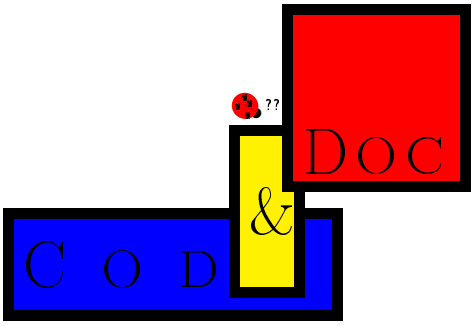
The CodeDoc class
v.0.1
2009/04/01

Paul Isambert
zappathustra@free.fr



CodeDoc is a class designed to produce LaTeX files such as packages and classes along with their documentations. It does not depart from LaTeX's ordinary syntax, unlike e.g. DocStrip, allows any existing class to be loaded with its options and offers various fully customizable verbatim environments that allows authors to typeset the code and documentation of their files as they want. To create the documentation, we compile the document as usual; to create the external file(s), we simply put `produce` in the class options and compile as before.

Despite my earliest expectations, CodeDoc is not better than DocStrip. It is simply different. If you want a well-delimited approach to literate programming, use DocStrip. On the other hand, CodeDoc is more natural, in the sense that it is ordinary LaTeX all the way down. Note that you can 'mimick' DocStrip, either by putting any character at the beginning of each line of your code and setting the `\Gobble` parameter to 1 (this would be 'inverted DocStrip'), or by setting the comment character to be of category 9 ('ignored') and beginning each line of the documentation with this character. In this latter case, only commands that are considered by CodeDoc when producing a file should not be commented out... but I'm going too fast here, and you should learn the basics first...

CodeDoc is still in its infancy, as indicated by its version number. Although it has passed the test of producing this documentation, countless bugs will probably be reported, and meaningful suggestions will be made. Be patient, and send them to me. *Any reported bug and meaningful suggestion will be rewarded by a musical note, played by a virtual instrument, and sent in the mp3 format.* Isn't it amazing? I know it is. I will have to hire musically educated secretaries to face the consequences of such a reckless proposition. But it is worth it. Once a stable version is reached, I might even write a symphony.[1]

Some of the ideas of this class are not mine; some were inspired by others; some are mine but were independantly implemented in other places; may all these people be thanked, as well as all the verbatim wizards around the world. And, oh, yeah, some ideas are mine, too.

[1] 'Meaningful suggestion' and 'stable version' are fuzzy terms, you complain. Of course they are. Give me a chance!

Contents

Part I
User's manual

1 Code & Documentation

The source of this documentation looks roughly like this:

```
\documentclass[article(a4paper),
%produce,
]{codedoc}

Preamble of the document

\begin{document}
\section{Code \& Documentation}
The source of this document...

\ProduceFile{codedoc.cls}[codedoc][v.0.1][2009/03/13]
\begin{code}
  Material here will be written to codedoc.cls
  and typeset verbatim in the documentation.
\end{code}

\ShortCode/
/
  This too...
/

\begin{invisible}
  This material will be written to codedoc.cls
  but not typeset in the documentation.
\end{invisible}

\end{document}
```

Everything between `\begin{code}` and `\end{code}` is written verbatim to the dvi file. It is also the case for everything between two `\ShortCode` symbol, in this example '/'. Finally, if the comment sign at the beginning of the second line were removed, thus enabling the `produce` option, then this code would be written to `codedoc.cls` and no documentation would be produced. This is CodeDoc's basic mechanism. Let's review it more precisely. In what follows, I will say 'normal mode' if the `produce` option is not turned on, that is when we're typesetting the documentation, and 'produce mode' otherwise, that is when `produce` is present among the class options and CodeDoc is used to create an external file.

The first two sections of this manual explain how CodeDoc works and provide many examples. The third section lists all commands in alphabetical order, and explains what they do in each mode in a more systematic fashion.

1.1 Writing code

- `\ProduceFile{`⟨*File*⟩`}[`⟨*File name*⟩`][`⟨*File version*⟩`][`⟨*File date*⟩`]`
  In normal mode, this macro provides four commands: `\FileSource` stores ⟨*File*⟩, and the next three arguments are stored in `\FileName`, `\FileVersion` and `\FileDate` respectively. Those are optional, as indicated by their syntax.

In produce mode, CodeDoc opens ⟨*File*⟩ and writes to it everything in a `code` environment. `\FileName`, `\FileVersion` and `\FileDate` are also provided and may be used in `\Header` (see below) or in the file itself with `\CodeEscape` (see page 15). Thus, you can avoid mismatches between your documentation and the `\ProvidesPackage` declaration, for instance.

- `\CloseFile{`⟨*File*⟩`}`

  In produce mode, when the `autoclose` option is on, `\ProduceFile` closes the file that was currently under production, if any. But you might want to keep a file open, in case you're writing to several files at the same time. That's why CodeDoc's basic behavior is to keep all files open. Thus

```
\ProduceFile{myfile}
\begin{code}
\def\foo{%
\end{code}

\ProduceFile{myotherfile}
\begin{code}
\relax
\end{code}

\ProduceFile{myfile}
\begin{code}
  FOO}
\end{code}
```

  will write `\def\foo{% FOO}` to `myfile` and `\relax` to `myotherfile`. This might not be very good practice, but who knows? that might be useful when building a complicated package.

  But TEX cannot keep open as many files as one wants. Actually, CodeDoc will start complaining when more than 16 files are simultaneously in production. `\CloseFile` is used to close those whose production is over and allocate their streams to new files.  Do files flow?

- `code`

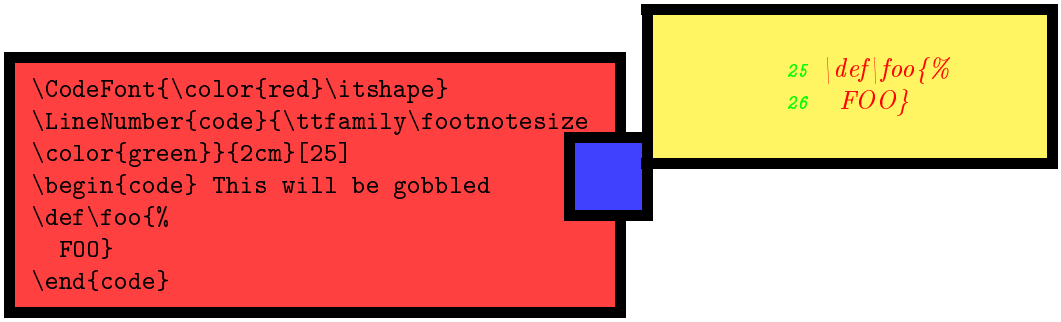
  This is the basic environment that writes its content to an external file in produce mode or displays it verbatim in the documentation in normal mode. There is nothing much to say. Each line is numbered, as one generally wants the implementation of a code to be. One important thing is that everything on the line after `\begin{code}` will be gobbled. `\end{code}` can appear wherever you want.

- `\CodeFont{`⟨*Font specifications*⟩`}`

  The font of the `code` environment may be changed with `\CodeFont` (by default, it's `\ttfamily`). Since everything is in a group, you can use 'spreading commands'.

- `\LineNumber{code}{`⟨*Font specifications*⟩`}{`⟨*Width*⟩`}[`⟨*Number*⟩`]`

  This sets the style of the line number, the width of the box it is put in (by default, it's `0pt`, so numbers are in the left margin), and the starting value. The first argument is `code` and not ⟨*code*⟩, because `\LineNumber` is a macro that applies to all `example` environments (see the next section), and its first argument is the name of the environment to modify. By default, `code` is not an `example` environment (although it might be redefined as such) but this command is nonetheless available.



Note that \LineNumber inherits the specification of \CodeFont that it doesn't override, in this example the italic shape. The \color command does *not* belong to CodeDoc, but to the xcolor package. If you want to do really interesting things with code, it is better to redefine it as an example (see next section).

As usual with verbatim environments, there exists a starred version of code that shows spaces.

- invisible

  In normal mode, everything in a invisible environment is skipped. In produce mode, however, the material is written to the file in production. This is useful to write code you don't want to comment in the documentation, like specifications at the beginning of the file or repetitive macro definitions. As you might imagine, there is no starred variant.

- \Header{⟨*Text*⟩}

  In produce mode, unless the noheader option is on, CodeDoc writes the following at the beginning of every file:

  ```
  % This is ⟨\FileName⟩, produced by the CodeDoc class
  % with the 'produce' option on.
  %
  % To create the documentation, compile ⟨jobname.tex⟩
  % without the 'produce' option.
  %
  % SOURCE: ⟨File (\input in File...)⟩
  % DATE: ⟨\FileDate⟩
  % VERSION: ⟨\FileVersion⟩
  ```

  where \FileName, \FileDate and \FileVersion are set by \ProduceFile. The '\input in file' part is optional and recursive, depending on files \input in your document. With \Header, you can change this and print ⟨*Text*⟩ instead. In ⟨*Text*⟩, ends of line are obeyed, and a comment sign followed by a space will start every line. Comment signs are normal sign. \Header should appear before \ProduceFile.

- \AddBlankLine

  In produce mode, CodeDoc writes a blank line to the file under production. Useful to delimit macros.

- \TabSize{⟨*Number*⟩}

  This is the number of spaces by which a tabulation will be represented in verbatim context. Default is 2. In produce mode, however, tabs are written as tabs, so this parameter has no effect.

- \Gobble{⟨*Number*⟩}

  The number of characters that will be gobbled at the beginning of each line. This works both in normal mode and in produce mode. This might be useful to indent code lines to make them more visible. When gobbling, a tab is considered as a single character and not as n characters, n being the value of \TabSize.

  A totally blank line is written as a totally blank line in both modes, i.e. CodeDoc does not fill its need for gobbled characters on the next line. The \end{code} line doesn't need to be indented, although it can be. If there are more characters than the value of \Gobble before \end{code}, then a new line is created.

```
\TabSize{3}
\Gobble{2}
My own value for \TabSize is 2,
hence the 2-space tab here,
but in the right panel it's 3
\begin{code*}
12\foo␣␣␣\foo

12\foo  \foo
\end{code*}
```

```
1 \foo␣␣␣\foo
2
3 \foo   \foo
```

- \BoxTolerance{⟨*Dimension*⟩}
  Verbatim lines often go into the right margin. This is the threshold above which TEX reports an overfull box. Default is 0pt.

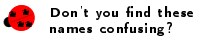
1.2 Macros to describe macros

Most of the commands in this section are similar to those in DocStrip. CodeDoc has an indexing mechanism that simply uses MakeIndex; if the `index` option is on, the `makeidx` package is loaded and \makeindex is executed. This also means that \printindex is available. CodeDoc does not require a special style file for MakeIndex. Thus, users can compile a documentation made with CodeDoc with MakeIndex's default specifications.

- \DescribeMacro{⟨*Macro*⟩}
- \DefineMacro{⟨*Macro*⟩}
  These commands print their argument according to \PrintMacro (see below). The first token is \string'ed,[2] so it can be a control sequence. They also create an index entry with the first token, and here lies their difference: they print the page number differently to distinguish whether a macro is described or defined (in the implementation). By default `described` macros have normal page numbers while `defined` macros have theirs in italics. This is not conventional, I agree, but it can be changed.[3]

- \DescribeEnvironment{⟨*Environment*⟩}
- \DefineEnvironment{⟨*Environment*⟩}
  This is similar to the `macro` version above, except that the entry is followed by '(environment)' in the index.

- \DescribeIndexFont{⟨*Font specifications*⟩}
- \DefineIndexFont{⟨*Font specifications*⟩}



  This sets how the page numbers are printed for `described` and `defined` macros (and environments) respectively. {⟨*Font specifications*⟩} should be commands like \ttfamily and not argument-taking commands like \texttt. You know that if you use MakeIndex.

- \PrintMacro{⟨*Macro or environment*⟩}
  This is the command that typeset the (\string'ed) macro. It takes one argument. It is shown here not to use it as is but to redefine it. Its default definition is:

```
\def\PrintMacro#1{%
  \noindent%
  \marginpar{\raggedleft\strut\ttfamily#1}%
  \ignorespaces}
```

[2] Verbatim text does not break by itself. I've used \VerbCommand here (see below) to include a discretionary.
[3] Since CodeDoc doesn't index macros when *used* in the code, I've found this choice more readable.

That is, it puts the macro in the margin. (Obviously, it was redefined in this documentation.) To achieve the same effect as with DocStrip, the following command is needed.

- \DocStripMarginpar
  This reverses marginpar and sets the right value for \marginparpush and \marginparwidth. They weren't included by default because you have the right to do what you want with your margins.

- \IgnorePrefix{⟨*Macro prefix*⟩}
  Many package and class authors prefix their internal commands with a string of letters to avoid clashes with other packages. For instance, if one writes a package mypack, one may name all internal commands \mp@foo, \mp@boo, \mp@moo, etc. Unfortunately, when indexed, they will all end up in the 'M' letter, whereas one might want to have them sorted without the prefix, with \mp@foo indexed as if it was \foo, etc. This is what \IgnorePrefix does; when sorting entries produced by \DescribeMacro and \DefineMacro, ⟨*Macro prefix*⟩ is ignored, although it is printed of course as part of the name. In our example, one would say \IgnorePrefix{mp@}. This command has two restrictions: first, ⟨*Macro prefix*⟩ should be no more than 8 characters long; second, any macro described with \DescribeMacro or \DefineMacro should have as many characters as \IgnorePrefix, 3 in our example. A simple way to circumvent the latter shortcoming is to temporarily define ⟨*Macro prefix*⟩ as an empty string:

```
\IgnorePrefix{mp@}
\DefineMacro\mp@foo Will be indexed as \foo

\DefineMacro\fo This will cause an error message

\IgnorePrefix{}
\DefineMacro\fo This is perfectly ok
\IgnorePrefix{mp@}
```

  You can have several \IgnorePrefix specifications, they are effective for the macros that follow them. For instance, some macros in CodeDoc are prefixed with cd@@, and when I define them in this documentation I specify \IgnorePrefix{cd@@} and then immediately \IgnorePrefix{cd@}, which is the normal prefix.

- \PrintPrefix{⟨*Macro prefix*⟩}
  Like \PrintMacro, this command is not shown here to be used but to be redefined. It is put just before ⟨*Macro prefix*⟩ when printing the index, so that you can typeset it differently. For instance, most CodeDoc's internal macros are prefixed with cd@. I have specified \IgnorePrefix{cd@} for this documentation and defined \PrintPrefix as \def\PrintPrefix{\textcolor{gray}} so that all prefixes are printed in gray (thanks to the xcolor package). For instance, \cd@BadChar is printed \cd@BadChar in the index (which you can verify if the obeystop option is commented out, thus including the implementation in the documentation). Obviously, \def\PrintPrefix#1{\textcolor{gray}{#1}} would have been equally efficient. Just note that since \PrintPrefix is \let to \relax by default, you have to use \newcommand and not \renewcommand when defining it for the first time, in case you prefer LaTeX's command definitions.

- \meta{⟨*Argument*⟩}
- \marg{⟨*Argument*⟩}
- \oarg{⟨*Argument*⟩}
- \parg{⟨*Argument*⟩}

  These are well-known. In case you've forgotten:
  \meta{Argument} ⇒ ⟨*Argument*⟩
  \marg{Mandatory argument} ⇒ {⟨*Mandatory argument*⟩}
  \oarg{Optional argument} ⇒ [⟨*Optional argument*⟩]
  \parg{Picture argument} ⇒ (⟨*Picture argument*⟩)

- \bslash
  Everybody needs a backslash. This one is meant to print equally well in usual contexts and in PDF bookmarks created by hyperref, if any. So it can be used in titles without restriction.

- \StopHere{⟨*Code*⟩}
  If the obeystop command is on, CodeDoc will execute ⟨*Code*⟩ and then \end{document}, otherwise nothing happens. If the index option is also on, \printindex will be automatically executed after ⟨*Code*⟩. This is useful to let the user print a version of the documentation with some part(s) left out, typically the implementation.

1.3 Choosing the class

CodeDoc by itself defines nothing that one wants a class to define. It lets the user call the desired class. To do so, just add the name of the class in the options of the \documentclass declaration. If you want the class to load options itself, put them after the name of the class, between parenthesis, and separated by semi-colons. Thus, \documentclass[memoir]{codedoc} loads the memoir class without options while \documentclass[memoir(a4paper;oneside)]{codedoc} loads it with the a4paper and oneside options.[4]

By default, CodeDoc loads the article class without options.

1.4 Dangerous strings 

In produce mode, CodeDoc becomes a string tester and nothing else. Hence, there are strings you don't want it to see because you don't want it to execute them. For instance, you don't want \end{document} to be executed unless at the end of the document. So when you say \verb+\end{document}+, you want CodeDoc to identify that \end{document} is not for real. Fortunately, CodeDoc does so. To some extent.

More precisely, CodeDoc identifies its own verbatim commands (described in the next section), LaTeX's \verb and verbatim environment, as well as verbatim environments created with the fancyvrb package and the 'short verb' characters defined with \DefineShortVerb from the same package. Thus, you can safely use fancyvrb and its companion fvrb-ex.

However, \begin's and \end's are not the only strings that must be used carefully. The most important things you want CodeDoc to ignore in case they shouldn't be executed are its own macros. For instance, you don't want \ProduceFile to be executed when there's no reason to do so. But, unless you're documenting CodeDoc itself, what might be the situation where \ProduceFile is executed wrongly? Simply if you use it in a statement with \let, \def, \newcommand, etc. In produce mode, CodeDoc does not recognizes these commands and for instance in \let\ProduceFile\mycommand, \let will be skipped and \ProduceFile executed. Hence the following.

- \DangerousEnvironment{⟨*List of environments*⟩}
  Whenever you want CodeDoc to skip an environment in produce mode, for instance because it's a verbatim environment designed by yourself, you can add its name to \DangerousEnvironment. If you add more than one name, use commas as separators.

- \StartIgnore
- \StopIgnore
  In produce mode, when CodeDoc encounters \StartIgnore, everything is skipped until \StopIgnore is found. This is useful to hide parts of your document that are irrelevant to the file you're building in produce mode (which is probably contained in the 'implementation' section). You should be careful to define your example environments and other verbatim devices outside the skipped material, if you want CodeDoc to identify them properly when it stops ignoring things.

[4] This means that if you specify an unknown option for CodeDoc, it will try to load an (probably) unknown class, and you will get the corresponding error message.

```
\DangerousEnvironment{myenv,myotherenv}
\begin{myenv}
\end{document} This will be skipped by CodeDoc
\end{myenv}

\StartIgnore
\let\ProduceFile\myproduce This too, but that will be taken into
                           account in normal mode
\StopIgnore
```

However, you should be aware of the following points:

- *Any command that has some effect in produce mode should appear verbatim in your document.* Conversely,
- *Commands that have some effect in produce mode cannot be redefined for that mode.* And when I say 'cannot', I mean 'you can try, it won't work'. This leads to the final principle:
- *You can redefine a command to have the desired effect in normal mode as long as you respect its arguments, so that it can work properly in produce mode. And this should be done between* `\StartIgnore` *and* `\StopIgnore`*, of course.*

For instance, you can say:

```
\StartIgnore
\renewcommand\CloseFile[1]{End of #1\clearpage}
\StopIgnore
```

and when you say `\CloseFile{myfile}`, '`End of myfile`' will be printed to the documentation, and a new page will be created, while in produce mode CodeDoc will do its usual job. On the other hand, although `\let\cf\CloseFile` is meaningful in normal mode, in produce mode it won't take effect, i.e. CodeDoc won't close anything. Finally, the previous example would have been catastrophic without `\StartIgnore` and `\StopIgnore`, because in produce mode, CodeDoc would have tried to execute `\CloseFile`.

