A New Implementation of LaTeX's `verbatim` and `verbatim*` Environments.

Rainer Schöpf
Internet: `Schoepf@Uni-Mainz.DE`

Bernd Raichle
Stettener Str. 73
D-73732 Wäldenbronn
Federal Republic of Germany
Internet: `raichle@azu.Informatik.Uni-Stuttgart.DE`

Chris Rowley
The Open University
Parsifal College
Finchley Road
London NW3 7BG, UK
Internet: `C.A.Rowley@open.ac.uk`

2001/03/12

Abstract

This package reimplements the LaTeX `verbatim` and `verbatim*` environments. In addition it provides a `comment` environment that skips any commands or text between `\begin{comment}` and the next `\end{comment}`. It also defines the command `verbatiminput` to input a whole file verbatim.

1 Usage notes

LaTeX's `verbatim` and `verbatim*` environments have a few features that may give rise to problems. These are:

- Due to the method used to detect the closing `\end{verbatim}` (i.e. macro parameter delimiting) you cannot leave spaces between the `\end` token and `{verbatim}`.

- Since TeX has to read all the text between the `\begin{verbatim}` and the `\end{verbatim}` before it can output anything, long verbatim listings may overflow TeX's memory.

Whereas the first of these points can be considered only a minor nuisance the other one is a real limitation.

This package file contains a reimplementation of the `verbatim` and `verbatim*` environments which overcomes these restrictions. There is, however, one incompatibility between the old and the new implementations of these environments: the old version would treat text on the same line as the `\end{verbatim}` command as if it were on a line by itself.

This new version will simply ignore it.

(This is the price one has to pay for the removal of the old `verbatim` environment's size limitations.) It will, however, issue a warning message of the form

```
LaTeX warning: Characters dropped after \end{verbatim*}!
```

This is not a real problem since this text can easily be put on the next line without affecting the output.

This new implementation also solves the second problem mentioned above: it is possible to leave spaces (but *not* begin a new line) between the `\end` and the `{verbatim}` or `{verbatim*}`:

```
\begin {verbatim*}
   test
   test
\end {verbatim*}
```

Additionally we introduce a `comment` environment, with the effect that the text between `\begin{comment}` and `\end{comment}` is simply ignored, regardless of what it looks like. At first sight this seems to be quite different from the purpose of verbatim listing, but actually the implementation of these two concepts turns out to be very similar. Both rely on the fact that the text between `\begin{...}` and `\end{...}` is read by TeX without interpreting any commands or special characters. The remaining difference between `verbatim` and `comment` is only that the text is to be typeset in the first case and to be thrown away in the latter. Note that these environments cannot be nested.

`\verbatiminput` is a command with one argument that inputs a file verbatim, i.e. the command `verbatiminput{xx.yy}` has the same effect as

> `\begin{verbatim}`
> ⟨*Contents of the file $xx.yy$*⟩
> `\end{verbatim}`

This command has also a `*`-variant that prints spaces as ␣.

2   Interfaces for package writers

The `verbatim` environment of LaTeX 2$_\varepsilon$ does not offer a good interface to programmers. In contrast, this package provides a simple mechanism to implement similar features, the `comment` environment implemented here being an example of what can be done and how.

2.1   Simple examples

It is now possible to use the `verbatim` environment to define environments of your own. E.g.,

```
\newenvironment{myverbatim}%
  {\endgraf\noindent MYVERBATIM:%
    \endgraf\verbatim}%
  {\endverbatim}
```

can be used afterwards like the `verbatim` environment, i.e.

```
\begin {myverbatim}
  test
  test
\end {myverbatim}
```

Another way to use it is to write

```
\let\foo=\comment
\let\endfoo=\endcomment
```

and from that point on environment `foo` is the same as the comment environment, i.e. everything inside its body is ignored.

