of `\@docinclude` is, oddly enough, space-delimited. This is inherited from the `\@include` macro in the LaTeX kernel, where it is a hack to make sure that the part `.aux` file that is opened for writing really gets the suffix `.aux` (in the worst case, TeX could start overwriting a `.tex` file instead).

```
\@ifpackagewith{xdoc2}{olddocinclude}{}{%
   \def\DocInclude#1{%
      \ifnum\@auxout=\@partaux
         \@latexerr{\string\include\space cannot be nested}{%
            Your \protect\DocInclude\space will be reduced to a
            \protect\DocInput.%
         }%
         \DocInput{#1.dtx}%
      \else \@docinclude#1 \fi
   }%
```

The only things in this `\@docinclude` that are not precisely as in `\@include` are the `\docincludeaux` and `\DocInput` commands.

```
   \def\@docinclude#1 {%
      \clearpage
      \if@filesw
         \immediate\write\@mainaux{\string\@input{#1.aux}}%
      \fi
      \@tempswatrue
      \if@partsw
         \@tempswafalse
         \edef\@tempb{#1}%
         \@for\@tempa:=\@partlist\do{%
            \ifx\@tempa\@tempb \@tempswatrue \fi
         }%
      \fi
      \if@tempswa
         \let\@auxout\@partaux
         \if@filesw
            \immediate\openout\@partaux #1.aux
            \immediate\write\@partaux{\relax}%
         \fi
         \docincludeaux{#1.dtx}%
         \DocInput{#1.dtx}%
         \clearpage
         \@writeckpt{#1}%
         \if@filesw \immediate\closeout\@partaux \fi
      \else
         \deadcycles\z@
         \@nameuse{cp@#1}%
      \fi
      \let\@auxout\@mainaux
   }%
}{}
```

`fileispart` option
`\docincludeaux`

The `fileispart` option works by (re)defining a couple of macros, of which the `\docincludeaux` macro is the most important. Its syntax is

`\docincludeaux{`⟨*filename*⟩`}`

where ⟨*filename*⟩ is the name of a file that will be inputted. The `fileispart` definition of this is to set `\currentfile` to the harmless character string of ⟨*filename*⟩, produce a `\part` heading whose text is that ⟨*filename*⟩, add the ⟨*filename*⟩ to the `\filekey` macro, set the page style to `docpart`, clear the `\filedate`, `\fileversion`, and `\fileinfo` macros, and write an `ExternalXRefWrap` XXR-command to the `.aux` file to record the new codeline number prefix.

```
\@ifpackagewith{xdoc2}{olddocinclude}{\iffalse}{
   \@ifpackagewith{xdoc2}{fileispart}{\iftrue}{
      \let\docincludeaux=\@gobble
      \iffalse
   }
} % If fileispart and not olddocinclude then
   \def\docincludeaux#1{%
      \MakeHarmless\currentfile{#1}%
      \part{\texttt{\currentfile}}%
      \pagestyle{docpart}%
      \let\filedate\@empty
      \let\fileversion\@empty
      \let\fileinfo\@empty
      \protected@xdef\filekey{%
         \filekey, \thepart=\texttt{\currentfile}%
      }%
      \if@filesw
         \immediate\write\@partaux{\@percentchar\@percentchar
            ExternalXRefWrap {\thepart\XD@page@compositor} {}%
         }%
      \fi
   }%
```

`\CodelineIndex` The `fileispart` option also adds the `codelineno` counter to the reset list for `part` and changes the format of codeline numbers written to the index.

```
   \@ifclassloaded{ltxdoc}{}{\@addtoreset{codelineno}{part}}%
   \renewcommand\CodelineIndex{%
      \makeindex
      \let\XD@if@index=\@firstoftwo
      \codeline@indextrue
      \def\TheXDIndexNumber{\thepart\XD@page@compositor\thecodelineno}%
   }%
```

`\partname`
`\thepart`
`\IndexParms`

Finally there are a couple of macros which are redefined for aesthetic rather than technical reasons. Passing the `fileispart` option sets `\partname` to `File`, sets `\thepart` to `\aalph{part}`, and adds a setting of pagestyle to `\IndexParms`. (The pagestyle setting is added to `\index@prologue` by ltxdoc, but I think `\IndexParms` is more appropriate.)

```
   \def\partname{File}
   \def\thepart{\aalph{part}}
   \expandafter\def \expandafter\IndexParms
      \expandafter{\IndexParms \pagestyle{docindex}}
```