`\StartIgnore` and `\StopIgnore` are also useful to make CodeDoc go faster and avoid errors, if you use it with `\input`. For instance, the following file would be perfect, provided everything that should be written to an external file is contained in `implementation.tex`

```
\documentclass{codedoc}
Write your verbatim definitions here, so that CodeDoc can see them
\begin{document}

\StartIgnore
\input{documentation}
\StopIgnore
\input{implementation}

\end{document}
```

This example leads us to the final restriction:

- *You should use* `\input` *in the LaTeX's way, i.e.* `\input{myfile}`*, and not in TeX's original way, i.e.* `\input myfile`*, if the file in question is to be read in produce mode.* In the example above, `documentation` can be `\input` as you want, but `implementation` should be `\input` as shown.

To know what commands have some effect in produce mode, see the summary of commands.

2 Verbatim Madness

2.1 Example environments

- example
- \CodeInput
- \CodeOutput

At first sight, the example environment is totally useless. Indeed, the following code does nothing:

```
\begin{example}
\TeX
\end{example}
```

However, it provides two commands \CodeInput and \CodeOuput. The former prints the code verbatim (and in typewriter font), and the latter executes it. So in the end it's very useful to document your package or class, because it avoids typing the code twice (and therefore errors are avoided).

```
\begin{example}
\TeX
\end{example}
\CodeInput\noindent yields \CodeOutput.
```

```
\TeX
yields TEX.
```

The example environment is just one instance of a family of environments that you can create by yourself with the following commands.

- \NewExample[⟨*Options*⟩]{⟨*Name*⟩}{⟨*Code input*⟩}{⟨*Code output*⟩}{⟨*Immediate execution*⟩}
- \RenewExample[⟨*Options*⟩]{⟨*Name*⟩}{⟨*Code input*⟩}{⟨*Code output*⟩}{⟨*Immediate execution*⟩}

These two macros (whose difference is similar to the one between \newcommand and \renewcommand) create an environment ⟨*Name*⟩ that will provide two commands, \CodeInput and \CodeOutput, whose effect is defined by ⟨*Code input*⟩ and ⟨*Code output*⟩. Moreover, ⟨*Name*⟩ will execute ⟨*Immediate execution*⟩.

⟨*Code input*⟩ and ⟨*Code output*⟩ have a peculiar syntax. The code to be processed is represented by #. For instance, the example environment is defined as:

```
\NewExample{example}     This is ⟨Name⟩
           {\ttfamily#}  \CodeInput yields but the code in typewriter font
           {#}           \CodeOutput simply executes the code
           {}            Nothing is done when example is called
```

You can do whatever you want. The code, represented by #, may be the argument of a macro. For instance:


```
\NewExample{myex}
           {\ttfamily\underline{#}}
           {}
           {\CodeInput}
\begin{myex}
Hello, world!
\end{myex}
```

Hello, world!

What does `myex` do? It sets the verbatim code in typewriter font and underlines it (which is admittedly not the most interesting thing you can do). ⟨*Code output*⟩ is empty, so `\CodeOutput` will yields nothing. Finally, ⟨*Immediate execution*⟩ calls `\CodeInput`, so there's no need to call it after the environment.

The following points apply:

- All environments thus defined have a starred variant that shows spaces as characters.
- `\CodeInput`, `\CodeOuput` and ⟨*Immediate execution*⟩ are groups, so you can put any command in them, they won't spread. For instance, in `myex` above, there's no need to add a group to restrict the application of `\ttfamily`.
- `\CodeOuput` *really* executes your code. Any error will appear as such.
- Since `\CodeOuput` is a group, the definition you make won't work for the rest of your document, unless you make them global. For instance:

```
\NewExample{myex}{}{#}{}

\begin{myex}
\def\foo{FOO!}
\end{myex}

\CodeOutput
\foo
```

will yield an error message, because `\foo` was only locally defined in `\CodeOuput`.

- Everything on the same line after the `\begin` statement of an environment will be gobbled.
- By default, CodeDoc does not add any space or `\par` before `\CodeInput`, `\CodeOuput` and ⟨*Immediate execution*⟩. A `\par` is added after `\CodeInput` if and only if the `\end` statement appears on its own line. Here's an illustration:

```
\NewExample{myex}{\ttfamily#}{#}{}
\parindentOpt

\begin{myex}
\TeX
\end{myex}
+\CodeInput+ yields +\CodeOutput+

\vskip1em
\begin{myex}
\TeX\end{myex}
+\CodeInput+ yields +\CodeOutput+
```

```
+\TeX
+ yields +TEX+

+\TeX+ yields +TEX+
```

- The `code` environment can be freely redefined as an example environment.
- All example environments obey `\TabSize` and `\Gobble` as defined in the previous section, as well as `\LineNumber` if they are numbered (see below). See the description of `\eTeXOff` and `\eTeXOn` below for a comment on `\Gobble`.

⟨*Options*⟩ may be one or several of the following (separated by commas):

`numbered`
Each line of `\CodeInput` is numbered. The count starts back to 1 at each occurrence of the environment.

`continuous`
Each line of `\CodeInput` is numbered. The count starts where the last occurrence of the same environment left. As an (utterly boring) example:

```
\NewExample[numbered]{myex1}{\ttfamily#}
  {}{\CodeInput}
\LineNumber{myex1}{\itshape}{1em}
\NewExample[continuous]{myex2}{\ttfamily
  \color{red}#}{}{\CodeInput}
\LineNumber{myex2}{\itshape}{2em}
\begin{myex2}
First line
Second line
\end{myex2}
\begin{myex1}
First line
Second line
\end{myex1}
\begin{myex2}
Third line
Fourth line
\end{myex2}
```

```
  1 First line
  2 Second line
1 First line
2 Second line
  3 Third line
  4 Fourth line
```

visibleEOL

This is more complicated and requires some knowledge of how CodeDoc builds examples.

Although you might not know it, your TEX distribution is very probably running on ε-TEX. That's the reason why CodeDoc can process some code verbatim and executes it at the same time, as in the example environments, without the need for an external file. This is completely impossible with the original implementation of TEX. If, for some reason, you don't have ε-TEX, or you're not running on it, then CodeDoc will use an external file.

However, ε-TEX's 'virtual external file' mechanism is not perfect, and CodeDoc has to cope with it. What happens is that when you use `\CodeOutput`, CodeDoc hacks your code a little in order to simulate a real TEX code; namely, before anything is processed, CodeDoc removes ends of lines and commented parts of lines. For instance, if you say:

```
\def\foo{%
  FOO!}
\foo
```

what CodeDoc really processes with `\CodeInput` is `\def\foo{FOO!}␣\foo`. Most of the time, that's exactly what you want. But it might happen that you're toying around with ends of lines or comment characters, and in that case everything will go wrong, as in:

```
\catcode`\%=12
I'm writing a % sign.
```

This will *not* produce 'I'm writing a % sign', because CodeDoc will remove everything from the comment sign to the end of the line, so that what `\CodeOuput` will try to execute is:

```
\catcode`\
I'm writing a
```

and of course the aborted `\catcode` declaration will yield an error message. To avoid this problem, the `visibleEOL` option makes CodeDoc keeps everything. But now there's another issue: comments and end of line characters are processed at the same time as other macros and aren't interpreted independantly as in normal TEX. For instance, the following code, if the `visibleEOL` option is on for the environment in question, will apply `\emph` to the end of line character and not to `A`.

```
\emph
A
```

So you should be sure that comments and line ends occur where they won't hinder anything. If you find this utterly complicated, then you can use an external file whenever you're hacking ends of line, thanks to the following two macros.

- `\eTeXOff`
- `\eTeXOn`

The former makes CodeDoc process all examples environments with an external file (whose extension is `.exp`). The latter makes everything back to normal. If `\eTeXOff` applies, the `visibleEOL` option is of course irrelevant. Note that these two macros apply to examples that follow them and not to example definitions. For instance, `\eTeXOff` and then `\NewExample{myex}{#}{}{}` will not lead CodeDoc to use an external file whenever `myex` is called, but simply as long as no `\eTeXOn` appears. To put simply, these two macros have no effect on `\NewExample`.

If `\Gobble` is positive, examples with ε-TEX and examples without behave differently. The latter gobble characters before writing to the external file. Thus, `\CodeOutput` will execute line with the first characters gobbled. With ε-TEX, however, nothing is gobbled in `\CodeOutput`. This means that first characters, if meant to be gobbled, will be executed. Most of the time, such characters are spaces, and the difference won't be noticed. If, for some reason, you use other characters instead, and if you want to call `\CodeOutput` nonetheless, then a switch to an external file may be a good idea.

2.2 `\ShortVerb` and friends

CodeDoc provides a number of facilities to act on verbatim contexts. They declare one or more character(s) to have a special effect under certain circumstances.

- `\ShortVerb{`⟨*Character*⟩`}`
- `\UndoShortVerb`

This is well-known. ⟨*Character*⟩ is turned into a shorthand for `\verb`. You can define only one such character, and that's why `\UndoShortVerb` doesn't take an argument (like all `\Undo...` below). In CodeDoc verbatim contexts, this character returns to its normal value.

```
\ShortVerb{*}
The command *\TeX* gives \TeX.

\begin{example}
And the star appeared:*
\end{example}\CodeInput
```

The command `\TeX` gives TEX.
`And the star appeared:*`

- \ShortCode[⟨*Example name*⟩]{⟨*Character*⟩}
- \UndoShortCode

This turns ⟨*Character*⟩ into an equivalent of \begin{code} and \end{code}. In normal mode, the verbatim material will be printed according to ⟨*Example name*⟩'s specifications. If this optional argument is not present, then \ShortCode will follow code's style. Most importantly, in produce mode everything between two ⟨*Characters*⟩ will be written to the file under production.

```
\RenewExample[continuous]{code}
{\itshape\ttfamily#}{}{\CodeInput}
\LineNumber{code}{\small\rmfamily}{1em}
\ShortCode+ Oh yeah, braces are never
            needed...

\begin{code}
Here it's +normal+
\end{code}

+
Here it writes code...
+
```

```
4 Here it's +normal+
5 Here it writes code...
```

\ShortVerb and \ShortCode have one caveat. If you \Undo... them and the next character (disregarding spaces, comments and ends of lines) is a short verb or a short code respectively, in produce mode it will fire as if it was still active. A pair of braces after the \Undo... statement prevents this.

- \VerbBreak{⟨*Character*⟩}
- \UndoVerbBreak

Every once in a while, breaking a verbatim line may be useful. In verbatim contexts, ⟨*Character*⟩ breaks the line, creates an unnumbered new one and indents it to the indentation of the original line. When \CodeOutput is processed, the \VerbBreak character is ignored. However, you should not break in the middle of a control sequence (admittingly a strange idea), or it won't form. It is also ignored, of course, when writing to a file in produce mode.

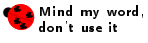
```
\VerbBreak{\=} An escape character is ok
\begin{example*}
\TeX
  \emph=\TeX
\end{example*}\CodeInput

And the result is: \CodeOutput
```

```
\TeX
␣␣\emph
  \TeX
And the result is: TEX TEX
```

- \VerbCommand{⟨*Escape*⟩}{⟨*Left brace*⟩}{⟨*Right brace*⟩}
- \UndoVerbCommand

In verbatim contexts, those three characters will serve to form control sequences. In \CodeOuput and produce mode, they are ignored. More specifically, ⟨*Escape*⟩ gobbles all letters following it (forming a putative command name) while everything vanishes that appears between ⟨*Left brace*⟩ and ⟨*Right brace*⟩. This is not a very sound device, and above all you should add a ⟨*Left brace*⟩-⟨*Right brace*⟩ pair after a command called with ⟨*Escape*⟩, if it precedes a command to be executed in \CodeOutput. That is, suppose \VerbCommand{!}{(}{)}, then !foo\foo is a very bad idea in your code, while !foo()\foo is ok. All the comments in the examples here are done with \VerbCommand.



Since numbered examples environment define the current label to be the number of the current line, an interesting application is to use `\label` to refer to it.

```
\VerbCommand!()

(Everything is normal here!)
\RenewExample[numbered]{example}
{\ttfamily#}{#}{}
\LineNumber{example}{}{1cm}

\begin{example}
\TeX !underline(\Tex!TeX)
%Second line !label(myline)
\end{example}

\CodeInput

gives \CodeOutput and the label is
 on line \ref{myline}.
```

(Everything is normal here!)

```
1 \TeX \TexTeX
2 %Second line
```

gives TEX and the label is on line 2.

- `\CodeEscape{`⟨*Character*⟩`}`
- `\UndoCodeEscape`

In normal mode, this command does absolutely nothing. However, in produce mode, ⟨*Character*⟩ becomes an escape character to form control sequences that will be expanded when writing to the file under production. It's useful mainly to put the values defined by `\ProduceFile` somewhere in your file. For instance, the following code

```
\CodeEscape!
\ProduceFile{mypack.sty}[mypack][v.2.1][2009/02/24]

\begin{code}
\ProvidesPackage{!FileName}[!FileDate!space !FileVersion!space My super package.]
\end{code}
```

will write `\ProvidesPackage{mypack}[2009/02/24 v.2.1 My super package.]` to `mypack.sty`.

2.3 Using fancyvrb

CodeDoc is minimally compatible with fancyvrb, in the sense that verbatim characters defined and undefined with `\DefineShortVerb` and `\UndefineShortVerb` are recognized in produce mode (hopefully). Besides, verbatim environments defined with `\DefineVerbatimEnvironment` are automatically added to the list of dangerous environments. The environments offered by fancyvrb and the fvrb-ex companion package already belong to that list.

You can even redefine the `code` environment with fancyvrb facilities.[5] However:

[5] It will indeed add `code` to the list of dangerous environment, which is already the case when `code` is redefined with `\RenewExample`. But CodeDoc evaluates whether an environment is `code` before checking the list of dangerous environments.

- `\ShortCode` will stick to the last style defined for `code` (if it is set to follow this environment).
- Since everything is gobbled after `\begin{code}` in produce mode, you can freely put your keyval pairs here, as usual with fancyvrb. However, *you should not input these pairs on the following line(s)*, although it's ok with fancyvrb. The following code will lead `xleftmargin=1cm]` to be written on the file under production.

```
\begin{code}[frame=single,
xleftmargin=1cm]
\def\foo{FOO}
\end{code}
```

- The `gobble` and `commandchars` parameters will be obeyed in normal mode (since fancyvrb is in charge), but *not in produce mode, unless you also specify the* `\Gobble` *and* `\VerbCommand` *parameters (see above) accordingly.*

3 Summary of commands

In this section I explain the behavior of all CodeDoc constructions in normal and produce mode respectively. Commands which have some effect in produce mode are subject to the restrictions given in section 1.4.

3.1 Class options

- `autoclose`
  Normal Mode: Does nothing.
  Produce Mode: *The current file is closed when a new one is opened with* `\ProduceFile`.
- `index`
  Normal Mode: Loads makeidx and calls `\makeindex`. `\StopHere` automatically launches `\printindex`.
  Produce Mode: *Does nothing.*
- `noheader`
  Normal Mode: Does nothing.
  Produce Mode: *No header is written to the file when it is opened.*
- `obeystop`
  Normal Mode: The document stops at `\StopHere{`⟨*Code*⟩`}` and executes ⟨*Code*⟩. If the `index` option is on, `\printindex` is executed after ⟨*Code*⟩.
  Produce Mode: *Does nothing.*
- `tracing0, tracing1, tracing2`
  Normal Mode: Does nothing.
  Produce Mode: *CodeDoc normally writes a report to the log file. If* `tracing0` *is on, there's no report; if* `tracing1` *is on (which is default), CodeDoc reports only about opening files and writing code. With* `tracing2`*, it also reports about characters defined as* `\ShortVerb` *or* `\CodeEscape`*, environments added to the list of dangerous environments, etc.*

3.2 Environments

- `code`
  Normal Mode: The content is displayed verbatim according to the style defined for `code`.
  Produce Mode: *The content is written to the file in production.*
- `example`
  Normal Mode: A minimal example environment that provides `\CodeInput` (in typewriter font) and `\CodeInput`.
  Produce Mode: *The content is skipped.*
- `invisible`
  Normal Mode: The content is skipped.
  Produce Mode: *The content is written to the file in production.*

3.3 Commands

- \AddBlankLine
  Normal Mode: Does nothing.
  Produce Mode: *Adds a blank line to the file in production.*
- \bslash
  Normal Mode: Prints \. Designed to adapt to hyperref's bookmarks.
  Produce Mode: *Does nothing.*
- \BoxTolerance{⟨*Dimension*⟩}
  Normal Mode: Excess size tolerated before a verbatim line is reported as an overfull box.
  Produce Mode: *Does nothing.*
- \CloseFile{⟨*File*⟩}
  Normal Mode: \FileName and others are not available anymore.
  Produce Mode: *Closes* ⟨File⟩. *No file is considered in production until the next* \ProduceFile, *even if there are open files. Useless in* autoclose *mode.*
- \CodeEscape{⟨*Character*⟩}
  Normal Mode: Does nothing.
  Produce Mode: ⟨Character⟩ *turns into an escape character in* code *contexts.*
- \CodeFont{⟨*Font specifications*⟩}
  Normal Mode: The style of the code environment if it has not been redefined with \RenewExample. Default is \ttfamily.
  Produce Mode: *Does nothing.*
- \CodeInput
  Normal Mode: Displays the code of the last example environment verbatim, according to the style defined for that environment.
  Produce Mode: *Does nothing.*
- \CodeOutput
  Normal Mode: Executes the code of the last example environment, according to the style defined for that environment.
  Produce Mode: *Does nothing.*
- \DangerousEnvironment{⟨*List of environments*⟩}
  Normal Mode: Does nothing.
  Produce Mode: *The environments in the list are skipped during processing.*
- \DefineEnvironment{⟨*Environment*⟩}
  Normal Mode: Prints ⟨*Environment*⟩ according to \PrintMacro and adds it to the index with '(environment)' and a line number typeset according to \DefineIndexFont.
  Produce Mode: *Gobbles the first characters of* ⟨Environment⟩, *just in case.*
- \DefineIndexFont{⟨*Font specifications*⟩}
  Normal Mode: Style of the page number in the index for \DefineMacro and \DefineEnvironment entries.
  Produce Mode: *Does nothing.*
- \DefineMacro{⟨*Macro*⟩}
  Normal Mode: Prints ⟨*Macro*⟩ according to \PrintMacro and adds it to the index with a line number typeset according to \DefineIndexFont.
  Produce Mode: *Gobbles the first characters of* ⟨Macro⟩, *just in case.*
- \DescribeEnvironment{⟨*Environment*⟩}
  Normal Mode: Prints ⟨*Environment*⟩ according to \PrintMacro and adds it to the index with '(environment)' and a line number typeset according to \DescribeIndexFont.
  Produce Mode: *Gobbles the first characters of* ⟨Environment⟩, *just in case.*
- \DescribeIndexFont{⟨*Font specifications*⟩}
  Normal Mode: Style of the page number in the index for \DescribeMacro and \DescribeEnvironment entries.
  Produce Mode: *Does nothing.*
- \DescribeMacro{⟨*Macro*⟩}
  Normal Mode: Prints ⟨*Macro*⟩ according to \PrintMacro and adds it to the index with a line number typeset according to \DescribeIndexFont.
  Produce Mode: *Gobbles the first characters of* ⟨Macro⟩, *just in case.*
- \DocStripMarginpar
  Normal Mode: Sets the adequate values for the proper printing of macros with \DescribeMacro and \DefineMacro (and variants for environments), so that they appear \marginpar'ed as with DocStrip. More precisely, it executes \reversmarginpar, and sets \marginparpush to 0pt and \marginparwidth to 8pc.
  Produce Mode: *Does nothing.*