You may also add special commands after the `\verbatim` macro is invoked, e.g.

```
\newenvironment{myverbatim}%
  {\verbatim\myspecialverbatimsetup}%
  {\endverbatim}
```

though you may want to learn about the hook `\every@verbatim` at this point. However, there are still a number of restrictions:

1. You must not use the `\begin` or the `\end` command inside a definition, e.g. the following two examples will *not* work:

   \newenvironment{myverbatim}%
   {\endgraf\noindent␣MYVERBATIM:%
   ␣\endgraf\begin{verbatim}}%
   {\end{verbatim}}
   \newenvironment{fred}
   {\begin{minipage}{30mm}\verbatim}
   {\endverbatim\end{minipage}}

   If you try these examples, TeX will report a "runaway argument" error. More generally, it is not possible to use `\begin…\end` or the related environments in the definition of the new environment. Instead, the correct way to define this environment would be

   \newenvironment{fred}
   {\minipage{30mm}\verbatim}
   {\endverbatim\endminipage}

2. You can*not* use the `verbatim` environment inside user defined *commands*; e.g.,

   \newcommand{\verbatimfile}[1]%
   ␣␣␣␣␣␣␣␣␣␣␣␣{\begin{verbatim}\input{#1}\end{verbatim}}

   does *not* work; nor does

   ```
   \newcommand{\verbatimfile}[1]{\verbatim\input{#1}\endverbatim}
   ```

3. The name of the newly defined environment must not contain characters with category code other than 11 (letter) or 12 (other), or this will not work.

2.2 The interfaces

Let us start with the simple things. Sometimes it may be necessary to use a special typeface for your verbatim text, or perhaps the usual computer modern typewriter shape in a reduced size.

You may select this by redefining the macro `\verbatim@font`. This macro is executed at the beginning of every verbatim text to select the font shape. Do not use it for other purposes; if you find yourself abusing this you may want to read about the `\every@verbatim` hook below.

By default, `\verbatim@font` switches to the typewriter font and disables the ligatures contained therein.

There is a hook (i.e. a token register) called `\every@verbatim` whose contents are inserted into TeX's mouth just before every verbatim text. Please use the

`\addto@hook` macro to add something to this hook. It is used as follows:

`\addto@hook`⟨*name of the hook*⟩`{`⟨*commands to be added*⟩`}`

After all specific setup, like switching of category codes, has been done, the `\verbatim@start` macro is called. This starts the main loop of the scanning mechanism implemented here. Any other environment that wants to make use of this feature should execute this macro as its last action.

These are the things that concern the start of a verbatim environment. Once this (and other) setup has been done, the code in this package reads and processes characters from the input stream in the following way:

1. Before the first character of an input line is read, it executes the macro `\verbatim@startline`.

2. After some characters have been read, the macro `\verbatim@addtoline` is called with these characters as its only argument. This may happen several times per line (when an `\end` command is present on the line in question).

3. When the end of the line is reached, the macro `\verbatim@processline` is called to process the characters that `\verbatim@addtoline` has accumulated.

4. Finally, there is the macro `\verbatim@finish` that is called just before the environment is ended by a call to the `\end` macro.

To make this clear let us consider the standard `verbatim` environment. In this case the three macros above are defined as follows:

1. `\verbatim@startline` clears the character buffer (a token register).

2. `\verbatim@addtoline` adds its argument to the character buffer.

3. `\verbatim@processline` typesets the characters accumulated in the buffer.

With this it is very simple to implement the `comment` environment: in this case `\verbatim@startline` and `\verbatim@processline` are defined to be no-ops whereas `\verbatim@addtoline` discards its argument.

Let's use this to define a variant of the `verbatim` environment that prints line numbers in the left margin. Assume that this would be done by a counter called `VerbatimLineNo`. Assuming that this counter was initialized properly by the environment, `\verbatim@processline` would be defined in this case as

```
\def\verbatim@processline{%
  \addtocounter{VerbatimLineNo}{1}%
  \leavevmode
  \llap{\theVerbatimLineNo\ \hskip\@totalleftmargin}%
  \the\verbatim@line\par}
```

A further possibility is to define a variant of the `verbatim` environment that boxes and centers the whole verbatim text. Note that the boxed text should be less than a page otherwise you have to change this example.

```
\def\verbatimboxed#1{\begingroup
  \def\verbatim@processline{%
    {\setbox0=\hbox{\the\verbatim@line}%
     \hsize=\wd0
     \the\verbatim@line\par}}%
  \setbox0=\vbox{\parskip=0pt\topsep=0pt\partopsep=0pt
                 \verbatiminput{#1}}%
  \begin{center}\fbox{\box0}\end{center}%
 \endgroup}
```

As a final nontrivial example we describe the definition of an environment called `verbatimwrite`. It writes all text in its body to a file whose name is given as an argument. We assume that a stream number called `\verbatim@out` has already been reserved by means of the `\newwrite` macro.