In case the index formatting is handled by the docindex package [2] (or its LaTeX 2ε incarnation docidx2e), the above addition to `\IndexParms` won't have any effect. Therefore xdoc also passes the `usedocindexps` option on to these packages.

```
   \PassOptionsToPackage{usedocindexps}{docindex}
   \PassOptionsToPackage{usedocindexps}{docidx2e}
\fi
```

`\ps@docpart` `\setfileinfo` `\XD@set@file@info` The `docpart` pagestyle is for pages made from the `\DocInclude`d files. The page footers contain the page number, the part (file) number, and the current file name. It also contains the file date and version if that information is available.

ltxdoc uses `\GetFileInfo` to get the date and version information, but that's a very peculiar practice. The data one wants to present are about the file being typeset—typically the version of the package that is documented in this file—whereas the `\GetFileInfo` command really extracts information about *unpacked* classes, packages, and similar files—files that contribute to the typesetting by defining commands, not by containing text. Such information may be of interest for documents which contain alternative code for incompatible versions of for example a package, but it is of no use for printing version information as above since the version of a package used for typesetting a `.dtx` file need not be the version actually contained in that `.dtx` file. Thus the only way to make this work is by doing as the LaTeX kernel source and include `\ProvidesFile` commands for the `.dtx` file in each such file, which is a rather peculiar use of the `\ProvidesFile` command.

The `\setfileinfo` command provides an equivalent feature in a less roundabout way. It has the syntax

> `\setfileinfo[`⟨*date*⟩␣⟨*version*⟩␣⟨*info*⟩`]`

and it sets `\filedate` to ⟨*date*⟩, `\fileversion` to ⟨*version*⟩, and `\fileinfo` to ⟨*info*⟩ if the optional argument is present; if the optional argument is missing or contains fewer than three words then the missing fields are set to `?`.

```
\@ifpackagewith{xdoc2}{olddocinclude}{}{%
   \def\ps@docpart{%
      \def\@oddfoot{%
         File: \texttt{\currentfile}%
         \ifx \filedate\@empty \else \ Date: \filedate\fi
         \ifx \fileversion\@empty \else \ Version: \fileversion\fi
         \hfill\thepage
      }%
      \if@twoside
         \def\@evenfoot{%
            \thepage\hfill
            File: \texttt{\currentfile}%
            \ifx \filedate\@empty \else \ Date: \filedate\fi
            \ifx \fileversion\@empty \else \ Version: \fileversion\fi
         }%
      \else \let\@evenfoot\@oddfoot \fi
   }
```

The corresponding definition in ltxdoc (there it appears in `\docincludeaux`) is peculiar in that the odd page footer is set globally but the even page footer only locally.

The definition of `\setfileinfo` follows that of `\GetFileInfo` except for the fact that the `\relax`es have been replaced by `\@empty`s.

```
   \newcommand\setfileinfo[1][]{%
```

```
      \edef\@tempa{#1}%
      \expandafter\XD@set@file@info \@tempa\@empty? ? \@empty\@empty
   }
   \def\XD@set@file@info#1 #2 #3\@empty#4\@empty{%
      \def\filedate{#1}%
      \def\fileversion{#2}%
      \def\fileinfo{#3}%
   }
}{}
```

The reason for making the argument of `\setfileinfo` optional is that with the `\ProvidesFile` practice one can (potentially) put all date and version information in one place through tricks like

```
%    \begin{macrocode}
\ProvidesPackage{foobar}
%    \end{macrocode}
% \ProvidesFile{foobar.dtx}
    [2000/02/02 v1.0 Silly example package]
%
```

By making the argument of `\setfileinfo` optional, I make sure that people who have used such tricks only have to replace the `\ProvidesFile{foobar.dtx}` by `\setfileinfo`.