- \eTeXOff
  Normal Mode: All subsequent example environments are processed with an external file, whose extension is .exp.
  Produce Mode: *Does nothing.*
- \eTeXOn
  Normal Mode: All subsequent example environments are processed without an external file. This is default. (Requires ε-TEX, of course.)
  Produce Mode: *Does nothing.*
- \Gobble{⟨*Number*⟩}
  Normal Mode: The number of characters that will be gobbled at the beginning of each example and code environments. In case of a blank line, nothing is gobbled, but a blank line is added. Tab characters count as one character.
  Produce Mode: *Same as in normal mode, but when writing to the file in production.*
- \Header{⟨*Text*⟩}
  Normal Mode: Does nothing.
  Produce Mode: *Text to be written at the beginning of a file when it is opened with* \ProduceFile*. Comment characters will be automatically added at the beginning of each line. Ends of lines are obeyed. If the* noheader *option is on, nothing is written.*
- \IgnorePrefix{⟨*Macro prefix*⟩}
  Normal Mode: Ignores ⟨*Macro prefix*⟩ when sorting index entries generated by \DescribeMacro and \DefineMacro. ⟨*Macro prefix*⟩ will be typeset according to \PrintPrefix in the index.
  Produce Mode: *Does nothing.*
- \LineNumber{⟨*Name*⟩}{⟨*Font specifications*⟩}{⟨*Width*⟩}[⟨*Number*⟩]
  Normal Mode: The line number of ⟨*Name*⟩ will be typeset according to ⟨*Font specifications*⟩ in a box that will spread from the left margin into the main text width by a length of ⟨*Width*⟩ (0pt by default). The next ⟨*Name*⟩ will start at ⟨*Number*⟩ if specified.
  Produce Mode: *Does nothing.*

- \marg{⟨*Argument*⟩}
  Normal Mode: \marg{Argument} prints {⟨*Argument*⟩} (mandatory argument).
  Produce Mode: *Does nothing.*
- \meta{⟨*Argument*⟩}
  Normal Mode: \meta{Argument} prints ⟨*Argument*⟩.
  Produce Mode: *Does nothing.*
- \NewExample[⟨*Options*⟩]{⟨*Name*⟩}{⟨*Code input*⟩}{⟨*Code output*⟩}{⟨*Immediate execution*⟩}
  Normal Mode: Creates ⟨*Name*⟩ as an example environment to provide \CodeInput as ⟨*Code input*⟩ (where the code to be typeset is represented by #) and \CodeOutput as ⟨*Code output*⟩ (where the code to be executed is represented by #). When encountered, ⟨*Name*⟩ executes ⟨*Immediate execution*⟩. ⟨*Code input*⟩, ⟨*Code output*⟩ and ⟨*Immediate execution*⟩ can be empty.
  Options are:
  numbered: Each line of ⟨*Name*⟩ is numbered.
  continuous: Each line of ⟨*Name*⟩ is numbered and numbering continues from one ⟨*Name*⟩ to the other.
  visibleEOL: If ⟨*Name*⟩ is processed with ε-TEX, This prevents ends of lines and commented parts of lines from being removed before anything is executed in \CodeInput. See page 12 for a discussion.
  Produce Mode: *Adds* ⟨Name⟩ *to the list of dangerous environments and gobbles the remaining arguments.*
- \oarg{⟨*Argument*⟩}
  Normal Mode: \oarg{Argument} prints [⟨*Argument*⟩] (optional argument).
  Produce Mode: *Does nothing.*
- \parg{⟨*Argument*⟩}
  Normal Mode: \parg{Argument} prints (⟨*Argument*⟩) (picture argument).
  Produce Mode: *Does nothing.*
- \PrintMacro{⟨*Macro or environment*⟩}
  Normal Mode: Typesets the argument to \DescribeMacro, \DefineMacro, \DescribeEnvironment and \DefineEnvironment. Should be freely redefined by users. By default, it prints its argument as with DocStrip, provided \DocStripMarginpar has been executed beforehand.
  Produce Mode: *Does nothing.*
- \PrintPrefix{⟨*Macro prefix*⟩}
  Normal Mode: Typesets ⟨*Macro prefix*⟩, as defined by \IgnorePrefix, in the index. Should be redefined by the user. By default, it does nothing.
  Produce Mode: *Does nothing.*
- \ProduceFile{⟨*File*⟩}[⟨*File name*⟩][⟨*File version*⟩][⟨*File date*⟩]
  Normal Mode: Provides ⟨*File*⟩ as \FileSource, ⟨*File name*⟩ as \FileName, ⟨*File version*⟩ as \FileVersion and ⟨*File date*⟩ as \FileDate.

Produce Mode: *Opens* ⟨File⟩ *and writes the header (unless* `noheader` *is on), unless* ⟨File⟩ *is already open and* `autoclose` *is not specified, in which case* CodeDoc *will simply puts* ⟨File⟩ *back in production. Subsequent code will be written to this file. Closes the current file if* `autoclose` *is on. Provides* ⟨File name⟩ *as* `\FileName`, ⟨File version⟩ *as* `\FileVersion` *and* ⟨File date⟩ *as* `\FileDate`, *to be used with* `\CodeEscape`.

- `\RenewExample[`⟨*Options*⟩`]{`⟨*Name*⟩`}{`⟨*Code input*⟩`}{`⟨*Code output*⟩`}{`⟨*Immediate execution*⟩`}`
  Normal Mode: Same as `\NewExample` to redefine ⟨*Name*⟩.
  Produce Mode: *Adds* ⟨Name⟩ *to the list of dangerous environments and gobbles the remaining arguments.*
- `\ShortCode{`⟨*Character*⟩`}`
  Normal Mode: Turns ⟨*Character*⟩ into a shorthand for `\begin{document}` and `\end{document}`.
  Produce Mode: *Like in normal mode: everything between two* ⟨Characters⟩ *will be written to the file in production.*
- `\ShortVerb{`⟨*Character*⟩`}`
  Normal Mode: Turns ⟨*Character*⟩ into a shorthand for `\verb`.
  Produce Mode: *Subsequently gobbles everything between two* ⟨Characters⟩.
- `\StartIgnore`
  Normal Mode: Does nothing.
  Produce Mode: *Stops executing anything until* `\StopIgnore`.
- `\StopHere{`⟨*Code*⟩`}`
  Normal Mode: If the `obeystop` option is on, executes ⟨*Code*⟩ followed by `\printindex` if `index` is on, and ends the document.
  Produce Mode: *Does nothing.*
- `\StopIgnore`
  Normal Mode: Does nothing.
  Produce Mode: *Marks the end of* `\StartIgnore`.
- `\TabSize{`⟨*Number*⟩`}`
  Normal Mode: Sets the number of spaces to represent a tab character in verbatim contexts.
  Produce Mode: *Does nothing.*
- `\UndoCodeEscape`
  Normal Mode: Does nothing.
  Produce Mode: *Sets the* `\CodeEscape` *character to a normal character.*
- `\UndoShortCode`
  Normal Mode: Sets the `\ShortCode` character to a normal character.
  Produce Mode: *Sets the* `\ShortCode` *character to a normal character.*
- `\UndoShortVerb`
  Normal Mode: Sets the `\ShortVerb` character to a normal character.
  Produce Mode: *Sets the* `\ShortVerb` *character to a normal character.*
- `\UndoVerbBreak`
  Normal Mode: Sets the `\VerbBreak` character to a normal character.
  Produce Mode: *Sets the* `\VerbBreak` *character to a normal character.*
- `\UndoVerbCommand`
  Normal Mode: Sets the `\VerbCommand` characters to normal characters.
  Produce Mode: *Sets the* `\VerbCommand` *character to normal characters.*
- `\VerbBreak{`⟨*Character*⟩`}`
  Normal Mode: Turns ⟨*Character*⟩ into a line breaker in verbatim contexts; more precisely, the line will break where ⟨*Character*⟩ appears and will be indented with the same amount of space as the original one. ⟨*Character*⟩ is ignored in `\CodeOutput`.
  Produce Mode: *Ignores* ⟨Character⟩ *when writing to the file in production.*
- `\VerbCommand{`⟨*Escape*⟩`}{`⟨*Left brace*⟩`}{`⟨*Right brace*⟩`}`
  Normal Mode: Turns ⟨*Escape*⟩ into an escape character in verbatim contexts, and ⟨*Left brace*⟩ and ⟨*Right brace*⟩ into characters of category 1 and 2 respectively. In `\CodeOutput`, ⟨*Escape*⟩ gobbles all subsequent letters and everything between ⟨*Left brace*⟩ and ⟨*Right brace*⟩ is gobbled too.
  Produce Mode: *Does the same as normal mode for* `\CodeOutput`. *Letters following* ⟨Escape⟩ *are gobbled, as is everything between* ⟨Left brace⟩ *and* ⟨Right brace⟩.

Part II
Implementation

The usual things (; is my \CodeEscape character). Turning ^^? into an active character is less usual but useful to delimit ends of code material.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{;FileName}[;FileDate ;FileVersion Code and documentation in one file.]
\makeatletter
\catcode`\^^?=13
```

4 Options and basic definitions

\cd@GetClass

Options are mostly conditional switching. \cd@tracingmode will be used in an \ifcase statement. \cd@GetClass will be analyzed to retrieve the class and its options.

```
\newif\ifcd@produce
\newif\ifcd@autoclose
\newif\ifcd@obeystop
\newif\ifcd@makeindex
\newif\ifcd@noheader
\newcount\cd@tracingmode
\cd@tracingmode1
\def\cd@GetClass{article()}

\DeclareOption{autoclose}{\cd@autoclosetrue}
\DeclareOption{produce}{\cd@producetrue}
\DeclareOption{index}{\cd@makeindextrue}
\DeclareOption{obeystop}{\cd@obeystoptrue}
\DeclareOption{noheader}{\cd@noheadertrue}
\DeclareOption{tracing0}{\cd@tracingmode0}
\DeclareOption{tracing1}{\cd@tracingmode1}
\DeclareOption{tracing2}{\cd@tracingmode2}
\DeclareOption*{\edef\cd@GetClass{\CurrentOption()}}
\ProcessOptions\relax
```

\cd@end
\cd@LoadClass
\cd@GetOptions

We define \cd@LoadClass as a recursive retrieval of options, then passed to the class with \PassOptionsToClass, which we load. This is done only if we're not in produce mode, in which case no class is loaded.

```
\def\cd@end{cd@end}
\ifcd@produce
\else
\def\cd@LoadClass#1(#2){%
  \def\cd@Class{#1}
  \expandafter\cd@GetOptions#2;cd@end;%
  \LoadClass{#1}%
  \@ifnextchar({\expandafter\@gobble\@gobble}{}}
\def\cd@GetOptions#1;{%
  \def\cd@TempArg{#1}
  \ifx\cd@TempArg\cd@end%
    \let\cd@next\relax
  \else%
    \PassOptionsToClass{#1}{\cd@Class}%
    \let\cd@next\cd@GetOptions%
  \fi\cd@next}
\expandafter\cd@LoadClass\cd@GetClass
```

\StopHere Still in normal mode, we load makeidx if required and define \StopHere accordingly.

```
\ifcd@makeindex
  \RequirePackage{makeidx}
  \makeindex
\else
  \let\printindex\relax
\fi
\ifcd@obeystop
  \ifcd@makeindex
    \long\def\StopHere#1{#1\relax\par\printindex\end{document}}
  \else
    \long\def\StopHere#1{#1\relax\par\end{document}}
  \fi
\else
  \long\def\StopHere#1{}
\fi
\fi
```

5 Normal mode

Although the following code is used in normal mode only, I did not feel like embedding hundreds of lines under a \ifcd@produce conditional. Pure superstition, perhaps.

Here's the switch for ε-TEX and some shorthands.

```
\newif\ifcd@eTeX
\@ifundefined{eTeXversion}{\cd@eTeXfalse}{\cd@eTeXtrue}

\def\cd@Warning{\ClassWarningNoLine{codedoc}}
\def\cd@Error#1{\ClassError{codedoc}{#1}{}}
```

5.1 Describing macros

\DocStripMarginpar \PrintMacro Most of the following macros are imitated from DocStrip, in a simpler but less careful manner. The first two are straightforward.

```
\def\DocStripMarginpar{\reversemarginpar\marginparpush0pt\relax\marginparwidth8pc\relax}
\def\PrintMacro#1{\noindent\marginpar{\raggedleft\strut\ttfamily#1}\ignorespaces}
```

\DescribeIndexFont \DescribeMacro \cd@DescribeMacro \DescribeEnvironment \cd@DescribeEnvironment \DefineIndexFont \DefineMacro \cd@DefineMacro \DefineEnvironment \cd@DefineEnvironment \DescribeMacro and its companions first turn @ into a letter, so that a control sequence containing it is recognized as such, sets \cd@Index, used in the \ifcase statement below (a simple conditional could do the job, since there are only two values, but there might be more someday if one wants to distinguish other index entries, like 'used' macros), and pass their arguments to \PrintMacro with the first token \string'ed (even in the case of an environment, because someone might describe its environment with a \begin{myenv} command). In case of a macro, the argument is also passed to \cd@MakeEntry to index it.

The hyperref package does not work properly with indexes if a style is specified with | in the entry. Since we use such styles, and since we want to use hyperref, we circumvent the problem with \hyperpage added to the style. By default, it does nothing, but if the user loads hyperref, it will have the adequate meaning.

```
\newcount\cd@Index
\def\hyperpage#1{#1}

\def\DescribeIndexFont#1{\gdef\cdatDescribeFont##1{{#1\hyperpage{##1}}}}
\DescribeIndexFont{}
\def\DescribeMacro{\makeatletter\cd@DescribeMacro}
\def\cd@DescribeMacro#1{%
  \makeatother%
  \cd@Index=0 %
  \cd@MakeEntry#1\cd@EndOfEntry%
  \PrintMacro{\string#1}}
\def\DescribeEnvironment{\makeatletter\cd@DescribeEnvironment}
\def\cd@DescribeEnvironment#1{%
```

```
  \makeatother%
  \index{#1@\texttt{#1} (environment)|cdatDescribeFont}%
  \PrintMacro{\string#1}}

\def\DefineIndexFont#1{\gdef\cdatDefineFont##1{{#1\hyperpage{##1}}}}
\DefineIndexFont{\itshape}
\def\DefineMacro{\makeatletter\cd@DefineMacro}
\def\cd@DefineMacro#1{%
  \makeatother%
  \cd@Index1 %
  \cd@MakeEntry#1\cd@EndOfEntry%
  \PrintMacro{\string#1}}
\def\DefineEnvironment{\makeatletter\cd@DefineEnvironment}
\def\DefineEnvironment#1{%
  \makeatother%
  \index{#1@\texttt{#1} (environment)|cdatDefineFont}%
  \PrintMacro{\string#1}}
```

`\cd@MakeEntry`

This takes two arguments but considers only the first one, so that `\DescribeMacro{\foo\marg{Argument}}` will ignore `\marg{Argument}`. We pass that argument to `\cd@AnalyzeEntry` with the escape character removed (for a proper indexing), call `\cd@AnalyzePrefix` on the result and finally `\cd@@MakeEntry`

```
\def\cd@MakeEntry#1#2\cd@EndOfEntry{%
  \def\cd@TempEntry{}%
  \begingroup\escapechar\m@ne\expandafter\cd@AnalyzeEntry\string#1\cd@end\endgroup%
  \expandafter\cd@AnalyzePrefix\cd@TempEntry\cd@end%
  \expandafter\cd@@MakeEntry\cd@TempEntry\cd@EndOfEntry}
```

`\cd@AnalyzeEntry` `\AtChar`

The aim of this macro is to process `@`. Indeed, `@` is MakeIndex's operator to signal that an entry should be indexed under another name (as done here). But `@` is also a very popular letter in TEX's world when it comes to macros. DocStrip's solution is to create a special style file for MakeIndex, so that the function of `@` is taken over by another character. But then, when a user compiles a DocStrip document, this style file must be indicated to MakeIndex, which many people might not do. So I prefer to leave MakeIndex alone and process the entry beforehand, replacing `@` by a character denotation. That's the job of `\cd@AnalyzeEntry`, which scans the macro name token by token and replace `@` by `\AtChar`.

```
\chardef\AtChar=`\@
\def\cd@AnalyzeEntry#1{%
  \let\cd@next\cd@AnalyzeEntry%
  \ifx#1\cd@end%
    \let\cd@next\relax%
  \else\if#1@%
    \expandafter\gdef\expandafter\cd@TempEntry\expandafter{\cd@TempEntry\AtChar}%
  \else%
    \expandafter\gdef\expandafter\cd@TempEntry\expandafter{\cd@TempEntry#1}%
  \fi\fi\cd@next}
```

`\IgnorePrefix`

Here comes the mechanism to remove prefixes when sorting entries. `\IgnorePrefix` simply resets some values and call `\cd@IgnorePrefix` on its argument along with a terminator.

```
\newcount\cd@PrefixCount
\def\IgnorePrefix#1{\cd@PrefixCount\z@\def\Prefix{}\cd@IgnorePrefix#1\cd@end}
```

`\cd@IgnorePrefix` `\cd@MakePrefix`

This analyzes the prefix just like `\cd@AnalyzeEntry` above and replaces all occurrences of `@` by `\AtChar`. Since the name of the macro is `\string`'ed when subjected to `\DefineMacro` and others, we also `\string` all letters of the prefix, which have then category code 12.

```
\def\cd@IgnorePrefix#1{%
  \let\cd@next\cd@IgnorePrefix%
  \ifx#1\cd@end%
    \def\cd@next{\expandafter\cd@ScanPrefix\Prefix\cd@end}%
  \else\if#1@%
```

```
    \expandafter\def\expandafter\Prefix\expandafter{\Prefix\AtChar}%
  \else%
    \edef\cd@PrefixLetter{\string#1}%
    \expandafter\cd@MakePrefix\cd@PrefixLetter%
  \fi\fi\cd@next}
\def\cd@MakePrefix#1{%
  \expandafter\def\expandafter\Prefix\expandafter{\Prefix#1}}%
```

`\cd@ScanPrefix` `\cd@DefPrefix` `\cd@AnalyzePrefix`

Then we just scan the prefix to compute the number of characters it is made of. `\cd@AnalyzePrefix` is defined accordingly to take the right number of characters out of a macro name (fed in `\cd@MakeEntry` above) and lump them into `\cd@TempPrefix`, and define the rest of the entry as the remaining characters up to the terminator.

```
\def\cd@ScanPrefix#1{%
  \ifx#1\cd@end%
    \let\cd@next\cd@DefPrefix%
  \else%
    \advance\cd@PrefixCount\@ne%
    \let\cd@next\cd@ScanPrefix%
  \fi\cd@next}
\def\cd@DefPrefix{%
  \ifcase\cd@PrefixCount%
    \def\cd@AnalyzePrefix##1\cd@end{}%
  \or\def\cd@AnalyzePrefix##1##2\cd@end{%
    \def\cd@TempPrefix{##1}\def\cd@RestOfEntry{##2}\cd@ComparePrefix}%
  \or\def\cd@AnalyzePrefix##1##2##3\cd@end{%
    \def\cd@TempPrefix{##1##2}\def\cd@RestOfEntry{##3}\cd@ComparePrefix}%
  \or\def\cd@AnalyzePrefix##1##2##3##4\cd@end{%
    \def\cd@TempPrefix{##1##2##3}\def\cd@RestOfEntry{##4}\cd@ComparePrefix}%
  \or\def\cd@AnalyzePrefix##1##2##3##4##5\cd@end{%
    \def\cd@TempPrefix{##1##2##3##4}\def\cd@RestOfEntry{##5}\cd@ComparePrefix}%
  \or\def\cd@AnalyzePrefix##1##2##3##4##5##6\cd@end{%
    \def\cd@TempPrefix{##1##2##3##4##5}\def\cd@RestOfEntry{##6}\cd@ComparePrefix}%
  \or\def\cd@AnalyzePrefix##1##2##3##4##5##6##7\cd@end{%
    \def\cd@TempPrefix{##1##2##3##4##5##6}\def\cd@RestOfEntry{##7}\cd@ComparePrefix}%
  \or\def\cd@AnalyzePrefix##1##2##3##4##5##6##7##8\cd@end{%
    \def\cd@TempPrefix{##1##2##3##4##5##6##7}\def\cd@RestOfEntry{##8}\cd@ComparePrefix}%
  \or\def\cd@AnalyzePrefix##1##2##3##4##5##6##7##8##9\cd@end{%
    \def\cd@TempPrefix{##1##2##3##4##5##6##7##8}\def\cd@RestOfEntry{##9}\cd@ComparePrefix}%
  \fi\ignorespaces}
```

`\cd@ComparePrefix`

Comparing prefixes is simply a matter of string testing. In case they match, the entry is redefined as the `\cd@RestOfEntry`, so that macros will be indexed with the prefix removed.