Let's begin with the definition of the macro `\verbatimwrite`.

```
\def\verbatimwrite#1{%
```

First we call `\@bsphack` so that this environment does not influence the spacing. Then we open the file and set the category codes of all special characters:

```
\@bsphack
\immediate\openout \verbatim@out #1
\let\do\@makeother\dospecials
\catcode`\^^M\active
```

The default definitions of the macros

```
\verbatim@startline
\verbatim@addtoline
\verbatim@finish
```

are also used in this environment. Only the macro `\verbatim@processline` has to be changed before `\verbatim@start` is called:

```
\def\verbatim@processline{%
  \immediate\write\verbatim@out{\the\verbatim@line}}%
\verbatim@start}
```

The definition of `\endverbatimwrite` is very simple: we close the stream and call `\@esphack` to get the spacing right.

```
\def\endverbatimwrite{\immediate\closeout\verbatim@out\@esphack}
```

3 The implementation

The very first thing we do is to ensure that this file is not read in twice. To this end we check whether the macro `\verbatim@@@` is defined. If so, we just stop reading this file. The 'package' guard here allows most of the code to be excluded when extracting the driver file for testing this package.

```
1 ⟨*package⟩
2 \NeedsTeXFormat{LaTeX2e}
3 \ProvidesPackage{verbatim}
4     [2003/08/22 v1.5q LaTeX2e package for verbatim enhancements]
5 \@ifundefined{verbatim@@@}{}{\endinput}
```

We use a mechanism similar to the one implemented for the `\comment`…`\endcomment` macro in $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-TeX: We input one line at a time and check if it contains the `\end{...}` tokens. Then we can decide whether we have reached the end of the verbatim text, or must continue.

3.1 Preliminaries

`\every@verbatim` The hook (i.e. token register) `\every@verbatim` is initialized to ⟨*empty*⟩.

```
6 \newtoks\every@verbatim
7 \every@verbatim={}
```

`\@makeother` `\@makeother` takes as argument a character and changes its category code to 12 (other).

```
8 \def\@makeother#1{\catcode`#112\relax}
```

\@vobeyspaces The macro `\@vobeyspaces` causes spaces in the input to be printed as spaces in the output.

```
9 \begingroup
10 \catcode`\ =\active%
11 \def\x{\def\@vobeyspaces{\catcode`\ \active\let \@xobeysp}}
12 \expandafter\endgroup\x
```

\@xobeysp The macro `\@xobeysp` produces exactly one space in the output, protected against breaking just before it. (`\@M` is an abbreviation for the number 10000.)

```
13 \def\@xobeysp{\leavevmode\penalty\@M\ }
```

\verbatim@line We use a newly defined token register called `\verbatim@line` that will be used as the character buffer.

```
14 \newtoks\verbatim@line
```

The following four macros are defined globally in a way suitable for the `verbatim` and `verbatim*` environments.

\verbatim@startline **\verbatim@addtoline** **\verbatim@processline** `\verbatim@startline` initializes processing of a line by emptying the character buffer (`\verbatim@line`).

```
15 \def\verbatim@startline{\verbatim@line{}}
```

`\verbatim@addtoline` adds the tokens in its argument to our buffer register `\verbatim@line` without expanding them.

```
16 \def\verbatim@addtoline#1{%
17   \verbatim@line\expandafter{\the\verbatim@line#1}}
```

Processing a line inside a `verbatim` or `verbatim*` environment means printing it. Ending the line means that we have to begin a new paragraph. We use `\par` for this purpose. Note that `\par` is redefined in `\@verbatim` to force TeX into horizontal mode and to insert an empty box so that empty lines in the input do appear in the output.

```
18 \def\verbatim@processline{\the\verbatim@line\par}
```

\verbatim@finish As a default, `\verbatim@finish` processes the remaining characters. When this macro is called we are facing the following problem: when the `\end{verbatim}` command is encountered `\verbatim@processline` is called to process the characters preceding the command on the same line. If there are none, an empty line would be output if we did not check for this case.