`\ps@docindex`
`\filekey`

The `docindex` pagestyle is for the index in `fileispart` documents. It prints a file key, which is a list of all the included files and their corresponding part letters, at the bottom of every page. The file key is stored in the macro `\filekey`, which should have been constructed file by file as they are included. To add a file to the file key, it is recommended that you do

`\protected@xdef\filekey{\filekey,` ⟨*entry for new file*⟩`}`

The `fileispart` version of `\docincludeaux` already does this. The initial value of `\filekey` is `\@gobble` so that the comma before the first entry is removed. The `\@empty` below is there in case no entry has been inserted.

```
% \@ifpackagewith{xdoc2}{olddocinclude}{}{%
   \def\ps@docindex{%
      \def\@oddfoot{%
         \parbox{\textwidth}{%
            \strut\footnotesize\raggedright
            \textbf{File Key:} \filekey\@empty
         }%
      }%
      \let\@evenfoot\@oddfoot
   }%
   \let\filekey\@gobble
% }
```

It should be observed that since `\ps@docindex` only sets the page style locally, the page style will revert to its previous setting at the end of the `theindex` environment. As that previous setting is probably that of the `docpart` page style, you might have to set the page style manually.

`\aalph`
`\@aalph`

`\aalph` is a variant of `\alph` which continues with the upper case letters for 27–52. It is defined by ltxdoc, so it is merely provided here.

```
\providecommand*\aalph[1]{\@aalph{\csname c@#1\endcsname}}
\providecommand*\@aalph[1]{%
   \ifcase#1\or a\or b\or c\or d\or e\or f\or g\or h\or i\or
          j\or k\or l\or m\or n\or o\or p\or q\or r\or s\or
          t\or u\or v\or w\or x\or y\or z\or A\or B\or C\or
          D\or E\or F\or G\or H\or I\or J\or K\or L\or M\or
          N\or O\or P\or Q\or R\or S\or T\or U\or V\or W\or
          X\or Y\or Z\else\@ctrerr\fi
}
```

In `source2e.tex` one can see that doc's standard `gind.ist` index style file won't sort the 35th file (part `I`) correctly since it causes makeindex to read an `I` as "upper case Roman numeral one", but I doubt very many people encounter that problem in their projects.

`\XD@page@compositor`

The `\XD@page@compositor` macro contains the string which is put between the parts of a composite number in the index; it corresponds to the `page_compositor` parameter of makeindex.

```
\providecommand*\XD@page@compositor{-}
```

11 Miscellanea

11.1 Some LaTeX 2_ε* stuff

`\BooleanFalse`
`\BooleanTrue`
`\NoValue`

These three macros are borrowed from the xparse package [6], where they work as the three values *boolean false*, *boolean true*, and *absence of value* respectively. The definitions are taken from xparse v 0.17 (1999/09/10).

```
\@ifundefined{BooleanFalse}{\def\BooleanFalse{TF}}{}
\@ifundefined{BooleanTrue}{\def\BooleanTrue{TT}}{}
\@ifundefined{NoValue}{\def\NoValue{-NoValue-}}{}
```

By using these macros (rather than some homegrown set of macros or tokens) for denoting these values here I hopefully simplify a transition to LaTeX 2_ε*, but I don't want to rely on LaTeX 2_ε* since it hasn't been released yet.

11.2 The `\meta` command

A reimplementation which has already (as of v 2.0k) found its way into the doc package is the one that the `\meta` command is made robust, but since some people might still have older versions of doc and since that feature is needed for `\describecsfamily`, I apply it here too. First I check whether the definition of `\meta` is the old non-robust definition, and only apply the fix if it is.

```
\begingroup
\obeyspaces%
\catcode`\^^M\active%
\gdef\@gtempa{\begingroup\obeyspaces\catcode`\^^M\active%
\let^^M\do@space\let \do@space%
\def\-{\egroup\discretionary{-}{}{}\hbox\bgroup\itshape}%
\m@ta}%
\endgroup
\ifx \meta\@gtempa
```

`\l@nohyphenation`

The new implementation needs a `\language` without any hyphenation patterns. By switching to that language, one can inhibit hyphenation in a piece of text regardless of what line-breaking parameter settings are in force when the paragraph is actually broken. This new language will be called `nohyphenation` and it is only allocated if it isn't already known (since some babel settings files already defines this `\language`).

```
  \@ifundefined{l@nohyphenation}{\newlanguage\l@nohyphenation}{}
```

`\meta`
`\meta@font@select`

This is the definition of `\meta` from doc v 2.0m. For an explanation of the implementation, se a `doc.dtx` at least that new or entry `latex/3170` in the LaTeX bugs database.

```
  \DeclareRobustCommand\meta[1]{%
     \ensuremath\langle
     \ifmmode \expandafter \nfss@text \fi
     {%
        \meta@font@select
        \edef\meta@hyphen@restore
           {\hyphenchar\the\font\the\hyphenchar\font}%
        \hyphenchar\font\m@ne
        \language\l@nohyphenation
        #1\/%
        \meta@hyphen@restore
     }\ensuremath\rangle
  }
```

```
  \let\meta@font@select=\itshape
\fi
```