```
\newif\ifcd@Prefix
\def\cd@ComparePrefix{%
  \ifx\cd@TempPrefix\Prefix%
    \expandafter\def\expandafter\cd@TempEntry\expandafter{\cd@RestOfEntry}%
    \cd@Prefixtrue%
  \else%
    \cd@Prefixfalse%
  \fi}
```

`\cd@@MakeEntry` `\PrintPrefix`

Finally, `\cd@@MakeEntry` indexes the macro under its name with a prefixed escapechar (since it was removed above) and `\Prefix` in case it was found to match. We also set some default values.

```
\def\cd@@MakeEntry#1\cd@EndOfEntry{%
  \ifcd@Prefix%
    \ifcase\cd@Index%
      \index{#1@\texttt{\char\escapechar\PrintPrefix\Prefix#1}|cdatDescribeFont}%
    \or%
      \index{#1@\texttt{\char\escapechar\PrintPrefix\Prefix#1}|cdatDefineFont}%
    \fi%
```

```
  \else%
  \ifcase\cd@Index%
      \index{#1@\texttt{\char\escapechar#1}|cdatDescribeFont}%
    \or%
      \index{#1@\texttt{\char\escapechar#1}|cdatDefineFont}%
    \fi%
  \fi}

\IgnorePrefix{}%
\let\PrintPrefix\relax
```

\meta \marg \oarg \parg

These again are imitated from the DocStrip bundle, with less care.

```
\def\meta#1{{\ensuremath\langle\emph{#1}\ensuremath\rangle}}
\def\marg#1{\texttt{\{}\meta{#1}\texttt{\}}}
\def\oarg#1{\texttt{[}\meta{#1}\texttt{]}}
\def\parg#1{\texttt{(}\meta{#1}\texttt{)}}
```

\cd@bslash \bslash

We define our backslash to adapt to hyperref. To this end, we use \texorpdfstring, an hyperref command that expands to its first argument in normal contexts and to its second one in bookmarks.

The only problem is that hyperref defines \textorpdfstring with \newcommand instead of \def. So we obviously can't define it here, and we wait for the beginning of the document.

```
\def\cd@bslash{\char`\\}
\def\bslash{\texorpdfstring{\cd@bslash}{\string\\}}
\AtBeginDocument{\@ifundefined{texorpdfstring}{\def\texorpdfstring#1#2{#1}}{}}
```

5.2 \ShortVerb and associates

\cd@CharErr \cd@BadChar

Before entering the intricate realm of verbatim text, here are some simpler definitions. First, we delimit what characters we consider to be acceptable in \ShortVerb and other. The choice might seem rather conservative, but things are less dangerous this way.

```
\def\cd@CharErr#1#2{%
  \bgroup
    \escapechar\m@ne
    \cd@Error{You can't use \string#1 for \string\\#2}
  \egroup}

\newif\ifcd@BadChar

\def\cd@BadChar#1#2{%
  \cd@BadChartrue
  \ifcase\catcode`#1 % \
    \cd@CharErr{\\}{#2}%
  \or% {
    \cd@CharErr{\{}{#2}%
  \or% }
    \cd@CharErr{\}}{#2}%
  \or% $
    \cd@BadCharfalse%
  \or% &
    \cd@BadCharfalse%
  \or% ^^M
  \or% #
    \cd@BadCharfalse%
  \or% ^
    \cd@BadCharfalse%
  \or% _
    \cd@BadCharfalse%
  \or% Ignored
  \or% Spaces
    \cd@CharErr{spaces}{#2}%
```

```
  \or% Letters
    \cd@CharErr{letters}{#2. \MessageBreak That's really bad}%
  \or% Other
    \cd@BadCharfalse%
  \or% Active
    \cd@CharErr{#1}{#2 - it's already active}%
  \or% %
    \cd@CharErr{#1}{#2}%
  \fi}
```

`\cd@UndoErr` `\cd@DefErr`

We also define two templates for error messages in case the user wants to `\Undo...` something that was never done or define a new character while one is already in use.

```
\def\cd@UndoErr#1{%
  \bgroup%
    \escapechar\m@ne%
    \cd@Error{%
      There is no \string\\\string#1\space defined.\MessageBreak%
      \string\\Undo\string#1\space on line \the\inputlineno\space is useless}%
  \egroup}
\def\cd@DefErr#1#2{%
  \bgroup%
  \escapechar\m@ne%
    \expandafter\xdef\csname cd@#2Error\endcsname{%
      \noexpand\cd@Error{%
        You've already defined \string#1 as a \string\\#2\noexpand\MessageBreak%
        on l. \the\inputlineno. You can't have two.\noexpand\MessageBreak%
        Say \string\\Undo#2\space and then \string\\#2\space to change}}%
  \egroup}
```

`\ShortVerb`

Before defining any character, we run some tests: is it a bad character, and is there another character already in use? In the latter case, `\ifcd@ShortVerb` should be switched to `true`.

```
\newif\ifcd@ShortVerb

\def\ShortVerb#1{%
  \cd@BadChar{#1}{ShortVerb}%
  \ifcd@BadChar%
  \else\ifcd@ShortVerb
    \cd@ShortVerbError
```

If none of the above applies, we switch the conditional to true define `\cd@ShortVerbError` with `\cd@DefErr`. We also store the character's original catcode to restore if undone.

```
  \else
    \cd@ShortVerbtrue
    \cd@DefErr{#1}{ShortVerb}
    \chardef\cd@ShortVerbCat\catcode`#1%
```

Then we use the ~ with lowercase trick to define the character.

```
    \bgroup%
      \lccode`\~=`#1%
      \lowercase{%
```

A `\ShortVerb` character makes the adequate modifications to display text verbatim. `\cd@Verbatim` is CodeDoc's container of all such modifications (mostly catcode changing). `\catcode`#1=13` is necessary because the character might be one of the specials whose catcode is changed in `\cd@Verbatim`, e.g. `&`. We also launch `\cd@ShortVerb` which works like `\verb`.

`\leavevmode` is needed in case the `\ShortVerb` character starts a paragraph, as in the one you're reading.

```
        \gdef~{\leavevmode\bgroup\ttfamily\cd@Verbatim\catcode`#1\active\cd@ShortVerb}%
        \gdef\cd@ShortVerb##1~{##1\egroup}%
```

Finally we (re)define \UndoShortVerb to restore the original catcode and switch the appropriate conditional. Last but not least, we make the character active.

```
253         \gdef\UndoShortVerb{%
254           \ifcd@ShortVerb%
255             \cd@ShortVerbfalse%
256             \catcode`~\cd@ShortVerbCat%
257           \else%
258             \cd@UndoErr{\ShortVerb}%
259           \fi}}%
260     \egroup%
261     \catcode`#1=13
262   \fi\fi}%
```

\UndoShortVerb

This is the default definition for this command, when no \ShortVerb has been defined.

```
263 \def\UndoShortVerb{\cd@UndoErr{\ShortVerb}}
```

\ShortCode

\ShortCode works with the same pattern as \ShortVerb with important variations. First, we check whether there's an optional argument.

```
264 \newif\ifcd@ShortCode
265 \newif\ifcd@ShortCodeChar
266
267 \def\ShortCode{%
268   \@ifnextchar[
269     {\cd@MakeShortCode}
270     {\cd@MakeShortCode[code]}}
```

\cd@MakeShortCode

Then we define the real macro. We store the name of the environment and run the same tests as above.

```
271 \bgroup
272 \catcode`\^^M13%
273 \gdef\cd@MakeShortCode[#1]#2{%
274   \def\cd@TempEnv{#1}%
275   \cd@BadChar{#2}{ShortCode}%
276   \ifcd@BadChar%
277   \else\ifcd@ShortCodeChar%
278     \cd@ShortCodeError%
```

Then we check whether the environment exists, thanks to \⟨*Environment*⟩@cd@EOL which is defined for ⟨*Environment*⟩ when created with \NewExample.

```
279   \else%
280     \expandafter\ifx\csname #1@cd@EOL\endcsname\relax%
281       \cd@Error{%
282         `#1' is not an example environment.\MessageBreak%
283         `code' is selected instead}%
284       \def\cd@TempEnv{code}%
285     \fi%
```

This is the same as above: we state that a character has been defined as a \ShortCode.

```
286     \cd@ShortCodeChartrue%
287     \cd@DefErr{#2}{ShortCode}%
288     \chardef\cd@ShortCodeCat=\catcode`#2%
```

\cd@ShortCode
\cd@ShortEnd
\cd@ActivateShortCode

Then we define the character to launch the appropriate environment, but with \ifcd@ShortCode turned to true. What will happen depends on the status of the environment. If it is the default code environement, it will call \cd@ShortCode as defined here, which is equivalent to \code itself (see below). On the other hand, if the environment is an example environement, the special example macro will be called and delimit its argument with \cd@ShortEnd, which is the \ShortCode character itself. \cd@ActivateShortCode is needed to reactivate the character in case it was one of the specials, as we did for \ShortVerb.

```
289     \bgroup%
```

```
      \lccode`\~=`#2%
      \lowercase{%
      \gdef~{\cd@ShortCodetrue\csname\cd@TempEnv\endcsname}%
      \gdef\cd@ShortEnd{~}%
      \gdef\cd@ShortCode##1^^M##2~{\cd@StartGobble##2^^?\egroup}%
      \gdef\cd@ActivateShortCode{\catcode`#2=13\relax}%
```

`\UndoShortCode` The rest is equivalent to `\ShortVerb` above.

```
      \gdef\UndoShortCode{%
        \ifcd@ShortCodeChar%
          \catcode`~=\cd@ShortCodeCat\relax%
          \let\cd@ActivateShortCode\relax%
          \cd@ShortCodeCharfalse%
        \else%
          \cd@UndoErr{\ShortCode}%
        \fi}}%
    \egroup%
    \catcode`#2=13 %
  \fi\fi}%
\egroup
\def\UndoShortCode{\cd@UndoErr{\ShortCode}}
```

`\VerbBreak` `\VerbBreak` starts as above.

```
\newif\ifcd@VerbBreak
\newtoks\cd@@Everypar

\def\VerbBreak#1{%
  \cd@BadChar{#1}{VerbBreak}%
  \ifcd@BadChar%
  \else\ifcd@VerbBreak%
    \cd@VerbBreakError%
  \else\cd@VerbBreaktrue
    \cd@DefErr{#1}{VerbBreak}%
    \bgroup%
      \lccode`\~`#1 %
      \lowercase{%
```

`\cd@ActivateVerbBreak` However, `\VerbBreak` characters become active only in verbatim contexts. We create `\cd@ActivateVerbBreak` to that end. When active the character stores the current value of `\everypar` and then empties it (because the broken line should start with nothing).

```
      \gdef\cd@ActivateVerbBreak{%
        \catcode`#1\active%
        \gdef~{%
          \cd@@Everypar\everypar%
          \everypar{}%
```

Then we set a scratch dimension to `\cd@FirstSpaces` times the width of a space in the current font. `\cd@FirstSpaces` is incremented by spaces and tabs at the beginning of each lines. In case the current environment is numbered, we increase our scratch dimension by the width of the box containing the number, stored in `\`⟨*Environment*⟩`@cd@boxwidth`.

```
          \dimen0=\cd@FirstSpaces\fontdimen2\font\relax%
          \expandafter\ifx\csname\cd@ExampleName @cd@boxwidth\endcsname\relax%
          \else%
            \advance\dimen0 \csname\cd@ExampleName @cd@boxwidth\endcsname\relax%
          \fi%
```

Finally, we create a paragraph, turn to horizontal mode, restore `\everypar` in its initial value and create a space of the desired width, namely the same as the space at the beginning of the original broken line.

```
          \endgraf\leavevmode\everypar\cd@@Everypar\hbox to\dimen0{\hss}}}}%
    \egroup%
```

`\cd@IgnoreVerbBreak`

The character should be ignored in `\CodeOutput`, and this is what we do here. The `\Undo...` variant simply sets these commands to `\relax`.

```
    \def\cd@IgnoreVerbBreak{\catcode`#1=9\relax}%
  \fi\fi}
\def\UndoVerbBreak{%
  \ifcd@VerbBreak%
    \let\cd@ActivateVerbBreak\relax
    \let\cd@IgnoreVerbBreak\relax
    \cd@VerbBreakfalse
  \else
    \cd@UndoErr{\VerbBreak}
  \fi}
\let\cd@ActivateVerbBreak\relax
```

`\VerbCommand`
`\cd@ActivateVerbCommand`
`\cd@IgnoreVerbCommand`
`\UndoVerbCommand`

`\VerbCommand` is similar once again. We define `\cd@ActivateVerbCommand` to change the catcodes of the characters to 0, 1 and 2 in verbatim contexts and `\cd@IgnoreVerbCommand` to turn the second character into a command that gobbles its argument, delimited by the third character. This is straightforward, but the first character is more complicated: it has to gobble letters and only letters.

```
\newif\ifcd@VerbCommand

\def\VerbCommand#1#2#3{%
  \cd@BadChar{#1}{VerbCommand}%
  \cd@BadChar{#2}{VerbCommand}%
  \cd@BadChar{#3}{VerbCommand}%
  \ifcd@BadChar%
  \else\ifcd@VerbCommand%
    \cd@VerbCommandError
  \else%
    \cd@DefErr{#1, \string#2 and \string#3}{VerbCommand}
    \cd@VerbCommandtrue%
    \def\cd@ActivateVerbCommand{\catcode`#1=0 \catcode`#2=1 \catcode`#3=2\relax}%
    \def\cd@IgnoreVerbCommand{%
      \catcode`#1=13 %
      \lccode`\~=`#1 %
      \lowercase{\def~{\cd@GobbleLetters}}%
      \catcode`#2=13 %
      \lccode`\~=`#2 %
      \lowercase{\def~####1#3{}}}%
  \fi\fi}
\def\UndoVerbCommand{%
  \ifcd@VerbCommand%
    \let\cd@ActivateVerbCommand\relax%
    \let\cd@IgnoreVerbCommand\relax%
    \cd@VerbCommandfalse%
  \else%
    \cd@UndoErr{\VerbCommand}%
  \fi}%
\let\cd@IgnoreVerbCommand\relax
\let\cd@ActivateVerbCommand\relax
```

\cd@GobbleLetters

Gobbling letters is not a very delicate process. We take the next token, check whether it is of category 11, and eat it away if it is the case. That's the reason why \VerbCommand is not very sound. If the next token happens to be a macro (as might be the case since in \CodeOutput, since the escape character is turned back to 0), trying to evaluate its catcode is not a good idea.

```
\def\cd@GobbleLetters#1{\ifnum\catcode`#1=11 \expandafter\cd@GobbleLetters\else\expandafter#1\fi}
```

\CodeEscape
\UndoCodeEscape

Finally, \CodeEscape doesn't do much in normal mode. We simply check characters.

```
\newif\ifcd@CodeEscape%

\def\CodeEscape#1{%
  \cd@BadChar{#1}{CodeEscape}%
  \ifcd@BadChar%
  \else\ifcd@CodeEscape%
    \cd@CodeEscapeError%
  \else%
    \cd@CodeEscapetrue%
    \cd@DefErr{#1}{CodeEscape}%
  \fi\fi}
\def\UndoCodeEscape{%
  \ifcd@CodeEscape%
    \cd@CodeEscapefalse%
  \else%
    \cd@UndoErr{\CodeEscape}%
  \fi}%
```

5.3 Verbatim definitions

\cd@SpaceChar

Here comes the time to do some verbatim. We start with space. \ifcd@Star is the conditional switched to true if we're in a starred verbatim environment. We define the visible space character to be space of category 12 in typewriter font, as usual.


```
\newif\ifcd@Star
\newif\ifcd@NewLine
\newcount\cd@FirstSpaces

\bgroup
\catcode`\ 12%
\gdef\cd@SpaceChar{\texttt{ }}%
```

\cd@MakeSpace
\cd@ObeySpaces

Since we want spaces at the beginning of a line to count how many they are, so that \VerbBreak can properly break the line, we don't equate the space character with \@xobeysp (LaTeX's verbatim space) or \cd@SpaceChar directly; instead, \cd@ObeySpaces will print the space, being called by real spaces in \cd@VerbTab and \cd@VerbSpace. (^^I denotes a tab character).

```
\catcode`\^^I=13\relax%
\catcode`\ =13\relax%
\gdef\cd@MakeSpace{%
\ifcd@Star%
\let\cd@ObeySpaces\cd@SpaceChar%
\else%
\let\cd@ObeySpaces\@xobeysp%
\fi%
\catcode`\ =13\relax%
\catcode`\^^I=13\relax%
\let =\cd@VerbSpace%
\let^^I=\cd@VerbTab}%
```

\cd@VerbSpace
\cd@VerbTab

In verbatim contexts, a space takes the next character as an argument; in case \ifcd@NewLine is true, which it is at the beginning of every line (thanks to an \everypar), it increments \cd@FirstSpaces, which is used by \VerbBreak. A tab character does the same except that the \cd@FirstSpaces is increased by the value of \TabSize (stored in \cd@TabSize). In case the next character is not a space or a tab, \ifcd@NewLine is set to false.

Spaces leaves a \cd@ObeySpaces while tabs create an empty box of width \TabSize times the width of a space in the current font.

```
\gdef\cd@VerbSpace#1{%
\cd@ObeySpaces%
\ifcd@NewLine\advance\cd@FirstSpaces1\relax\fi%
\ifx#1^^I\else\ifx#1 \else\cd@NewLinefalse\fi\fi#1}%
\gdef\cd@VerbTab#1{%
\leavevmode\hbox%
to\cd@TabSize\fontdimen2\font{\hss}%
\ifcd@NewLine\advance\cd@FirstSpaces\cd@TabSize\fi%
\ifx#1^^I\else\ifx#1 \else\cd@NewLinefalse\fi\fi#1}
\egroup
```

\cd@Verbatim

Here comes the verbatimizer. First, we cancel the parindent and sets \hfuzz to \cd@BoxTolerance, which stores the argument of \BoxTolerance.

```
\def\cd@Verbatim{%
  \parindent\z@%
  \hfuzz=\cd@BoxTolerance%
```

Then, if a \ShortVerb was defined, we undo it, so that it appears as any other character in this context. If this verbatim was called by the \ShortVerb character itself, remember that it restores itself to 13.

```
  \ifcd@ShortVerb%
    \UndoShortVerb%
  \fi%
```

If we're not in a verbatim context called by \ShortCode, we undo it, for the same reason.

```
  \ifcd@ShortCode%
    \else%
    \ifcd@ShortCodeChar%
      \UndoShortCode%
    \fi%
  \fi%
```

We change the usual catcodes and reactivate the \ShortCode character, just in case it was changed by \dospecials or \@noligs. We activate the verb break and the verb command, and the rest is straightforward.