If the line is empty `\the\verbatim@line` expands to nothing. To test this we use a trick similar to that on p. 376 of the TeXbook, but with `$…$` instead of the `!` tokens. These `$` tokens can never have the same category code as a `$` token that might possibly appear in the token register `\verbatim@line`, as such a token will always have been read with category code 12 (other). Note that `\ifcat` expands the following tokens so that `\the\verbatim@line` is replaced by the accumulated characters

```
19 \def\verbatim@finish{\ifcat$\the\verbatim@line$\else
20   \verbatim@processline\fi}
```

3.2 The `verbatim` and `verbatim*` environments

\verbatim@font We start by defining the macro `\verbatim@font` that is to select the font and to set font-dependent parameters. Then we expand `\@noligs` (defined in the LaTeX 2_ε kernel). Among possibly other things, it will go through `\verbatim@nolig@list` to avoid certain ligatures. `\verbatim@nolig@list` is a macro defined in the LaTeX 2_ε kernel to expand to

```
\do\`\do\<\do\>\do\,\do\'\do\-
```

All the characters in this list can be part of a ligature in some font or other.

```
21 \def\verbatim@font{\normalfont\ttfamily
22                    \hyphenchar\font\m@ne
23                    \@noligs}
```

\@verbatim The macro \@verbatim sets up things properly. First of all, the tokens of the \every@verbatim hook are inserted. Then a trivlist environment is started and its first \item command inserted. Each line of the verbatim or verbatim* environment will be treated as a separate paragraph.

```
24 \def\@verbatim{\the\every@verbatim
25  \trivlist \item \relax
```

The following extra vertical space is for compatibility with the LaTeX kernel: otherwise, using the verbatim package changes the vertical spacing of a verbatim environment nested within a quote environment.

```
26   \if@minipage\else\vskip\parskip\fi
```

The paragraph parameters are set appropriately: the penalty at the beginning of the environment, left and right margins, paragraph indentation, the glue to fill the last line, and the vertical space between paragraphs. The latter space has to be zero since we do not want to add extra space between lines.

```
27   \@beginparpenalty \predisplaypenalty
28   \leftskip\@totalleftmargin\rightskip\z@
29   \parindent\z@\parfillskip\@flushglue\parskip\z@
```

There's one point to make here: the list environment uses TeX's \parshape primitive to get a special indentation for the first line of the list. If the list begins with a verbatim environment this \parshape is still in effect. Therefore we have to reset this internal parameter explicitly. We could do this by assigning 0 to \parshape. However, there is a simpler way to achieve this: we simply tell TeX to start a new paragraph. As is explained on p. 103 of the TeXbook, this resets \parshape to zero.

```
30   \@@par
```

We now ensure that \par has the correct definition, namely to force TeX into horizontal mode and to include an empty box. This is to ensure that empty lines do appear in the output. Afterwards, we insert the \interlinepenalty since TeX does not add a penalty between paragraphs (here: lines) by its own initiative. Otherwise a verbatim environment could be broken across pages even if a \samepage declaration were present.

However, in a top-aligned minipage, this will result in an extra empty line added at the top. Therefore, a slightly more complicated construct is necessary. One of the important things here is the inclusion of \leavevmode as the first macro in the first line, for example, a blank verbatim line is the first thing in a list item.

```
31   \def\par{%
32    \if@tempswa
33     \leavevmode\null\@@par\penalty\interlinepenalty
34    \else
35     \@tempswatrue
36     \ifhmode\@@par\penalty\interlinepenalty\fi
37    \fi}%
```

But to avoid an error message when the environment doesn't contain any text, we redefine \@noitemerr which will in this case be called by \endtrivlist.

```
38   \def\@noitemerr{\@warning{No verbatim text}}%
```