`\MetaNormalfont`

The `\MetaNormalfont` command redefines `\meta@font@select` to do a `\normalfont` before the `\itshape`. It is useful if `\meta` is going to be used to make `\rmfamily` interjections in `\ttfamily` text.

```
\newcommand\MetaNormalfont{\def\meta@font@select{\normalfont\itshape}}
```

`\XD@harmless\meta`

This macro is needed for making `\meta` behave as described in the argument of `\describecsfamily`, i.e., in text which is going to be converted into a harmless character string.

```
\@namedef{XD@harmless\string\meta}#1{%
  \toks@=\expandafter{\the\toks@ \meta{#1}}%
  \XD@harmless@
}
```

11.3 The checksum feature

The checksum mechanism in doc is a remnant from the times when file truncation was a common problem and a mechanism for detecting this was a great help.[12] Today its main usefulness seems to lie in that it distinguishes versions of a file that are "being worked on" (where the checksum probably doesn't match) from versions of a file that are "polished and ready for upload" (someone has bothered to fix the checksum), and as it exists it might as well stay. There is a problem

---

[12] Even though I suspect that the recommended use of it—to put the checking `\Finale` at the end of the `.dtx` file—may have reduced its usefulness dramatically, as that `\Finale` would have been the one thing that surely disappears if the file is truncated.

however with files which do not contain TEX code, as simply counting backslashes quite probably isn't a good (or even reasonable) way of forming a checksum for these files (if the checksum turns out to be zero, doc will complain no matter what you do).

`\check@checksum`

For that reason, the `\check@checksum` macro is redefined to only write the "no checksum" warning to the log file if the checksum hasn't been set.

```
\renewcommand\check@checksum{%
   \relax
   \ifnum \check@sum=\z@
      \PackageInfo{doc}{This macro file has no checksum!\MessageBreak
         The checksum should be \the\bslash@cnt}%
   \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.\MessageBreak Try to get a correct version.}%
   \fi\fi
   \global\check@sum\z@
}
```

11.4 The `\theCodelineNo` situation

doc incorporates formatting of the value of the `CodelineNo` counter in the `\theCodelineNo` macro, which is a bit awkward since it prevents using this macro in making e.g. index entries. To get around this, xdoc introduces the alternative name `codelineno` for this counter so that `\thecodelineno` can produce the value representation without formatting.

`\c@codelineno`
`\cl@codelineno`
`\p@codelineno`
`\thecodelineno`

The control sequences connected to the `codelineno` counter are `\let` so that they refer to the same `\count` register as the `CodelineNo` counter. Note that `CodelineNo` isn't a proper LaTeX counter, so the macros `\cl@CodelineNo` and `\p@CodelineNo` are undefined. `\thecodelineno` is set to the default value for a new counter.

```
\@ifundefined{c@codelineno}{}{%
   \PackageInfo{xdoc2}{Overwriting codelineno counter}%
}
\let\c@codelineno=\c@CodelineNo
\let\cl@codelineno=\@empty
\let\p@codelineno=\@empty
\def\thecodelineno{\@arabic\c@codelineno}
```

`\PrintCodelineNo`

The `\PrintCodelineNo` command is the new recommended command for printing the formatted form of the codeline number counter. People who write their own `macrocode`-like environments should use `\PrintCodelineNo` instead of doc's `\theCodelineNo`.

```
\newcommand\PrintCodelineNo{\reset@font\scriptsize\thecodelineno}
```

`\theCodelineNo` Finally `\theCodelineNo` is redefined to reference `\PrintCodelineNo`. This is done for the sake of backwards compability; I didn't feel like redefining `\macro@code` just for the sake of changing the `\theCodelineNo` into a `\PrintCodelineNo`).

```
\def\theCodelineNo{\PrintCodelineNo}
```

```
⟨/pkg⟩
```

12 Problems and things to do

This section lists some problems that exist with the current implementations of commands in `xdoc`. The list is rather unstable—items are added as I realize there is a problem and removed when I find a solution—an in parts it is rather esoteric since most of the problems have only been found theoretically.