```
  \let\do\@makeother\dospecials\@noligs%
  \ifcd@ShortCode%
    \cd@ActivateShortCode%
  \fi%
  \cd@ActivateVerbBreak%
  \cd@ActivateVerbCommand%
  \frenchspacing%
  \catcode`\^^M=13\relax%
  \cd@MakeSpace}%
```

\BoxTolerance
\TabSize
\Gobble

These are pretty straigthforward too. I defined a macro instead of a simple dimension or number, because it seems to me that something like \TabSize{25} is much more common in the LaTeX world than \TabSize25. Besides, a \relax is automatically added, which avoids errors.

```
\newdimen\cd@BoxTolerance
\def\BoxTolerance#1{\cd@BoxTolerance=#1\relax}
\def\TabSize#1{\chardef\cd@TabSize=#1\relax}
\TabSize2
```

```
\def\Gobble#1{\chardef\cd@GobbleNum=#1\relax}
\Gobble0
```

5.4 The default `code` environment

`\CodeFont` The basic `code` environment is quite simple. First, we define `\CodeFont`, which simply stores its argument in `\cd@CodeFont`, to be released later. The following macros are explained more properly in the definition of `\NewExample` below.

```
\def\CodeFont#1{\def\cd@CodeFont{#1}}
\CodeFont{\ttfamily}
\newcount\code@cd@LineNumber
\def\code@cd@boxwidth{0pt}
\def\code@cd@BoxStyle{\rmfamily\footnotesize}
\gdef\code@cd@LineNumberBox{%
  \global\advance\code@cd@LineNumber1\relax%
  \def\@currentlabel{\code@cd@LineNumber}%
  \hbox to\code@cd@boxwidth{%
  \hss%
  \code@cd@BoxStyle\relax%
  \the\code@cd@LineNumber\enspace}}%
\let\code@cd@EOL\iffalse%
```

`\code` We create a paragraph and stores the name of the environment (used in `\VerbBreak` to check the width of the line number box).

```
\def\code{%
  \endgraf%
  \bgroup%
    \def\cd@ExampleName{code}%
```

We launch the verbatim definitions and the complicated `\cd@ObeyLines` (see below) that makes ends of lines work properly (gobbling characters if needed).

```
    \cd@Verbatim%
    \cd@ObeyLines%
```

Every new paragraph, i.e. every line in that context, typeset the line number and switches some values exlplained above. We also set the font.

```
    \everypar{%
      \code@cd@LineNumberBox
      \cd@NewLinetrue%
      \cd@FirstSpaces0\relax}%
    \cd@CodeFont%
```

Finally, we call the proper macro, depending on whether `\code` was called by `\begin{code}`, `\begin{code*}` or the `\ShortCode` character.

```
    \ifcd@ShortCode%
      \global\cd@ShortCodefalse%
      \let\cd@next\cd@ShortCode%
    \else\ifcd@Star%
      \global\cd@Starfalse%
      \let\cd@next\cd@StarCode%
    \else%
      \let\cd@next\cd@Code%
    \fi\fi\cd@next}
```

`\invisible` The starred variant of `\code` switches to true the conditional used just above. Let's also define the `invisible` environment, which takes an argument delimited by `\end{invisible}` and thus needs to turn some catcodes.

```
\expandafter\def\csname code*\endcsname{\cd@Startrue\code}
\def\invisible{%
  \bgroup%
  \catcode`\\=12 \catcode`\{=12 \catcode`\}=12 \catcode`\^^M=13 %
```

```
  \cd@Invisible}
```

The `^^?` character is used to delimit the end of the verbatim material (this is important because all ends of line scan ahead, see below). Since it is compared in an `\ifx` conditional, I define it to do nothing but with a distinct definition.

```
\gdef^^?{\cd@UnlikelyCommand}
\gdef\cd@UnlikelyCommand{}
```

`\cd@Code` `\cd@StarCode` `\cd@Invisible`

`\begin{code}` expects `\end{code}` while `\begin{code*}` expects `\end{code*}`. That's the reason why we distinguish `\cd@Code` and `\cd@StarCode`. Apart from that, they do the same: they typeset their argument (the first one is the end of the line) and close the environment. `\cd@StartGobble` is, obviously, the character gobbler for the first line. `\cd@Invisible` also matches its end but prints nothing.

```
\begingroup
\catcode`|=0
\catcode`<=1
\catcode`>=2
\catcode`{=12
\catcode`}=12
\catcode`\^^M=13 %
\catcode`\\=12 %
|gdef|cd@Code#1^^M#2\end{code}<|cd@StartGobble#2^^?|egroup|end<code>>%
|gdef|cd@StarCode#1^^M#2\end{code*}<|cd@StartGobble#2^^?|egroup|end<code*>>%
|gdef|cd@Invisible#1^^M#2\end{invisible}<|egroup|end<invisible>|ignorespaces>%
|endgroup
```

Here comes a fastidious part. Because we want to gobble characters at the beginning of each line (according to `\Gobble`), ends of lines do not simply create a new paragraph, they also give a look at the next line and gobble the adequate number of characters. Unfortunately, their definition changes slightly according to the context (default `code` and examples with or without ε-TEX). Let's set the stage.

```
\newcount\cd@GobbleCount%
\begingroup
\catcode`\^^M13\relax%
```

`\cd@StartGobble`

This is the gobbler called at the beginning of the material enclosed in a default `code` environment. If we meet `^^?`, i.e. if the environment is empty, we do nothing.

```
\gdef\cd@StartGobble#1{%
  \ifx#1^^?%
    \cd@GobbleCount=0 %
    \let\cd@next\relax%
```

Else, if we have reached the value set by `\Gobble` (stored in `\cd@GobbleNum`), we replace the token we were considering in the stream.

```
  \else\ifnum\cd@GobbleCount=\cd@GobbleNum%
    \cd@GobbleCount=0 %
    \def\cd@next{#1}%
```

If we meet an end of line character, that is, if the environment begins with a blank line, we put it back too (it will create a paragraph, among other things).

```
  \else\ifx#1^^M%
    \cd@GobbleCount=0 %
    \def\cd@next{^^M}%
```

Finally, if none of the above apply, we keep gobbling.

```
  \else%
    \advance\cd@GobbleCount1 %
    \let\cd@next\cd@StartGobble%
  \fi\fi\fi\cd@next}%
```

\cd@ObeyLines

In the code environment, ends of lines act exactly like \cd@StartGobble except that they create a paragraph in the first three cases.

```
\gdef\cd@ObeyLines{%
  \def^^M##1{%
    \ifx##1^^?%
    \cd@GobbleCount=0 %
    \def\cd@next{\leavevmode\endgraf}%
  \else\ifnum\cd@GobbleCount=\cd@GobbleNum%
    \cd@GobbleCount=0 %
    \def\cd@next{\leavevmode\endgraf##1}%
  \else\ifx##1^^M%
    \cd@GobbleCount=0 %
    \def\cd@next{\leavevmode\endgraf^^M}%
  \else%
    \advance\cd@GobbleCount1 %
    \let\cd@next^^M%
  \fi\fi\fi\cd@next}}%
\endgroup
```

5.5 Example environments

Examples are quite different from the default code environment, since they provide both the input and the output of a code. Besides, if available, they make use of ε-TEX.

\eTeXOn
\eTeXOff

Here's the command to switch from ε-TEX to external file.

```
\def\eTeXOn{%
  \@ifundefined{eTeXversion}%
    {\cd@Error{%
    You're not running on eTeX.\MessageBreak%
    Command \string\eTeXOn\space ignored}}%
    {\cd@eTeXtrue}}
\def\eTeXOff{\cd@eTeXfalse}
```

\NewExample
\cd@NewExample
\RenewExample
\cd@RenewExample
\cd@GobbleThree

\NewExample and \RenewExample work similarly but in an inverted way. Both test for options and launch \cd@@NewExample on the options and example name if nothing is wrong. Beforehand, they turn # into an active character, which will be \let later to the code material with additional macros.

```
\def\NewExample{%
  \@ifnextchar[%
    {\cd@NewExample}%
    {\cd@NewExample[]}}
\def\cd@NewExample[#1]#2{%
  \expandafter\ifx\csname #2\endcsname\relax
    \def\cd@next{\catcode`\#=13 \cd@@NewExample{#1}{#2}}%
  \else%
    \let\cd@next\relax%
    \cd@Error{%
    Style `#2' already defined or the name\MessageBreak%
    is already in use.\MessageBreak%
    Use \protect\RenewExample\space if you want to redefine it}%
    \let\cd@next\cd@GobbleThree%
  \fi\cd@next}

\def\RenewExample{%
  \@ifnextchar[%
    {\cd@RenewExample}%
    {\cd@RenewExample[]}}
\def\cd@RenewExample[#1]#2{%
  \expandafter\ifx\csname #2\endcsname\relax
    \let\cd@next\relax%
    \cd@Error{%
    Style `#2' is undefined.\MessageBreak%
```

```
    Use \protect\NewExample\space to redefine it}%
    \let\cd@next\cd@GobbleThree%
  \else\expandafter\ifx\csname #2\endcsname\code%
    \def\CodeFont{%
      \cd@Error{%
      You have redefined the ‘code’ environment.\MessageBreak%
      \string\CodeFont\space is no longer operative}}
  \fi%
    \def\cd@next{\catcode‘\#=13 \cd@@NewExample{#1}{#2}}%
  \fi\cd@next}

\def\cd@GobbleThree#1#2#3{}
```

`\cd@@NewExample`

Here is the working mechanism behind both `\NewExample` and `\RenewExample`. Since `#` will have a special function, we do some catcode changing. The definition is `\long`, of course.

```
\begingroup
\catcode‘\"=6 %
\catcode‘\#=13 %
\long\gdef\cd@@NewExample"1"2"3"4"5{%
```

We define some default values: `\⟨Example⟩@cd@EOL` is a switch used when the example is processed with ε-TEX, indicating whether ends of lines are visible or not. By default, they aren't, but options may change it. `\⟨Example⟩@cd@LineNumberBox` is the command used in examples to typeset the line number. By default, it is set to `\relax` because examples have no line number.

`\cd@ExampleName`

We store the name of the example to be retrieved when the environment is processed, but actually it is stored here for the options. Finally, we analyze options with a terminator.

```
  \expandafter\gdef\csname"2@cd@EOL\endcsname{\iffalse}%
  \expandafter\let\csname"2@cd@LineNumberBox\endcsname\relax%
  \def\cd@ExampleName{"2}%
  \cd@ExampleOptions"1,cd@end,%
```

`\CodeInput` `\CodeOutput` `\cd@MakeExample`

Now we define `\⟨Example⟩`, which will be called by `\begin{⟨Example⟩}`, as usual in LATEX. Each time, it redefines `\CodeInput` and `\CodeOutput`. Both store the name of the example, `\let #` to `\cd@Input` and `\cd@Output` respectively, whose definitions depends on the way the example is processed (ε-TEX or not), and finally execute the definition given by the user. `\cd@MakeExample` simply executes the last argument; it will be called at the end of the environment. Note the extra pairs of braces in all cases.

```
  \expandafter\def\csname"2\endcsname{%
    \gdef\CodeInput{{%
      \def\cd@ExampleName{"2}%
      \let#\cd@Input%
      "3}}%
    \gdef\CodeOutput{%
      \def\cd@ExampleName{"2}%
      \let#\cd@Output{"4}}%
    \gdef\cd@MakeExample{{"5}}%
```

Finally, we launch the example maker with the name of the environment (to match its proper end).

```
    \cd@Example{"2}}%
```

We also define the starred version of `\⟨Example⟩`, whose only difference is to switch the star conditional. Finally, we restore the category code of `#` and close.

```
  \expandafter\def\csname"2*\endcsname{%
    \global\cd@Startrue%
    \gdef\CodeInput{{%
    \def\cd@ExampleName{"2}%
    \cd@Startrue%
    \let#\cd@Input%
```

```
    "3}}%
    \gdef\CodeOutput{%
      \def\cd@ExampleName{"2}%
      \let#\cd@Output{"4}}%
    \gdef\cd@MakeExample{{"5}}%
    \cd@Example{"2*}}%
  \catcode`\#=6\relax}%
\endgroup
```

`\cd@numbered` `\cd@continuous` `\cd@visibleEOL` `\cd@empty`

Now we process options. First we define some keywords.

```
\def\cd@numbered{numbered}
\def\cd@continuous{continuous}
\def\cd@visibleEOL{visibleEOL}
\def\cd@empty{}
```

`\cd@ExampleOptions`

This is the option processor. It is recursive and stops when it meets the terminator. It simply stores the name of the option and acts accordingly.

```
\def\cd@ExampleOptions#1,{%
  \def\cd@TempOption{#1}%
  \let\cd@next\cd@ExampleOptions%
  \ifx\cd@TempOption\cd@end%
    \let\cd@next\relax%
```

If the option is `numbered`, we create a new count register, set the width of the box containing the number to 0pt by default, and define the style of this number to be `\relax` by default too. They will be modified by `\LineNumber`.

```
  \else\ifx\cd@TempOption\cd@numbered%
    \global\expandafter\newcount\csname\cd@ExampleName @cd@LineNumber\endcsname%
    \expandafter\gdef\csname\cd@ExampleName @cd@boxwidth\endcsname{0pt}%
    \expandafter\let\csname\cd@ExampleName @cd@BoxStyle\endcsname\relax%
```

We then define the macro executed by the environment for the line number; it increments the count, stores its value as the current label for `\label` and `\ref`, create a box of the desired width, flushes everything to the right, executes the style and typeset the value of the counter.

```
    \expandafter\gdef\csname\cd@ExampleName @cd@LineNumberBox\endcsname{%
      \expandafter\advance\csname\cd@ExampleName @cd@LineNumber\endcsname1\relax%
      \def\@currentlabel{\expandafter\the\csname\cd@ExampleName @cd@LineNumber\endcsname}%
      \hbox to\csname\cd@ExampleName @cd@boxwidth\endcsname{%
        \hss%
        \csname\cd@ExampleName @cd@BoxStyle\endcsname\relax%
        \expandafter\the\csname\cd@ExampleName @cd@LineNumber\endcsname\enspace}}%
```

If the option is `continuous`, we do the same thing, except that the count register is created if and only if it does not already exists (so that a modified `continuous` example environment will continue where it stopped; the user may use `\LineNumber` to start back from 0), and the `\advance` of the count is `\global`, so that the last value is always retained from one environment to the other.

```
  \else\ifx\cd@TempOption\cd@continuous%
    \expandafter\ifx\csname\cd@ExampleName @cd@LineNumber\endcsname\relax%
      \global\expandafter\newcount\csname\cd@ExampleName @cd@LineNumber\endcsname%
    \fi%
    \expandafter\gdef\csname\cd@ExampleName @cd@boxwidth\endcsname{0pt}%
    \expandafter\let\csname\cd@ExampleName @cd@BoxStyle\endcsname\relax%
    \expandafter\gdef\csname\cd@ExampleName @cd@LineNumberBox\endcsname{%
      \global\expandafter\advance\csname\cd@ExampleName @cd@LineNumber\endcsname1\relax%
      \def\@currentlabel{\expandafter\the\csname\cd@ExampleName @cd@LineNumber\endcsname}%
      \hbox to\csname\cd@ExampleName @cd@boxwidth\endcsname{%
        \hss%
        \csname\cd@ExampleName @cd@BoxStyle\endcsname\relax%
        \expandafter\the\csname\cd@ExampleName @cd@LineNumber\endcsname\enspace}}%
```

The `visibleEOL` option simply sets the relevant conditional to `true`.

```
  \else\ifx\cd@TempOption\cd@visibleEOL%
    \expandafter\gdef\csname\cd@ExampleName @cd@EOL\endcsname{\csname iftrue\endcsname}%
  \else\ifx\cd@TempOption\cd@empty%
  \else%
    \cd@Error{‘#1’ is not a valid option}%
  \fi\fi\fi\fi\fi\cd@next}%
```

`\LineNumber` `\cd@SetLineNumber`

`\LineNumber` is straightforward. After some testing, it sets the macro created above to the values specified. If a a square bracket follows, it executes `\cd@SetLineNumber`.

```
\def\LineNumber#1#2#3{%
  \expandafter\ifx\csname#1@cd@EOL\endcsname\relax%
    \cd@Error{‘#1’ is not an example environment’}%
  \else\expandafter\ifx\csname #1@cd@LineNumber\endcsname\relax%
    \cd@Warning{%
      ‘#1’ is not ‘numbered’ nor ‘continuous’.\MessageBreak%
      \string\LineNumber\space on line \the\inputlineno\space is useless}{}%
  \else%
    \expandafter\gdef\csname #1@cd@BoxStyle\endcsname{#2}%
    \expandafter\gdef\csname #1@cd@boxwidth\endcsname{#3}%
  \fi\fi%
  \@ifnextchar[{\cd@SetLineNumber#1}\relax}
\def\cd@SetLineNumber#1[#2]{%
  \expandafter\ifx\csname#1@cd@LineNumber\endcsname\relax%
  \else%
    \csname#1@cd@LineNumber\endcsname=#2\relax%
    \expandafter\advance\csname#1@cd@LineNumber\endcsname\m@ne%
  \fi}
```

`example`

The default `example` environment is thus easily created.

```
\NewExample{example}{\ttfamily#}{#}{}
```

`\CodeInput` `\CodeOutput`

If no example has been created, these two macros yields error messages.

```
\def\CodeInput{%
  \cd@Error{%
    No example environment has been created.\MessageBreak%
    \string\CodeInput\space is void}}
\def\CodeOutput{%
  \cd@Error{%
    No example environment has been created.\MessageBreak%
    \string\CodeOutput\space is void}}
```

And here comes the core example environment. First, some catcode changing.

```
\begingroup
\catcode‘|=0 %
\catcode‘<=1 %
\catcode‘>=2 %
\catcode‘{=12 %
\catcode‘}=12 %
\catcode‘\\=12 %
```

`\cd@Example`

This prepares the conditions for the processing of the material. Let’s start with the usual stuff:

```
|gdef|cd@Example#1<%
  |bgroup%
    |let|do|@makeother%
    |dospecials%
```

Now, if the environment was called by a \ShortCode character, there is no environment to close (\cd@EndEnv executes \end{⟨*Environment*⟩}). We call \cd@MakeExampleEnd, defined below, on the character, and we reactivate this character just in case it was one of the special.

```
    |ifcd@ShortCode%
      |global|let|cd@EndEnv|relax
      |expandafter|cd@MakeExampleEnd|expandafter<|cd@ShortEnd>%
      |global|cd@ShortCodefalse%
      |cd@ActivateShortCode%
```

If the environment was called by a regular \begin⟨*Environment*⟩ statement, we define the proper end (the argument comes from \⟨*Example*⟩, see the definition in \cd@@NewExample above). If there exists a \ShortCode character, we undefine it.

```
    |else%
      |gdef|cd@EndEnv<|end<#1>>%
      |cd@MakeExampleEnd<\end{#1}>%
      |ifcd@ShortCodeChar%
        |UndoShortCode%
      |fi%
    |fi%
```

If there's a short verb, we turn it off, we set tabs to 12 so they are written to the file as any other character, we activate ends of lines and in case ε-TEX is to process the example, we also activate comment characters (ε-TEX's scanning mechanism is peculiar and commented parts of the code wouldn't be taken into account otherwise).