Now we call \obeylines to make the end of line character active,

```
39   \obeylines
```

change the category code of all special characters, to 12 (other).

```
40   \let\do\@makeother \dospecials
```

and switch to the font to be used.

```
41    \verbatim@font
```

To avoid a breakpoint after the labels box, we remove the penalty put there by the list macros: another use of \unpenalty!

```
42    \everypar \expandafter{\the\everypar \unpenalty}}
```

\verbatim Now we define the toplevel macros. \verbatim is slightly changed: after setting
\verbatim* up things properly it calls \verbatim@start. This is done inside a group, so that
\verbatim can be used directly, without \begin.

```
43 \def\verbatim{\begingroup\@verbatim \frenchspacing\@vobeyspaces
44                 \verbatim@start}
```

\verbatim* is defined accordingly.

```
45 \@namedef{verbatim*}{\begingroup\@verbatim\verbatim@start}
```

\endverbatim   To end the verbatim and verbatim* environments it is only necessary to finish
\endverbatim* the trivlist environment started in \@verbatim and close the corresponding group.

```
46 \def\endverbatim{\endtrivlist\endgroup\@doendpe}
47 \expandafter\let\csname endverbatim*\endcsname =\endverbatim
```

3.3 The `comment` environment

\comment The \comment macro is similar to \verbatim*. However, we do not need to switch
\endcomment fonts or set special formatting parameters such as \parindent or \parskip. We need only set the category code of all special characters to 12 (other) and that of ^^M (the end of line character) to 13 (active). The latter is needed for macro parameter delimiter matching in the internal macros defined below. In contrast to the default definitions used by the \verbatim and \verbatim* macros, we define \verbatim@addtoline to throw away its argument and \verbatim@processline, \verbatim@startline, and \verbatim@finish to act as no-ops. Then we call \verbatim@. But the first thing we do is to call \@bsphack so that this environment has no influence whatsoever upon the spacing.

```
48 \def\comment{\@bsphack
49            \let\do\@makeother\dospecials\catcode`\^^M\active
50            \let\verbatim@startline\relax
51            \let\verbatim@addtoline\@gobble
52            \let\verbatim@processline\relax
53            \let\verbatim@finish\relax
54            \verbatim@}
```

\endcomment is very simple: it only calls \@esphack to take care of the spacing. The \end macro closes the group and therefore takes care of restoring everything we changed.

```
55 \let\endcomment=\@esphack
```

3.4 The main loop

Here comes the tricky part: During the definition of the macros we need to use the special characters \, {, and } not only with their normal category codes, but also with category code 12 (other). We achieve this by the following trick: first we tell TeX that \, {, and } are the lowercase versions of !, [, and]. Then we replace every occurrence of \, {, and } that should be read with category code 12 by !, [, and], respectively, and give the whole list of tokens to \lowercase, knowing that category codes are not altered by this primitive!

But first we have ensure that !, [, and] themselves have the correct category code! To allow special settings of these codes we hide their setting in the macro \vrb@catcodes. If it is already defined our new definition is skipped.

```
56 \@ifundefined{vrb@catcodes}%
57   {\def\vrb@catcodes{%
58     \catcode`\!12\catcode`\[12\catcode`\]12}}{}
```

This trick allows us to use this code for applications where other category codes are in effect.

We start a group to keep the category code changes local.

```
59 \begingroup
60 \vrb@catcodes
61 \lccode`\!=`\\ \lccode`\[=`\{ \lccode`\]=`\}
```

We also need the end-of-line character ^^M, as an active character. If we were to simply write \catcode`\^^M=\active then we would get an unwanted active end of line character at the end of every line of the following macro definitions. Therefore we use the same trick as above: we write a tilde ~ instead of ^^M and pretend that the latter is the lowercase variant of the former. Thus we have to ensure now that the tilde character has category code 13 (active).

```
62 \catcode`\~=\active \lccode`\~=`\^^M
```

The use of the \lowercase primitive leads to one problem: the uppercase character 'C' needs to be used in the code below and its case must be preserved. So we add the command:

```
63 \lccode`\C=`\C
```