One of the less well-known features of the `\verb` command is that it automatically inhibits the known syntactic ligatures. There is no such mechanism implemented for the harmless character strings, so some (in TEX `macrocode` uncommon) character sequences (such as `` !` ``) may produce unwanted results. The quick hack to circumvent this is to use the `\SetHarmState` command to mark one of the characters involved as problematic, as the `\PrintChar` command is implemented so that the character it prints will not be involved in ligaturing or kerning. On the other hand, `doc` does nothing to suppress syntactic ligatures in macro or environment names when they are printed in the margin, so for that material the `xdoc` implementation might actually improve things, although it could perform worse for verbatim material in the index and list of changes.

Things to do and/or think about:

- Examine how complicated it would be to convert the `\PrintChar` commands for visible characters in a harmless character string back to explicit characters, for possible use in sort keys. (This could be used to ensure that visible characters are sorted in strict ASCII order.)
- Should those "letters" which are commonly used as word separators—in LATEX code mainly `@`—be ignored when sort keys are being formed (just like the backslash is)? (This would require a change in the implementation of the `macro` environment.)

  A mechanism for doing this is included as of prototype version 2.1.
- Examine how much more efficient it would be to put temporary additions to the index exclude list in a separate list instead of the main list. This could be advantageous for deeply nested `macro` environments, as TEX will otherwise store as many (almost identical and often rather long) copies of the exclude list as there are nested environments.

  When asked about it, Frank Mittelbach didn't think there was any gains worth mentioning in this. On the other hand it might be worth investigating reimplementations that avoid calling `\trivlist` at the beginning of each `macro`-like environment when they are nested, since `\trivlist` does quite a lot of assignments.

- In an automatically generated index one often faces the problem that the entries at the innermost level are best formatted in one way when there is only one, but in a completely different way when there are several of them. To get optimal formatting in both cases, one would like to let the `\item`, `\subitem`, `\subsubitem` or corresponding macros detect the situation in this respect and choose the optimal formatting at each case.

  A mechanism for this is implemented by the docindex package.

References

[1] David Carlisle: *The file `ltxdoc.dtx` for use with LaTeX 2ε*, The LaTeX3 Project; CTAN: `macros/latex/base/ltxdoc.dtx`.

[2] Lars Hellström: *The docindex package*, 2001, CTAN: `macros/latex/exptl/xdoc/docindex.dtx`.

[3] Alan Jeffrey, Sebastian Rahtz, Ulrik Vieth (and as of v 1.9 Lars Hellström): *The fontinst utility*, v 1.8 ff., documented source code, CTAN: `fonts/utilities/fontinst/source/`

[4] The LaTeX3 Project: *LaTeX 2ε for class and package writers*, The LaTeX3 Project; CTAN: `macros/latex/base/clsguide.tex`.

[5] Frank Mittelbach, B. Hamilton Kelly, Andrew Mills, Dave Love, and Joachim Schrod: *The doc and shortvrb Packages*, The LaTeX3 Project; CTAN: `macros/latex/base/doc.dtx`.

[6] Frank Mittelbach, Chris Rowley, and David Carlisle: *The xparse package*, The LaTeX3 Project, 1999. Currently not available by anonymous FTP, but available by HTTP from `www.latex-project.org` (look for "experimental code").

Change History

prot1
  General: Started writing first prototype. (LH) 1

prot2
  General: Began work on the second prototype. (LH) 1
    Lots of utility macros were removed: `\XDWrapText`, `\XDAltWrapText`, `\XDSortAndText`, `\MultipleApply`, `\ApplicableUsageIndex` and `\XD@index@family`. (LH) 52
    I finally decided that it would be better to make the XXR-commands look like comments to TeX. (LH) 30
  `\add@specials`: Redefinition added. (LH) 18
  `\describecsfamily`: Renamed `\DescribeCSFamily` and incorporated the code from `\XD@index@family`. (LH) . . . 58
  `\describeoption`: Command added—I realised that the need to describe options is probably as large as that to mark out their definition. (LH) 58
  `\GenericDescribePrint`: `\leavevmode` and `\ignorespaces` moved to `\NewDescribeCommand`. (LH) . 55
    Added `\strut`. Removed it from arguments passed to `\GenericDescribePrint`. (LH) 55
  `\MetaNormalfont`: Removed ro-

bustness; protected it explicitly wherever needed instead. (LH) 68

\NewDescribeCommand: Command added. (LH) 57

\NewMacroEnvironment: Command added. (LH) 51

Changed syntax in conformity with the syntax change in \XD@m@cro. (LH) 51

\SendExternalXRefMsg: Added \if@filesw test. (LH) 30

\SetCharProblematic: Command added. (LH) 18

\syncexternalxref: New name for \SendExternalXRefSync. Also added \if@filesw test. (LH) 32