```
    |ifcd@ShortVerb%
      |UndoShortVerb%
    |fi%
    |catcode`|^^I=12 %
    |catcode`|^^M=13 %
    |ifcd@eTeX%
      |catcode`|%=13 %
    |fi%
    |cd@ExampleEnd>%
|endgroup
```

\cd@MakeExampleEnd
\cd@ExampleEnd

\cd@MakeExampleEnd defines \cd@ExampleEnd so that the environment meets its proper end. It also launches the real processing, depending on the use of ε-TEX or not.

The argument has been passed in \cd@Example above, and is either \end{⟨*Environment*⟩} (with the proper catcodes) or the \ShortCode character.

In case we're using ε-TEX, we close some groups and environments, empty \everypar and assign the input. We switch the star conditional after that, because it is needed when the input is assigned and \cd@Verbatim is called.

```
\begingroup
\catcode`\^^M=13 %
%
\gdef\cd@MakeExampleEnd#1{%
  \ifcd@eTeX%
    \gdef\cd@ExampleEnd##1^^M##2#1{%
      \egroup%
      \cd@EndEnv%
      \bgroup%
        \everypar{}%
        \cd@AssigneTeXInput{##2}%
      \global\cd@Starfalse}%
```

If we're not using ε-TEX, we do some testing beforehand. We just want to inform the user that we're opening an external file. If it already exists, we keep silent.

```
  \else%
    \def\cd@ExampleEnd##1^^M##2#1{%
      \expandafter\ifx\csname cd@TestRead\endcsname\relax%
        \newread\cd@TestRead%
      \fi%
      \openin\cd@TestRead=\jobname.exp %
      \ifeof\cd@TestRead\relax%
        \cd@Warning{%
        You're not running on eTeX or you've said \string\eTeXOff.\MessageBreak%
        I create the file \jobname.exp to produce\MessageBreak%
        the example environment on line \the\inputlineno.\MessageBreak%
        You can delete it whenever you want, but\MessageBreak%
        keeping it prevents this message from reappearing.}%
      \fi%
      \closein\cd@TestRead %
```

`\cd@expFile` If it does not already exists, we create the output stream `\cd@expFile`, which opens an external scratch file for example processing.

```
      \expandafter\ifx\csname cd@expFile\endcsname\relax%
        \newwrite\cd@expFile%
      \fi%
        \immediate\openout\cd@expFile=\jobname.exp %
```

We `\let` ends of lines to a macro equivalent to the one described above for the default `code` environment, except that each line is written to the external file. We launch it on the material suffixed with a complicated tail to match all cases.

```
      \let^^M\cd@noeTeXEOL%
      ^^M##2^^?^^M^^?%
```

Finally, we close everything and assign input once again.

```
      \egroup%
      \cd@EndEnv%
      \immediate\closeout\cd@expFile%
      \bgroup%
        \everypar{}%
        \cd@AssignInput%
      \egroup\global\cd@Starfalse}%
  \fi}%
\endgroup
```

5.5.1 Examples without ε-TEX

`\cd@noeTeXEOL` Here's how ends of lines are processed when writing the code material to an external file. If we find `^^?`, which marks the end of the material, we stop.

```
\begingroup
\catcode`\^^M\active%
\gdef\cd@noeTeXEOL#1{%
  \ifx#1^^?%
    \cd@GobbleCount=0 %
    \let^^M\relax%
    \let\cd@next\relax%
```

If we find an end of line, that means there's a blank line, and we write it to the `jobname.exp`.

```
  \else\ifx#1^^M%
    \cd@GobbleCount=0 %
    \def\cd@next{\immediate\write\cd@expFile{}\cd@noeTeXEOL}%
```

If we have gobbled enough characters, we write the line to the external file. Otherwise, we repeat.

```
  \else\ifnum\cd@GobbleCount=\cd@GobbleNum%
    \cd@GobbleCount=0 %
    \def\cd@next{\cd@LineWrite#1}%
  \else%
    \advance\cd@GobbleCount1 %
    \let\cd@next\cd@noeTeXEOL%
  \fi\fi\fi\cd@next}%
```

`\cd@LineWrite` The line written is delimited by its end. This explains the `^^?^^M^^?` suffix at the end of the material on line 742. In case `\end{`⟨*Example*⟩`}` occurs on its own line, we need a terminator, hence the first `^^?`. If it occurs at the end of the last line, as in `... end of code\end{code}`, we need `^^M` so that the argument of `\cd@LineWrite` is properly delimited. The first `^^?` is then written to the file, but it expands to nothing. Since `\cd@LineWrite` calls `\cd@noeTeXEOL`, we need another delimitator, hence the second `^^?`.

```
\gdef\cd@LineWrite#1^^M{\immediate\write\cd@expFile{#1}\cd@noeTeXEOL}%
```

`\cd@AssignInput` Now we define the macro that will be used in `\CodeInput` (where `#` is `\let` to `\cd@Input`) and `\CodeOutput` (where it is `\let` to `\cd@Output`).

`\cd@Input` The input is quite similar to the default `code` environment. We define ends of lines as usual in verbatim contexts and we read from the scratch file.

```
\newtoks\cd@Everypar
%
\gdef\cd@AssignInput{%
  \gdef\cd@Input{%
    \bgroup%
      \cd@Everypar\everypar%
      \everypar{%
        \leavevmode\csname\cd@ExampleName @cd@LineNumberBox\endcsname\relax%
        \cd@NewLinetrue\cd@FirstSpaces0\relax\the\cd@Everypar\relax}%
      \cd@Verbatim%
      \def^^M{\leavevmode\endgraf}%
      \input{\jobname.exp}%
    \egroup}%
```

`\cd@Output` The output also reads from the file and simply ignores verb breaks and commands.

```
  \gdef\cd@Output{%
    \bgroup%
      \cd@IgnoreVerbBreak%
      \cd@IgnoreVerbCommand%
      \input{\jobname.exp}%
    \egroup}%
```

Finally, we execute the last argument to `\NewExample`, i.e. what was dubbed here ⟨*Immediate execution*⟩.

```
    \cd@MakeExample}%
```

5.5.2 Examples with ε-TEX

`\cd@AssigneTeXInput` Examples with ε-TEX are much more complicated. We use the `\scantokens` command, whose function is to read its argument as if catcodes were not fixed. For instance,
`\def\scan#1{{\catcode‘\\=12\scantokens{#1}}}`
`\scan\foo`
yields `\foo`, although the backslash was an escape character when read. The problem is that `\scantokens` interprets ends of lines and comments characters with their current values. Ends of lines yields a `\par` token as usual; the problem is that this token is scanned anew, and if you have turned the backslash to a category 12 character, it will appear as such. Moreover, commented parts of a line are ignored. For instance,
`\scan{`

```
a% mycomment

b}
```

yields `a\par b`. So `\scantokens` as it stands is not appropriate for verbatim material. 

The solution is to turned ends of lines and comments to other catcodes beforehand. Thus the previous example yields `a% mycomment^^M^^Mb^^M`. (The final end of line is added by `\scantokens`.) Now we need some hacking to produce the desired result.

`\cd@Input` The input begins with the usual verbatim preparation.

```
\long\gdef\cd@AssigneTeXInput#1{%
  \gdef\cd@Input{%
    \bgroup%
      \cd@Everypar\everypar%
      \everypar{%
      \leavevmode\csname\cd@ExampleName @cd@LineNumberBox\endcsname\relax%
      \cd@NewLinetrue\cd@FirstSpaces0\relax\the\cd@Everypar\relax}%
      \cd@Verbatim%
```

We define ends of lines as yet another gobbling mechanism. We use `^^?` once again to delimit material, and define it to make ends of lines ignored in case it is read, so that the additional `^^M` at the end of `\scantokens` will be ineffective.

```
      \catcode`\^^M=13 %
      \let^^M\cd@eTeXStartGobble%
      \catcode`\^^?13 %
      \def^^?{\catcode`\^^M=9\relax}%
      \scantokens{^^M#1^^?}%
    \egroup}%
```

`\cd@Output` Output is still worse. Even comments are active.

```
  \gdef\cd@Output{%
    \bgroup%
      \cd@IgnoreVerbBreak%
      \catcode`\^^?13 %
      \catcode`\%=13 %
      \catcode`\^^M=13 %
```

The next step depends on the user's choice about ends of lines. If they are visible, we process the material as is, with special definitions of `%` and `^^M` to mimick TEX's normal behavior.

```
      \csname\cd@ExampleName @cd@EOL\endcsname%
        \cd@VisibleComment%
        \let^^M\cd@eTeXOutVisibleEOL%
        \def^^?{\let^^M\relax}%
        \cd@IgnoreVerbCommand%
        \scantokens{#1^^?}%
```

If ends of lines are not visible, we execute the material beforehand with only `%`, `^^M` and `^^?` effective, to remove unwanted code. Macros are not executed because the backslash is still of category 12. Once ends of lines are thus processed, we scan everything anew, ignoring the last `^^M` and `^^@`, which has a special function (see below).

```
      \else%
        \cd@ActiveComment%
        \let^^M\cd@eTeXOutEOL%
        \def^^?{\catcode`\^^M9\relax}%
        \xdef\cd@exinput{#1^^?}%
        \cd@IgnoreVerbCommand%
        \catcode`\^^M=9 %
        \catcode`\^^@=9 %
        \expandafter\scantokens\expandafter{\cd@exinput}%
      \fi%
    \egroup}%
  \cd@MakeExample\egroup}%
```

\cd@eTeXStartGobble
\cd@eTeXEOL

Once again, macros to gobble the right number of characters at the beginning of each line. These are for the input. It is not possible to put `\cd@eTeXStartGobble` directly at the beginning of `\scantokens`, because the backslash would not be understood as an escape character. Thus we have to `\let ^^M` to it, and once it has done its job, make it change the meaning of `^^M` to `\cd@eTeXEOL`. (That's also the reason why we couldn't reuse the gobble macro of the default `code` environment, although they are quite similar.)

```
\gdef\cd@eTeXStartGobble#1{%
  \ifx#1^^?%
    \cd@GobbleCount=0 %
    \let\cd@next\relax%
  \else\ifnum\cd@GobbleCount=\cd@GobbleNum%
    \cd@GobbleCount=0 %
    \let^^M\cd@eTeXEOL%
    \def\cd@next{#1}%
  \else\ifx#1^^M%
    \cd@GobbleCount=0 %
    \let^^M\cd@eTeXEOL%
    \let\cd@next^^M%
  \else%
    \advance\cd@GobbleCount1 %
    \let\cd@next\cd@eTeXStartGobble%
  \fi\fi\fi\cd@next}%
%
\gdef\cd@eTeXEOL#1{%
  \ifx#1^^?%
    \cd@GobbleCount=0 %
    \def\cd@next{\let^^M\relax\leavevmode\endgraf}%
  \else\ifx#1^^M%
    \cd@GobbleCount=0 %
    \def\cd@next{\leavevmode\endgraf^^M}%
  \else\ifnum\cd@GobbleCount=\cd@GobbleNum%
    \cd@GobbleCount=0 %
    \def\cd@next{\leavevmode\endgraf#1}%
  \else%
    \advance\cd@GobbleCount1 %
    \let\cd@next^^M%
  \fi\fi\fi\cd@next}%
```

\cd@eTeXOutVisibleEOL

And now, the output. If ends of lines are visible, we set them to create a `\par` if the next character is another end of line (i.e. if we find a blank line) or to put it back into the stream otherwise, with a space before.

```
\gdef\cd@eTeXOutVisibleEOL#1{%
  \ifx#1^^?%
    \let^^M\relax%
    \let\cd@next\relax%
  \else\ifx#1^^M%
    \par%
    \let\cd@next^^M%
  \else%
    \def\cd@next{ #1}%
  \fi\fi\cd@next}%
```

\cd@eTeXOutEOL

If ends of lines are not visible, i.e. if they are processed before anything else, we do something similar, except that we add a dummy character, which will be ignored when the material is scanned, but will nonetheless prevent the formation of macro names across lines. Tail recursion is forbidden, since this will be used in a `\edef`, so we `\expandafter` instead.

```
\catcode`\^^@=12\relax%
\gdef\cd@eTeXOutEOL#1{%
  \ifx#1^^?%
  \else\ifx#1^^M%
```

```
    \par%
    \expandafter^^M%
  \else%
  ^^@ \expandafter\expandafter\expandafter#1%
  \fi\fi}%
```

Now we deal with comments. First we do some catcode changing. (We need a comment character since we're currently in a group where ends of lines are active).

```
\catcode`\/=14\relax%
\catcode`\%=13\relax/
\catcode`\ =12\relax/
\catcode`\^^I=12\relax/
```

`\cd@VisibleComment` `\cd@EatBOL`

If ends of lines are visible we define comments to eat everything until the end of the line and then launch a macro whose sole purpose is to remove spaces at the beginning of the next line.

```
\gdef\cd@VisibleComment{/
\def%##1^^M{\cd@EatBOL}/
\def\cd@EatBOL##1{/
\let\cd@next\cd@EatBOL/
\ifx##1 /
\else\ifx##1^^I/
\else\ifx##1^^M/
\let\cd@next\par/
\else/
\def\cd@next{##1}/
\fi\fi\fi\cd@next}}/
```

`\cd@ActiveComment` `\cd@EatBOL`

If ends of line are not visible, we do the same in the `\expandafter` way.

```
\gdef\cd@ActiveComment{/
\def%##1^^M{\cd@EatBOL}/
\def\cd@EatBOL##1{/
\ifx##1 /
\expandafter\cd@EatBOL/
\else\ifx##1^^I/
\expandafter\expandafter\expandafter\cd@EatBOL/
\else\ifx##1^^M/
\par/
\else/
\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter##1/
\fi\fi\fi}}/
\endgroup
```

5.6 File management

Here are some simple macro for the reader's relief.

`\CloseFile` Closing a file in normal mode simply makes all file identification macros unavailable.

```
\def\CloseFile#1{%
  \def\FileSource{%
    \cd@Error{%
      No file in production. \string\FileSource\space is empty}}%
  \def\FileName{%
    \cd@Error{%
      No file in production. \string\FileName\space is empty}}%
  \def\FileVersion{%
    \cd@Error{%
      No file in production. \string\FileVersion\space is empty}}%
  \def\FileDate{%
    \cd@Error{%
      No file in production. \string\FileDate\space is empty}}}
```

\@cd@LineCount

That's why, in normal mode, we close a file right now. We nonetheless create a dummy file name for the sake of \ProduceFile below.

```
\ifcd@produce%
  \def\FileName{}
  \def\FileVersion{}
  \def\FileDate{}
\else
  \CloseFile{}
  \def\FileSource{}
  \newcount\@cd@LineCount%
\fi
```

\ProduceFile

In normal mode, the main job of \ProduceFile is to reset some line number counts. In autoclose mode, there's only one counter, since files are closed when a new one is opened.

```
\def\ProduceFile#1{%
  \ifcd@autoclose%
    \code@cd@LineNumber0\relax%
```

If autoclose is off, we allocate a count for each file, so lines are numbered according to the file they belong to. We store the last value for the file we're going to close (stored in \FileSource), and set the line number of the code to the number for the file we're going to (re)open. That's why we needed a dummy \FileSource above, when \ProduceFile is executed for the first time.

```
  \else%
    \expandafter\csname\FileSource @cd@LineCount\endcsname=\code@cd@LineNumber%
    \expandafter\ifx\csname #1@cd@LineCount\endcsname\relax%
      \expandafter\newcount\csname #1@cd@LineCount\endcsname%
      \code@cd@LineNumber0\relax%
    \else%
      \expandafter\code@cd@LineNumber\csname #1@cd@LineCount\endcsname%
    \fi%
  \fi%
```

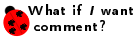
\FileSource

We reset \FileName and others, because their definition is optional. \FileSource is mandatory and is the actual argument of \ProduceFile. We launch the appropriate macro if a left bracket follows.

```
  \def\FileName{%
    \cd@Error{%
      No \string\FileName\space has been given to \FileSource}}%
  \def\FileVersion{%
  \cd@Error{%
    No \string\FileVersion\space has been given to \FileSource}}%
  \def\FileDate{%
  \cd@Error{%
    No \string\FileDate\space has been given to \FileSource}}%
  \edef\FileSource{#1}%
  \@ifnextchar[%
  {\cd@GetFileName}%
  \relax}
```

\cd@GetFileName
\FileName
\cd@GetFileVersion
\FileVersion
\cd@GetFileDate
\FileDate

These are straightforward and don't need any comment.




```
\def\cd@GetFileName[#1]{%
  \edef\FileName{#1}%
  \@ifnextchar[\cd@GetFileVersion\relax}
\def\cd@GetFileVersion[#1]{%
  \edef\FileVersion{#1}%
  \@ifnextchar[\cd@GetFileDate\relax}
\def\cd@GetFileDate[#1]{%
  \edef\FileDate{#1}}
```

\Header
\cd@HeaderGobble
\AddBlankLine
\StartIgnore
\StopIgnore
\DangerousEnvironment

Finally, we define those macros that have no effect in normal mode to have, well, no effect. Since comment signs are 'other' characters in produce mode, we change their catcode here too, so that the user may close the argument to \Header after a comment sign.

```
\def\Header{\bgroup\catcode`\%=12 \cd@HeaderGobble}
\long\def\cd@HeaderGobble#1{\egroup}
\let\AddBlankLine\relax
\let\StartIgnore\relax
\let\StopIgnore\relax
\def\DangerousEnvironment#1{}
```

6 Produce mode

We now turn to produce mode, where codedoc becomes CodeDoc and strange things happen.

6.1 Messages

\cd@Tracing
\cd@TChar
\cd@TUChar
\cd@TCode

CodeDoc may be quite talkative. According to the tracing option, we define some messages.

```
\ifcase\cd@tracingmode
  \def\cd@Tracing#1{}
  \def\cd@TChar#1#2{}
  \def\cd@TUChar#1{}
  \let\cd@TCode\relax
\or
  \def\cd@Tracing#1{}
  \def\cd@TChar#1#2{}
  \def\cd@TUChar#1{}
  \def\cd@TCode{\immediate\write17{%
    *** Code written from line \the\cd@ProduceLine\space to
    \the\inputlineno\space to \cd@CurrentFile. ***}}
\or
  \def\cd@Tracing#1{\immediate\write17{On line \the\cd@ProduceLine: #1.}}
  \def\cd@TChar#1#2{
    \bgroup
      \escapechar\m@ne\cd@Tracing{`\string#1' defined as \string\\#2}
    \egroup}
  \def\cd@TUChar#1{
    \bgroup
      \escapechar\m@ne\cd@Tracing{\string\\#1 undone}
    \egroup}
  \def\cd@TCode{\immediate\write17{%
    *** Code written from line \the\cd@ProduceLine\space to
    \the\inputlineno\space to \cd@CurrentFile. ***}}
\fi
```

\cd@Error
\cd@CDWarning
\cd@NoFileWarning

We also define errors and warnings; there's no need to follow LaTeX's ordinary syntax here.

```
\def\cd@CDError#1{%
  \immediate\write17{%
    ^^J! CodeDoc Error:^^J#1^^Jl.\the\cd@ProduceLine^^J }}
\def\cd@CDWarning#1{%
  \immediate\write17{%
    ^^J? CodeDoc Warning: ^^J#1^^Jl.\the\cd@ProduceLine^^J }}
\def\cd@NoFileWarning{\cd@CDWarning{No file in production. This code will be lost.}}
```

6.2 Testing strings

In produce mode, CodeDoc is a string tester; more precisely it imitates TeX's normal mechanism: the escape character is turned into an active character that gathers letters following it and executes the name they form (in a modified fashion, however, to execute only relevant macros).

\@documentclasshook

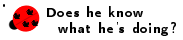
First, we redefine what happens at the end of the class to alter the behavior of special characters. However, we maintain comments and turn \ into an active character.

```
\ifcd@produce
\def\@documentclasshook{
  \let\do\@makeother
  \dospecials
  \catcode`\^^I=12\relax
  \catcode`\%=14\relax
  \catcode`\\\active
```

By default, `\normalsize` is an error message, so we redefine it. We start the report.