Now we start the token list passed to \lowercase. We use the following little trick (proposed by Bernd Raichle): The very first token in the token list we give to \lowercase is the \endgroup primitive. This means that it is processed by TeX immediately after \lowercase has finished its operation, thus ending the group started by \begingroup above. This avoids the global definition of all macros.

```
64 \lowercase{\endgroup
```

\verbatim@start The purpose of \verbatim@start is to check whether there are any characters on the same line as the \begin{verbatim} and to pretend that they were on a line by themselves. On the other hand, if there are no characters remaining on the current line we shall just find an end of line character. \verbatim@start performs its task by first grabbing the following character (its argument). This argument is then compared to an active ^^M, the end of line character.

```
65   \def\verbatim@start#1{%
66     \verbatim@startline
67     \if\noexpand#1\noexpand~%
```

If this is true we transfer control to \verbatim@ to process the next line. We use \next as the macro which will continue the work.

```
68       \let\next\verbatim@
```

Otherwise, we define \next to expand to a call to \verbatim@ followed by the character just read so that it is reinserted into the text. This means that those characters remaining on this line are handled as if they formed a line by themselves.

```
69     \else \def\next{\verbatim@#1}\fi
```

Finally we call \next.

```
70       \next}%
```

\verbatim@ The three macros \verbatim@, \verbatim@@, and \verbatim@@@ form the "main loop" of the verbatim environment. The purpose of \verbatim@ is to read exactly one line of input. \verbatim@@ and \verbatim@@@ work together to find out whether the four characters \end (all with category code 12 (other)) occur in that line. If so, \verbatim@@@ will call \verbatim@test to check whether this \end is part of \end{verbatim} and will terminate the environment if this is the case. Otherwise we continue as if nothing had happened. So let's have a look at the definition of \verbatim@:

```
71     \def\verbatim@#1~{\verbatim@@#1!end\@nil}%
```

Note that the ! character will have been replaced by a \ with category code 12 (other) by the \lowercase primitive governing this code before the definition of this macro actually takes place. That means that it takes the line, puts \end (four character tokens) and \@nil (one control sequence token) as a delimiter behind it, and then calls \verbatim@@.

\verbatim@@ \verbatim@@ takes everything up to the next occurrence of the four characters \end as its argument.

```
72        \def\verbatim@@#1!end{%
```

That means: if they do not occur in the original line, then argument #1 is the whole input line, and \@nil is the next token to be processed. However, if the four characters \end are part of the original line, then #1 consists of the characters in front of \end, and the next token is the following character (always remember that the line was lengthened by five tokens). Whatever #1 may be, it is verbatim text, so #1 is added to the line currently built.

```
73          \verbatim@addtoline{#1}%
```

The next token in the input stream is of special interest to us. Therefore \futurelet defines \next to be equal to it before calling \verbatim@@@.

```
74          \futurelet\next\verbatim@@@}%
```

\verbatim@@@ \verbatim@@@ will now read the rest of the tokens on the current line, up to the final \@nil token.

```
75        \def\verbatim@@@#1\@nil{%
```

If the first of the above two cases occurred, i.e. no \end characters were on that line, #1 is empty and \next is equal to \@nil. This is easily checked.

```
76          \ifx\next\@nil
```

If so, this was a simple line. We finish it by processing the line we accumulated so far. Then we prepare to read the next line.

```
77            \verbatim@processline
78            \verbatim@startline
79            \let\next\verbatim@
```

Otherwise we have to check what follows these \end tokens.

```
80          \else
```

Before we continue, it's a good idea to stop for a moment and remember where we are: We have just read the four character tokens \end and must now check whether the name of the environment (surrounded by braces) follows. To this end we define a macro called \@tempa that reads exactly one character and decides what to do next. This macro should do the following: skip spaces until it encounters either a left brace or the end of the line. But it is important to remember which characters are skipped. The \end⟨*optional spaces*⟩{ characters may be part of the verbatim text, i.e. these characters must be printed.

Assume for example that the current line contains

␣␣␣␣␣\end␣{AVeryLongEnvironmentName}

As we shall soon see, the scanning mechanism implemented here will not find out that this is text to be printed until it has read the right brace. Therefore we need a way to accumulate the characters read so that we can reinsert them if necessary. The token register \@temptokena is used for this purpose.