\XD@endgrab: The grabbed arguments are no longer returned wrapped up in a group. There is no longer a need for storing the base call separately in \toks 2. (LH) 44

\XD@harmless@v: Moved code for adding to \toks@ here and changed it to append the contents of \XD@harmless@32, not necessarily a \PrintChar. (LH) 15

\XD@m@cro: Put the ⟨*changes*⟩ argument before the ⟨*assign*⟩ argument. Executing the ⟨*assign*⟩ code after the ⟨*changes*⟩ \edef. Changed the descriptions of these arguments a little. (LH) 50

\XD@situation: Changed to allow multiple \XD@situation commands for the same situation. This is necessary for coping with documents which \include files. (LH) 36

Made redefinition at begin document global. (LH) 36

\XDMainIndex: New name and syntax for \ApplicableMainIndex. (LH) 52

\XDStackItemLabels: Made it work like a \vtop (but hide the height) if \XD@macro@dimen is -\maxdimen. (LH) 47

prot2.1

General: Additional \NewDescribeCommand code example added. (LH) . . 59

\theCodelineNo situation cleared up. (LH) 69

Sort key making commands added. (LH) 21

\check@checksum: Redefinition added. (LH) 69

\CodelineIndex: Using \thecodelineno. (LH) 21, 61, 64

\describecsfamily: Using \MakeSortKey. (LH) 58

\DescribeMacro: Using \MakeSortKey. (LH) 57

\DoNotIndex: Also changing catcode of %. (LH) 28

\LevelSame: New name for \levelsame. (LH) 19

\LevelSorted: New name for \levelsorted. (LH) 19

macro: Using \MakeSortKey to make index entry. (LH) 53

\NewMacroEnvironment: Introduced star form with different semanics for the ⟨*print*⟩ argument. This uses the helper macros \XD@NewMacroEnvironment and \XD@NewMacroEnvironment@. (LH) 51

\PrintVisibleChar: Made it possible to select the alternative defintion of \PrintVisibleChar through an xdoc package option. (LH) . . 11

\StepPageSituation: Now also setting \IfOddPageSituation, instead of having that macro performing the test each time it is used. This fixes a rarely occuring bug which occurs when a page is shipped out between \StepPageSituation and a corresponding \IfOddPageSituation. (LH) . 35

switch: Using \MakeSortKey. (LH) 53

\XD@grab@harmless@oarg: Macro added. (LH) 45

\XD@grab@harmless@oarg@: Macro added. (LH) 45

\XD@m@cro: Removed \XDToMargin from the argument of \item. It should now be included in #1 instead. (LH) 50

\XD@paren@PrintChar: \number added. (LH) 28

\XD@special@index: Using \MakeSortKey to make the sort key. (LH) 25
\XD@threedignum: Braces inserted by \XD@threedignum are given catcode other. (LH) 10
\XDParToMargin: Command added. (LH) 49
\XDToMargin: New name for \XD@to@margin. (LH) 48

prot2.2
\@wrglossary: Redefinition added. (LH) 40
\@wrindex: Redefinition added. (LH) 25
\filesep: Redefined to use \XD@page@compositor. (LH) 61
\SortIndex: Redefinition added. (LH) 23
\XD@glossary@keyword: Macro added. (LH) 40
\XD@index@keyword: Macro added and ⟨*encap*⟩ macros changed to use it. (LH) 21
\XD@page@compositor: Macro added, other macros changed to use it. (LH) 67

prot2.3
\GenericDescribePrint: Changed formatting to match that of \XDParToMargin. (LH) 55
\XD@grab@sarg: Macro added. (LH) 44

prot2.4
\changes@: Added support for \uintver. (LH) 40
\if@version@key@: Switch added. (LH) 39
\uintver: Command added. (LH) 40
\XD@usechange: Added support for \uintver. (LH) 41

prot2.5
General: Reregeristing options in case they were global. (LH) 8

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.