```
  \let\normalsize\relax
  \ifnum\cd@tracingmode>0
    \immediate\write17{^^J*** CODEDOC REPORT ***^^J}
  \fi
```

We don't load any font, so there's no need to bother with overfull boxes nor outputs. However, by pure superstition, I prefer some care.




```
  \hfuzz=100cm%
  \output={\deadcycles0\setbox0\box255}
  \everypar{}
```

Most of the following are already 0. However, `\tracingcommands2` would explode the log file, so we take some care once again.

```
  \tracingcommands\z@\tracingmacros\z@\tracingoutput\z@\tracingparagraphs\z@
  \tracingpages\z@\tracinglostchars\z@\tracingrestores\z@\tracingstats\z@}
\fi
```

`\cd@LeftBrace` `\cd@RightBrace` `\cd@LeftBracket` `\cd@Space` `\cd@Tab` `\cd@EndOfLine` `\cd@Comment`

Some characters are special, to say the least. We need to be able to recognize them.

```
\begingroup
\catcode`\{=12 %
\catcode`\}=12 %
\catcode`\<=1 %
\catcode`\>=2 %
\gdef\cd@LeftBrace<{>
\gdef\cd@RightBrace<}>
\gdef\cd@LeftBracket<[>
\catcode`\ =12\relax
\catcode`\^^I=12\relax
\gdef\cd@Space< >
\gdef\cd@Tab<^^I>
\catcode`\^^M=12\relax%
\gdef\cd@EndOfLine<^^M>%
\catcode`\/=14\relax/
\catcode`\%=12\relax/
\gdef\cd@Comment<%>/
\endgroup
```

`\cd@Escape`

Here comes the definition of the escape character as itself... The backslash can't be allowed to have catcode 0, otherwise control sequences would form and fire. We don't want that, obviously. On the other hand, some control sequences should be executed, so they must be form beforehand. Here's how \ works. First, it stores the current line number for messages.

```
\newcount\cd@ProduceLine

\begingroup
\catcode`|=0 %
\catcode`\\=13 %
|gdef|cd@Escape{\}%
|gdef\#1{%
  |cd@ProduceLine|inputlineno%
```

Then it turns ends of lines and comments to other characters, because we don't want to pass them unnoticed. If the next character is of category code 11, we start forming a control sequence. Otherwise, we gobble it and stop.

```
  |bgroup
    |catcode`|^^M=12 %
    |catcode`|%=12 %
    |gdef|cd@MacroName{}%
    |ifnum|catcode`#1=11 %
      |def|cd@next{|cd@Gather#1}%
    |else
      |def|cd@next{|egroup|relax}
    |fi
    |cd@next}
```

\cd@Gather
\cd@MacroName
\cd@NextChar

Forming macro names is quite simple: if the next character is a letter, we add it to the temporary name. Otherwise, we store it in `\cd@NextChar` and start doing what TEX does when it has formed a control sequence.

```
|long|gdef|cd@Gather#1{%
  |ifnum|catcode`#1=11 %
    |xdef|cd@MacroName{|cd@MacroName#1}%
    |let|cd@next|cd@Gather%
  |else%
    |gdef|cd@NextChar{#1}%
    |let|cd@next|cd@GobbleSpace%
    |fi|cd@next}
|endgroup
```

\cd@GobbleSpace

That is, we skip spaces and ends of lines, so that the *real* next character will be put next to the formed control sequence, in case it is an argument.

In case the next argument is none of the above, we call `\cd@Evaluate`, which will expand the macro, on the next character.

```
\long\def\cd@GobbleSpace{%
  \let\cd@next\cd@TakeNextChar
  \ifx\cd@NextChar\cd@Space
  \else\ifx\cd@NextChar\cd@Tab
  \else\ifx\cd@NextChar\cd@EndOfLine
  \else\ifx\cd@NextChar\cd@Comment
    \let\cd@next\cd@GobbleEndOfLine
  \else
    \egroup
    \def\cd@next{\expandafter\cd@Evaluate\cd@NextChar}
  \fi\fi\fi\fi\cd@next}
```

\cd@TakeNextChar
\cd@GobbleEndOfLine

These do what they say.

```
\long\def\cd@TakeNextChar#1{\gdef\cd@NextChar{#1}\cd@GobbleSpace}
\begingroup
\catcode`\^^M=12%
\gdef\cd@GobbleEndOfLine#1^^M#2{%
  \gdef\cd@NextChar{#2}%
  \cd@GobbleSpace}%
\endgroup
```

\cd@Evaluate

Finally, we take the name thus formed, and execute \⟨*Name*⟩`@Produce`. As you might imagine, the only macros containing the `@Produce` suffix are defined by CodeDoc. So, most of the time, this execution will be no more than a `\relax`. Which is exactly what we want.

```
\def\cd@Evaluate{\csname\cd@MacroName @Produce\endcsname}
```

6.3 Macros executed in produce mode

To understand what follows, simply remember that \⟨*Macro*⟩`@Produce` is executed when CodeDoc encounters \⟨*Macro*⟩. So, for instance, `\ShortVerb@Produce` is `\ShortVerb` in produce mode.

Macro names will become quite long, so we add some left margin.

`\cd@Gobble` First, some gobbler.

```
\def\cd@Gobble#1{}
```

`\cd@PrepareChar` Macros like `\ShortVerb` can take four kinds of argument. If you want + to be a `\ShortVerb`, you can say `\ShortVerb+`, `\ShortVerb\+`, `\ShortVerb{+}` and `\ShortVerb{\+}`. Since CodeDoc has already considered the next character when executing `\ShortVerb@Produce`, its catcode can't be changed, and a left brace is of category 12 and a backslash of category 13. So we have to gobble the next character if it is one of them.

`\cd@PrepareChar` takes a macro as an argument and replaces it in the stream with the next character gobbled or not. The backslash is turned into an escape character to handle the `\ShortVerb{\+}` case, where the left brace is gobbled; the backslash hasn't been read yet, so we can use it.

```
\def\cd@PrepareChar#1{%
  \catcode`\\=0 %
  \def\cd@next{\expandafter#1\cd@Gobble}
  \ifx\cd@NextChar\cd@LeftBrace%
  \else\ifx\cd@NextChar\cd@Escape%
  \else%
    \def\cd@next{#1}
  \fi\fi}%
```

`\ShortVerb@Produce` `\DefineShortVerb@Produce` `\cd@VerbList` Thus, `\ShortVerb@Produce` calls `\cd@PrepareChar` with `\cd@MakeShortVerb@Produce`, which will do the real job to the character. We define fancyvrb's `\DefineShortVerb` to do the same thing. `\cd@VerbList` contains all such characters, since `\DefineShortVerb` can define several of them. It will be used in writing environments to neutralize them.

```
\def\ShortVerb@Produce{\cd@PrepareChar\cd@MakeShortVerb@Produce\cd@next}
\let\DefineShortVerb@Produce\ShortVerb@Produce
\def\cd@VerbList{}
```

`\cd@MakeShortVerb@Produce` Now we inform the user that the character was `\ShortVerb`'ed.

```
\def\cd@MakeShortVerb@Produce#1{%
  \cd@TChar{#1}{ShortVerb}
```

We add it to `\cd@VerbList`.

```
  \expandafter\def\expandafter\cd@VerbList\expandafter{\cd@VerbList#1,}
```

`\cd@ShortVerb@Produce` `\UndoShortVerb@Produce` And we simply define the character to gobble everything until its next occurrence. We also define the `\Undo...` variant.

```
  \lccode`\~=`#1 %
  \lowercase{%
    \def~{\bgroup\let\do\@makeother\dospecials\catcode`#1\active\cd@ShortVerb@Produce}%
    \def\cd@ShortVerb@Produce##1~{\egroup}}%
  \def\UndoShortVerb@Produce{\cd@TUChar{ShortVerb}\catcode`#1=12\relax}%
  \catcode`#1=13 %
  \catcode`\\=13\relax}
\let\UndoShortVerb@Produce\relax
```

`\UndefineShortVerb@Produce` `\cd@UndefineShortVerb@Produce` We define a variant for fancyvrb, because it takes an argument.

```
\def\UndefineShortVerb@Produce{\cd@PrepareChar\cd@UndefineShortVerb@Produce\cd@next}%
\def\cd@UndefineShortVerb@Produce#1{
  \cd@TUChar{ShortVerb (from fancyvrb)}
  \catcode`#1=12 \catcode`\\=13\relax}
```

\VerbBreak@Produce
\cd@MakeVerbBreak@Produce
\cd@IgnoreVerbBreak
\UndoVerbBreak@Produce

In produce mode, the \VerbBreak character is simply ignored.

```
\def\VerbBreak@Produce{\cd@PrepareChar\cd@MakeVerbBreak@Produce\cd@next}
\def\cd@MakeVerbBreak@Produce#1{
  \cd@TChar{#1}{VerbBreak}
  \def\cd@IgnoreVerbBreak{\catcode`#1=9\relax}
  \def\UndoVerbBreak@Produce{\cd@TUChar{VerbBreak}\let\cd@IgnoreVerbBreak\relax}
  \catcode`\\=13\relax}
\let\cd@IgnoreVerbBreak\relax
\let\UndoVerbBreak@Produce\relax
```

\cd@GobbleOptions

This is useful for \ShortCode and also \NewExample

```
\def\cd@GobbleOptions#1[#2]#3{\def\cd@NextChar{#3}\expandafter#1\cd@NextChar}
```

\ShortCode@Produce

We check for options.

```
\def\ShortCode@Produce{%
  \ifx\cd@NextChar\cd@LeftBracket%
    \def\cd@next{\cd@GobbleOptions\ShortCode@Produce}%
  \else%
  \cd@PrepareChar\cd@MakeShortCode@Produce
  \fi\cd@next}
```

\cd@MakeShortcode@Produce
\cd@ShortWriteFile
\ActivateShortCode@Produce
\cd@UndoShortCode@Produce

The \ShortCode character in produce mode is similar to its counterpart in normal mode, except that it follows what code does in this mode. So give a look at the definition of the code environment to understand what is going on here.

```
\begingroup
\catcode`\^^M13%
\gdef\cd@MakeShortCode@Produce#1{%
  \cd@TChar{#1}{ShortCode}
  \lccode`\~=`#1%
  \lowercase{%
  \def~{\cd@ProduceLine\inputlineno\cd@ShortCodetrue\cd@CodeWrite}%
  \def\ActivateShortCode@Produce{\catcode`#1\active}%
  \def\cd@ShortWriteFile##1^^M##2~{%
    ^^M##2^^?^^M^^?%
    \ifx\cd@NoFileWarning\relax%
      \cd@TCode%
    \else%
      \cd@NoFileWarning%
    \fi\egroup}}%
  \def\UndoShortCode@Produce{\cd@TUChar{ShortCode}\catcode`#1=12\relax}%
  \catcode`\\=13 %
  \catcode`#1=13\relax}%
\endgroup
\let\ActivateShortCode@Produce\relax
\let\UndoShortCode@Produce\relax
```

\VerbCommand@Produce
\cd@VerbEscape@Produce
\cd@TempEsc

\VerbCommand characters do what they do in \CodeOutput in normal mode. The escape gobble letters and the braces gobble what they contain.

First, we store the escape character for the message.

```
\def\VerbCommand@Produce{\cd@PrepareChar\cd@VerbEscape@Produce\cd@next}
\def\cd@VerbEscape@Produce#1{
  \bgroup\escapechar\m@ne\xdef\cd@TempEsc{\string#1}\egroup
```

\cd@IgnoreEscape@Produce

Then we turn it into a letter gobbler.

```
  \def\cd@IgnoreEscape@Produce{
    \catcode`#1=13
    \lccode`\~=`#1
    \lowercase{\def~{\cd@GobbleLetters}}}
```

This is not what you think it is. We're not considering whether the character to come is a left brace, but whether `\cd@NextChar`, i.e. the character following `\VerbCommand`, *was* a left brace; this means that a right brace is to come, and we want to gobble it before processing what follows.

```
    \ifx\cd@NextChar\cd@LeftBrace
      \def\cd@next{\expandafter\cd@VerbBraces@Produce\cd@Gobble}
    \else
      \let\cd@next\cd@VerbBraces@Produce
    \fi\cd@next}
```

`\cd@VerbBraces@Produce` `\cd@@VerbBraces@Produce` `\cd@IgnoreBraces@Produce` `\UndoVerbCommand@Produce`

The rest is pretty straightforward and similar to what we did in normal mode.

```
\def\cd@VerbBraces@Produce{\catcode`\{=1 \catcode`\}=2 \cd@@VerbBraces@Produce}
\def\cd@@VerbBraces@Produce#1#2{%
  \expandafter\cd@TChar\expandafter{\cd@TempEsc', `\string#1' and `\string#2'}{VerbCommand}
  \def\cd@IgnoreBraces@Produce{%
    \catcode`#1=13
    \lccode`\~=`#1
    \lowercase{\def~####1#2{}}}
  \catcode`\\=13 \catcode`\{=12 \catcode`\}=12\relax}
\def\UndoVerbCommand@Produce{
  \cd@TUChar{VerbCommand}
  \let\cd@IgnoreEscape@Produce\relax
  \let\cd@IgnoreBraces@Produce\relax}
\let\cd@IgnoreEscape@Produce\relax
\let\cd@IgnoreBraces@Produce\relax
```

`\CodeEscape@Produce` `\cd@CodeEscape@Produce` `\cd@ActivateCodeEscape` `\UndoCodeEscape@Produce`

`\CodeEscape` is easy: we simply define a macro to turn the character into an escape in code contexts.

```
\def\CodeEscape@Produce{\cd@PrepareChar\cd@CodeEscape@Produce\cd@next}
\def\cd@CodeEscape@Produce#1{%
  \cd@TChar{#1}{CodeEscape}
  \def\cd@ActivateCodeEscape{\catcode`#1=0\relax}\catcode`\\=13\relax}
\let\cd@ActivateCodeEscape\relax
\def\UndoCodeEscape@Produce{\cd@TUChar{CodeEscape}\let\cd@ActivateCodeEscape\relax}
```

`\NewExample@Produce` `\RenewExample@Produce`

These two macros launch the option gobbler if there are any. `\cd@DangerousExample@Produce` is defined later because it takes its argument between braces of category 12, like other macros.

```
\def\NewExample@Produce{%
  \ifx\cd@NextChar\cd@LeftBracket%
    \def\cd@next{\cd@GobbleOptions\NewExample@Produce}%
  \else%
    \let\cd@next\cd@DangerousExample@Produce%
  \fi\cd@next}
\let\RenewExample@Produce\NewExample@Produce
```

`\cd@@Evaluate` `\StartIgnore@Produce` `\cd@FindIgnore` `\cd@StopIgnore`

Ignoring the input boils down to modifying the definition of `\cd@Evaluate` until it founds `\StopIgnore`. Meanwhile, it does nothing.

```
\let\cd@@Evaluate\cd@Evaluate
\def\StartIgnore@Produce{
  \cd@Tracing{\string\StartIgnore\space found. I will ignore everything from now on}
  \let\cd@Evaluate\cd@FindIgnore}
\def\cd@FindIgnore{
  \expandafter\ifx\csname cd@\cd@MacroName\endcsname\cd@StopIgnore
    \cd@Tracing{\string\StopIgnore\space found. I resume my normal behavior}
    \let\cd@Evaluate\cd@@Evaluate
  \fi}
\def\cd@StopIgnore{\cd@StopIgnore}
```

\verb@Produce
\cd@VerbEater
\cd@@VerbEater

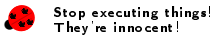
The produce version of LaTeX's \verb gobbles its argument after it has checked for a star.

```
\def\verb@Produce{\count@=0 \cd@VerbEater}
\def\cd@VerbEater#1{%
  \ifcase\count@ %
    \ifx#1*
      \count@=1 %
      \let\cd@@VerbEater\cd@VerbEater
    \else
      \def\cd@@VerbEater##1#1{}
    \fi
  \else
    \def\cd@@VerbEater##1#1{}
  \fi\cd@@VerbEater}
```

\DescribeMacro@Produce
\DefineMacro@Produce
\DescribeEnvironment@Produce
\DefineEnvironment@Produce
\noexpand@Produce
\string@Produce
\protect@Produce

The normal counterparts of these might take dangerous arguments, so we need to neutralize them. The first four gobble two tokens, i.e. a left brace and/or an escape character, so the following macro won't form. The last three just gobble the escape character.

```
\def\DescribeMacro@Produce#1#2{}
\def\DefineMacro@Produce#1#2{}
\def\DescribeEnvironment@Produce#1#2{}
\def\DefineEnvironment@Produce#1#2{}
\def\noexpand@Produce#1{}
\def\string@Produce#1{}
\def\protect@Produce#1{}
```



\begin@Produce
\end@Produce

\begin and \end statements are executed if and only if there follows a left brace. This decreases the number of possible errors. The double-@ versions take their arguments in 'other' braces, so they are defined later.

```
\def\begin@Produce{
  \ifx\cd@NextChar\cd@LeftBrace
    \expandafter\begin@@Produce
  \fi}
\def\end@Produce{
  \ifx\cd@NextChar\cd@LeftBrace
    \expandafter\end@@Produce
  \fi}
```

\Gobble@Produce

The produce version of \Gobble is similar to the normal version, except that it take cares of braces. \Gobble@@Produce is defined below.

```
\def\Gobble@Produce#1{%
  \ifx\cd@NextChar\cd@LeftBrace%
    \def\cd@next{\expandafter\Gobble@@Produce\cd@NextChar}
  \else
    \def\cd@next{\chardef\cd@GobbleNum=#1\relax}%
  \fi\cd@next}
```

\Header@Produce
\cd@HeaderEOL

The header is an easy matter. The only thing not to forget is to change the catcode of \back to 0.

```
\newif\ifcd@HeaderFirstLine
\begingroup
\catcode`\^^M=13 %
\catcode`\/=14 %
\catcode`\%=12 /
\gdef\Header@Produce{/
  \bgroup/
  \catcode`\^^M=13 /
  \catcode`\%=12 /
  \catcode`\\=0 /
```

```
  \Header@@Produce}/
\gdef\cd@HeaderEOL{\def^^M{^^J% }}
\endgroup
```

`\cd@DocumentString` `\cd@CodeString` `\cd@StarCodeString` `\cd@InvisibleString` `\cd@StoredEnvironments`

We'll need these presently.

```
\def\cd@DocumentString{document}
\def\cd@CodeString{code}
\def\cd@StarCodeString{code*}
\def\cd@InvisibleString{invisible}
\def\cd@StoredEnvironments{example,verbatim,Verbatim,BVerbatim,
LVerbatim,SaveVerbatim,VerbatimOut,Example,CenterExample,
SideBySideExample,PCenterExample,PSideBySideExample,}
```

Here comes the macros that take their arguments bewteen braces of category 12.

```
\begingroup
\catcode`\{=12 %
\catcode`\}=12 %
\catcode`\<=1 %
\catcode`\>=2 %
```

`\Header@@Produce`

This defines `\cd@Header`, which is executed in `\ProduceFile`, to write the text input by the user to the newly opened file. The group we close was opened in `\Header@Produce`.

```
\long\gdef\Header@@Produce{#1}<
  \gdef\cd@Header<\bgroup\cd@HeaderEOL\cd@ProduceFile<\cd@Comment\space#1>\egroup>
  \egroup>
```

`\Gobble@@Produce`

This is launched by `\Gobble@Produce`

```
\gdef\Gobble@@Produce{#1}<\chardef\cd@GobbleNum=#1\relax>
```

`\DangerousEnvironment@Produce` `\cd@DangerousExample@Produce` `\DefineVerbatimEnvironment`

Here we add dangerous environments to the list above, to be checked below.

`\cd@DangerousExample@Produce` has such a cumbersome definition because it is meant to gobble the remaining three arguments of `\NewExample` and `\RenewExample`. They might be separated by spaces, and since spaces have category 12 in produce mode, they won't be skipped and `\cd@DangerousExample@Produce` wouldn't match its definition, as TEX likes to say.

```
\gdef\DangerousEnvironment@Produce{#1}<
  \cd@Tracing<#1 added to dangerous environments>
  \xdef\cd@StoredEnvironments<\cd@StoredEnvironments#1,>>
\gdef\cd@DangerousExample@Produce{#1}#2{#3}#4{#5}#6{#7}<
  \cd@Tracing<#1 added to dangerous environments (CodeDoc examples)>
  \xdef\cd@StoredEnvironments<\cd@StoredEnvironments#1,>>
\let\DefineVerbatimEnvironment@Produce\DangerousEnvironment@Produce
```

`\begin@@Produce`

`\begin` statements simply check their argument: if it is `code`, `code*` or `invisible`, it turns to writing mode. Otherwise, the name of the argument is checked against the list of dangerous environments. See below where normal braces are restored.

```
\gdef\begin@@Produce{#1}<
  \def\cd@TempArg<#1>
  \ifx\cd@TempArg\cd@CodeString
    \let\cd@next\cd@CodeWrite
  \else\ifx\cd@TempArg\cd@StarCodeString
    \cd@Startrue
    \let\cd@next\cd@CodeWrite
  \else\ifx\cd@TempArg\cd@InvisibleString
    \cd@Invisibletrue
    \let\cd@next\cd@CodeWrite
  \else
    \def\cd@next<\cd@CheckEnvironment<#1>>
  \fi\fi\fi\cd@next>
```

\end@Produce

There's only one thing that can wake an \end statement: document. If it finds \end{document}, CodeDoc stops. Otherwise, \end statements are ignored.

```
\gdef\end@@Produce{#1}<
  \def\cd@TempArg<#1>
  \ifx\cd@TempArg\cd@DocumentString
    \def\cd@next<\cd@Tracing<\string\end{document}>
    \ifnum\cd@tracingmode=0 %
    \else
      \immediate\write17<^^J*** END OF CODEDOC REPORT ***^^J>
    \fi\@@end>
  \else
    \let\cd@next\relax
  \fi\cd@next>
```

\ProduceFile@Produce
\CloseFile@Produce

We define these right now, to be used later.

```
\gdef\ProduceFile@Produce{#1}<\ProduceFile@@Produce<#1>>
\gdef\CloseFile@Produce{#1}<\CloseFile@@Produce<#1>>
```

\input@Produce
\cd@CurrentSource

We need a terribly boring definition of \input for the default header, so that files are properly tracked back to their source. Besides, \input in TEX's way, i.e. without braces, is not allowed anymore, if it is to be read by CodeDoc in produce mode. I feel like removing the whole thing altogether.

```
\newcount\cd@InputDepth
\gdef\input@Produce{#1}<
  \cd@Tracing<\string\input\space file #1>
  \expandafter\let\csname cd@MasterSource\the\cd@InputDepth\endcsname\cd@CurrentSource
  \edef\cd@CurrentSource<#1 (\string\input\space in \cd@CurrentSource)>
  \advance\cd@InputDepth1\relax
  \@@input #1\relax
  \advance\cd@InputDepth-1\relax
  \expandafter\let\expandafter\cd@CurrentSource\csname cd@MasterSource
  \the\cd@InputDepth\endcsname>
```

\cd@MakeSpecialEater
\cd@SpecialEater

If we find a dangerous environment, we launch this on its name, which eats everything until \end{⟨*Name*⟩}.

```
\catcode`\|=0 %
\catcode`\\=13 %
|gdef|cd@MakeSpecialEater#1<
  |long|def|cd@SpecialEater##1\end{#1}<>
  |cd@SpecialEater>
|endgroup
```

\cd@CurrentSource

Back to normal braces. This is a default value needed in \input@Produce. The extension is just a guess, of course.

```
\edef\cd@CurrentSource{\jobname.tex}
```

\cd@CheckEnvironment
\cd@@CheckEnvironment

This is the checking mechanism used in \begin statement to detect dangerous environments. Note that we check all environments in their starred version too.

```
\def\cd@CheckEnvironment#1{
  \def\cd@TempEnv{#1}
  \expandafter\cd@@CheckEnvironment\cd@StoredEnvironments cd@end,}
\def\cd@@CheckEnvironment#1,{
  \def\cd@@TempEnv{#1}
  \def\cd@@StarTempEnv{#1*}
  \ifx\cd@@TempEnv\cd@end
    \let\cd@next\relax
  \else\ifx\cd@@TempEnv\cd@TempEnv
    \def\cd@next{\cd@MakeSpecialEater{#1}}
  \else\ifx\cd@@StarTempEnv\cd@TempEnv
    \def\cd@next{\cd@MakeSpecialEater{#1*}}
  \else
```

```
    \let\cd@next\cd@@CheckEnvironment
  \fi\fi\fi
  \cd@next}
```

6.4 Writing environments

CodeDoc looks for `code`, `code*` and `invisible` environments and process them line by line.

`\cd@MakeOther` First, we need a recursive catcode changer.

```
\def\cd@MakeOther#1,{%
  \def\cd@TempArg{#1}%
  \ifx\cd@TempArg\cd@end%
  \else%
    \catcode`#1=12 %
    \expandafter\cd@MakeOther%
  \fi}
```

`\cd@CodeWrite` This is the writing macro, called by `\begin` when the appropriate argument is found, or by the `\ShortCode` character. `\dospecials` is probably useless since all specials are already done, but at least it changes the category of the escape and the comment.

```
\newif\ifcd@Invisible
\begingroup
\catcode`\^^M=13\relax%
\gdef\cd@CodeWrite{%
  \bgroup%
    \let\do\@makeother%
    \dospecials%
    \catcode`\^^I=12 %
```

We turn all verb characters (defined by fancyvrb's `\DefineShortVerb`) into other characters, ignore the verb break, neutralize the short code if we're not in a short code environment (the redefinition of `\cd@TUChar` just prevents an unwanted message sent to the user if `tracing` is 2) an reactivate it otherwise, ignore `\VerbCommand` and activate `\CodeEscape`. We turn ends of lines into proper gobbler once again.

```
    \expandafter\cd@MakeOther\cd@VerbList cd@end,%
    \cd@IgnoreVerbBreak%
    \ifcd@ShortCode%
      \ActivateShortCode@Produce%
    \else%
      \let\cd@TempTUChar\cd@TUChar
      \def\cd@TUChar##1{}
      \UndoShortCode@Produce%
      \let\cd@TUChar\cd@TempTUChar
    \fi%
    \cd@IgnoreEscape@Produce%
    \cd@IgnoreBraces@Produce%
    \cd@ActivateCodeEscape%
    \catcode`\^^M=13\relax%
    \let^^M\cd@produceEOL%
```

Finally we launch the adequate macro. They all do the same thing, but they look for different `\end` statements.

```
    \ifcd@ShortCode%
      \global\cd@ShortCodefalse\let\cd@next\cd@ShortWriteFile%
    \else\ifcd@Star%
      \global\cd@Starfalse\let\cd@next\cd@StarWriteFile%
    \else\ifcd@Invisible%
      \global\cd@Invisiblefalse\let\cd@next\cd@InvisibleWriteFile%
    \else%
      \let\cd@next\cd@WriteFile%
    \fi\fi\fi\cd@next}%
```

\cd@ProduceEOL
\cd@LineWrite@Produce

This is similar to the version for examples without ε-TEX in normal mode, i.e. it writes to an external file, specified in `\cd@ProduceFile`.

```
\gdef\cd@produceEOL#1{%
  \ifx#1^^?%
    \cd@GobbleCount=0 %
    \let^^M\relax%
    \let\cd@next\relax%
  \else\ifx#1^^M%
    \cd@GobbleCount=0 %
    \def\cd@next{\cd@ProduceFile{}\cd@produceEOL}%
  \else\ifnum\cd@GobbleCount=\cd@GobbleNum%
    \cd@GobbleCount=0 %
    \def\cd@next{\cd@LineWrite@Produce#1}%
  \else%
    \advance\cd@GobbleCount1 %
    \let\cd@next\cd@produceEOL%
  \fi\fi\fi\cd@next}%
\gdef\cd@LineWrite@Produce#1^^M{\cd@ProduceFile{#1}\cd@produceEOL}%
```

\cd@WriteFile
\cd@StarWriteFile
\cd@InvisibleWriteFile

And here is the end. It is the first `^^M`, `\let` to `\cd@ProduceEOL`, which launches everything. The conditional switches between an error message (no file in production) and a report (code written).

```
\catcode`|=0 %
\catcode`<=1 %
\catcode`>=2 %
\catcode`{=12 %
\catcode`}=12 %
\catcode`\\=12 %
|long|gdef|cd@WriteFile#1^^M#2\end{code}<%
  ^^M#2^^?^^M^^?%
  |ifx|cd@NoFileWarning|relax%
    |cd@TCode%
  |else%
    |cd@NoFileWarning%
  |fi|egroup>%
|long|gdef|cd@StarWriteFile#1^^M#2\end{code*}<%
  ^^M#2^^?^^M^^?%
  |ifx|cd@NoFileWarning|relax%
    |cd@TCode%
  |else%
    |cd@NoFileWarning%
  |fi|egroup>%
|long|gdef|cd@InvisibleWriteFile#1^^M#2\end{invisible}<%
  ^^M#2^^?^^M^^?%
  |ifx|cd@NoFileWarning|relax%
    |cd@TCode%
  |else%
    |cd@NoFileWarning%
  |fi|egroup>%
|endgroup
```

6.5 File management

This the final step: handling files in produce mode.

\cd@Closed
\cd@Open
\cd@Wait

First, some keywords.

```
\def\cd@Closed{closed}
\def\cd@Open{open}
\def\cd@Wait{wait}
```

\cd@CurrentFile
\cd@ProduceFile
\AddBlankLine@Produce

Some basic definitions. \@unused is LaTeX's unattributed stream for messages. We let it write to he log file. \cd@ProduceFile is the writing macro (used in writing environments above); as long as no file is open, it does nothing.

```
\newcount\cd@ProduceCount

\def\cd@CurrentFile{}
\chardef\@unused=17

\def\cd@ProduceFile#1{}
\def\AddBlankLine@Produce{\cd@ProduceFile{}}
```

\ProduceFile@@Produce

This is called by \ProduceFile, via \ProduceFile@Produce above. If the file is closed ore already in production, we signal it to the user:

```
\def\ProduceFile@@Produce#1{%
  \let\cd@next\relax
  \expandafter\ifx\csname #1@Status\endcsname\cd@Closed
    \cd@CDError{%
    File `#1' has already been closed.^^J%
    If I open it again, it will be erased.^^J%
    I can't do that. I quit. Sorry.}
    \let\cd@next\@@end
  \else\expandafter\ifx\csname #1@Status\endcsname\cd@Open
    \cd@CDWarning{%
    File `#1' is currently in production.^^J%
    Why do you try to open it again?}
```

The file is waiting if it has been opened previously and another one has been opened too afterward, provided autoclose is off. In which case, we set it to open:

```
  \else\expandafter\ifx\csname #1@Status\endcsname\cd@Wait
    \expandafter\let\csname #1@Status\endcsname\cd@Open
```

We disable the warning about the absence of a file in production and define \cd@ProduceFile to write to this file.

```
    \let\cd@NoFileWarning\relax
    \def\cd@ProduceFile{\immediate\write\csname #1@Stream\endcsname}
```

We set the current file to wait and define the one we're dealing with to be the current file.

```
    \expandafter\let\csname \cd@CurrentFile @Status\endcsname\cd@Wait
    \def\cd@CurrentFile{#1}
```

Now, if the file has never been opened, we need an output stream. If they were all allocated, we look whether some were made available thanks to a \CloseFile.

```
  \else\ifnum\cd@ProduceCount>15
    \chardef\cd@ProduceStream=16
    \expandafter\cd@FindStream\cd@StreamList cd@end,
```

If no stream is found, CodeDoc feels so bad that it quits.

```
    \ifnum\cd@ProduceStream=16 %
      \cd@CDError{%
      No more stream for a new file. Close one with \string\CloseFile\space^^J%
      (or use the `autoclose' option).^^J%
      This situation makes me feel bad. I quit.}
      \let\cd@next\@@end
```

Else, we're very happy, and if there is already a file in production, we close it or let it wait.

```
    \else
      \cd@Tracing{I will now produce file #1}
      \ifx\cd@CurrentFile\cd@empty
```

```
      \else
        \ifcd@autoclose
          \cd@Tracing{I close file \cd@CurrentFile\space (autoclose mode)}
          \expandafter\let\csname \cd@CurrentFile @Status\endcsname\cd@Closed
        \else
          \expandafter\let\csname \cd@CurrentFile @Status\endcsname\cd@Wait
        \fi
      \fi
```

Then we define our file as the current one, let the world know that it is open, allocate the stream to its name, open it, etc., and launch a macro to retrieve some information if any.

```
      \def\cd@CurrentFile{#1}
      \expandafter\let\csname #1@Status\endcsname\cd@Open
      \expandafter\chardef\csname #1@Stream\endcsname\cd@ProduceStream
      \immediate\openout\cd@ProduceStream=#1 %
      \let\cd@NoFileWarning\relax
      \def\cd@ProduceFile{\immediate\write\cd@ProduceStream}
      \let\cd@next\cd@GetFile@Produce
    \fi
```

If there was an available stream in the first place, we do exactly the same.

```
  \else\chardef\cd@ProduceStream\cd@ProduceCount
    \cd@Tracing{I will now produce file #1}
    \ifx\cd@CurrentFile\cd@empty
    \else
      \ifcd@autoclose
        \cd@Tracing{I close file \cd@CurrentFile\space (autoclose mode)}
        \expandafter\let\csname \cd@CurrentFile @Status\endcsname\cd@Closed
      \else
        \expandafter\let\csname \cd@CurrentFile @Status\endcsname\cd@Wait
      \fi
    \fi
    \def\cd@CurrentFile{#1}
    \expandafter\let\csname #1@Status\endcsname\cd@Open
    \expandafter\chardef\csname #1@Stream\endcsname\cd@ProduceStream
    \immediate\openout\cd@ProduceStream=#1 %
    \let\cd@NoFileWarning\relax
    \def\cd@ProduceFile{\immediate\write\cd@ProduceStream}
    \ifcd@autoclose
    \else
      \advance\cd@ProduceCount\@ne
    \fi
    \let\cd@next\cd@GetFile@Produce
  \fi\fi\fi\fi\cd@next}
```

`\cd@GetFile@Produce`
`\cd@GetFileName@Produce`
`\cd@GetFileVersion@Produce`
`\cd@GetFileDate@Produce`

This is designed to retrieve optional information following `\ProduceFile`. We undo the `\ShortVerb` and `\ShortCode` because they might appear there. (My `\ShortCode` is a slash, which is used in date too.) We also set the backslash as an escape character, because control sequences might appear here.

In all cases, if nothing follows, and if the `noheader` option is off, we write the header to the file.

```
\def\cd@GetFile@Produce{
  \bgroup
    \UndoShortCode@Produce
    \UndoShortVerb@Produce
    \catcode`\\\z@
    \gdef\FileName{}
    \gdef\FileVersion{}
    \gdef\FileDate{}
    \@ifnextchar[
```

```
      \cd@GetFileName@Produce
      {\ifcd@noheader\else\cd@Header\fi}}
\def\cd@GetFileName@Produce[#1]{
  \xdef\FileName{#1}
  \@ifnextchar[
    \cd@GetFileVersion@Produce
    {\ifcd@noheader\else\cd@Header\fi\egroup}}
\def\cd@GetFileVersion@Produce[#1]{%
  \xdef\FileVersion{#1}
  \@ifnextchar[
    \cd@GetFileDate@Produce
    {\ifcd@noheader\else\cd@Header\fi\egroup}}
\def\cd@GetFileDate@Produce[#1]{%
  \xdef\FileDate{#1}
  \ifcd@noheader\else\cd@Header\fi\egroup}
```

`\CloseFile@@Produce` Closing a file is a lot of uninteresting testing...

```
\def\CloseFile@@Produce#1{
  \ifcd@autoclose
    \expandafter\ifx\csname #1@Status\endcsname\relax
      \cd@CDWarning{%
        You haven't opened `#1'. Closing it does nothing.^^J%
        Besides, you're in autoclose mode. \string\CloseFile\space is redundant.}
    \else\expandafter\ifx\csname #1@Status\endcsname\cd@Closed
      \cd@CDWarning{%
        `#1' was already closed. Closing it again does nothing.^^J%
        Besides, you're in autoclose mode. \string\CloseFile\space is redundant.}
    \else
      \cd@CDWarning{%
        You're in autoclose mode.  \string\CloseFile\space is redundant.}
    \fi\fi%
  \else
    \expandafter\ifx\csname #1@Status\endcsname\relax
      \cd@CDWarning{%
        You haven't opened `#1'. Closing it does nothing.}
    \else\expandafter\ifx\csname #1@Status\endcsname\cd@Closed
      \cd@CDWarning{%
        `#1' was already closed. Closing it again does nothing.}
```

If everything is okay, beside closing the file, we also define the no-file warning and neutralize the writing macro. We also add the stream allocated to that file to `\cd@StreamList`, so that it may be retrieved if all other streams are unavailable.

```
    \else
      \cd@Tracing{I close file #1}
      \expandafter\let\csname #1@Status\endcsname\cd@Closed
      \def\cd@TempFile{#1}
      \ifx\cd@TempFile\cd@CurrentFile
        \def\cd@NoFileWarning{\cd@CDWarning{No file in production.
        This code will be lost.}}
        \def\cd@ProduceFile##1{}%
      \fi
      \edef\cd@StreamList{%
        \cd@StreamList\expandafter\the\csname #1@Stream\endcsname,}
  \fi\fi\fi}
```

`\cd@StreamList` `\cd@BuildList` The last thing to do is to build that list of streams made available by the closing of a file.

```
\def\cd@StreamList{}
\def\cd@BuildList#1cd@end,{\def\cd@StreamList{#1}}
```

`\cd@FindStream` When we look for a stream, we simply check the content of `\cd@BuildList`, and if we find the terminator, this means that no stream has been made available. Otherwise, we define `\cd@ProduceStream`, which will be allocated to the file we're trying to open, as the first stream we find in the list, and we rebuild the latter with the remaining numbers.

```
\newif\ifcd@stream
\def\cd@FindStream#1,{%
  \def\cd@TempArg{#1}
  \ifx\cd@TempArg\cd@end
    \cd@streamfalse
    \let\cd@@next\relax
  \else
    \cd@streamtrue
    \chardef\cd@ProduceStream=#1 %
    \let\cd@@next\cd@BuildList
  \fi\cd@@next}
```

`\cd@Header` Finally, here's the default header.

```
\catcode`\%=12\relax
\edef\cd@Header{
\noexpand\cd@ProduceFile{% This is \noexpand\FileName, produced by the CodeDoc class
^^J% with the 'produce' option on.
^^J%
^^J% To create the documentation, compile \cd@CurrentSource
^^J% without the 'produce' option.
^^J%
^^J% SOURCE: \noexpand\cd@CurrentSource
^^J% DATE: \noexpand\FileDate
^^J% VERSION: \noexpand\FileVersion
}}
\catcode`\%=14\relax
```

... and we say goodbye. The end. 


```
\makeatother
```

Index

This index was generated by the `\DescribeMacro`*-like commands. It only reports where macros are described (page numbers in normal font) and defined (page numbers in italics). In the current version,* CodeDoc *does not index macros when used in the code.*
Entries are sorted ignoring the `cd@` *and* `cd@@` *prefixes.*