Before we do this we have to get rid of the superfluous \end tokens at the end of the line. To this end we define a temporary macro whose argument is delimited by \end\@nil (four character tokens and one control sequence token) to be used below on the rest of the line, after appending a \@nil token to it. (Note that this token can never appear in #1.) We use the following definition of \@tempa to get the rest of the line (after the first \end).

```
81          \def\@tempa##1!end\@nil{##1}%
```

We mentioned already that we use token register `\@temptokena` to remember the characters we skip, in case we need them again. We initialize this with the `\end` we have thrown away in the call to `\@tempa`.

```
82            \@temptokena{!end}%
```

We shall now call `\verbatim@test` to process the characters remaining on the current line. But wait a moment: we cannot simply call this macro since we have already read the whole line. Therefore we have to first expand the macro `\@tempa` to insert them again after the `\verbatim@test` token. A `^^M` character is appended to denote the end of the line. (Remember that this character comes disguised as a tilde.)

```
83            \def\next{\expandafter\verbatim@test\@tempa#1\@nil~}%
```

That's almost all, but we still have to now call `\next` to do the work.

```
84            \fi \next}%
```

`\verbatim@test` We define `\verbatim@test` to investigate every token in turn.

```
85        \def\verbatim@test#1{%
```

First of all we set `\next` equal to `\verbatim@test` in case this macro must call itself recursively in order to skip spaces.

```
86            \let\next\verbatim@test
```

We have to distinguish four cases:

1. The next token is a `^^M`, i.e. we reached the end of the line. That means that nothing special was found. Note that we use `\if` for the following comparisons so that the category code of the characters is irrelevant.

```
87                \if\noexpand#1\noexpand~%
```

   We add the characters accumulated in token register `\@temptokena` to the current line. Since `\verbatim@addtoline` does not expand its argument, we have to do the expansion at this point. Then we `\let` `\next` equal to `\verbatim@` to prepare to read the next line.

```
88                \expandafter\verbatim@addtoline
89                  \expandafter{\the\@temptokena}%
90                \verbatim@processsline
91                \verbatim@startline
92                \let\next\verbatim@
```

2. A space character follows. This is allowed, so we add it to `\@temptokena` and continue.

```
93                \else \if\noexpand#1
94                  \@temptokena\expandafter{\the\@temptokena#1}%
```

3. An open brace follows. This is the most interesting case. We must now collect characters until we read the closing brace and check whether they form the environment name. This will be done by `\verbatim@testend`, so here we let `\next` equal this macro. Again we will process the rest of the line, character by character. The characters forming the name of the environment will be accumulated in `\@tempc`. We initialize this macro to expand to nothing.

```
95                \else \if\noexpand#1\noexpand[%
96                  \let\@tempc\@empty
97                  \let\next\verbatim@testend
```

   Note that the [character will be a { when this macro is defined.

4. Any other character means that the \end was part of the verbatim text. Add the characters to the current line and prepare to call \verbatim@ to process the rest of the line.

```
98              \else
99                \expandafter\verbatim@addtoline
100                 \expandafter{\the\@temptokena}%
101                \def\next{\verbatim@#1}%
102              \fi\fi\fi
```

The last thing this macro does is to call \next to continue processing.

```
103              \next}%
```

\verbatim@testend \verbatim@testend is called when \end⟨*optional spaces*⟩{ was seen. Its task is to scan everything up to the next } and to call \verbatim@@testend. If no } is found it must reinsert the characters it read and return to \verbatim@. The following definition is similar to that of \verbatim@test: it takes the next character and decides what to do.

```
104      \def\verbatim@testend#1{%
```

Again, we have four cases:

1. ^^M: As no } is found in the current line, add the characters to the buffer. To avoid a complicated construction for expanding \@temptokena and \@tempc we do it in two steps. Then we continue with \verbatim@ to process the next line.

```
105              \if\noexpand#1\noexpand~%
106                \expandafter\verbatim@addtoline
107                 \expandafter{\the\@temptokena[}%
108                \expandafter\verbatim@addtoline
109                 \expandafter{\@tempc}%
110              \verbatim@processline
111              \verbatim@startline
112              \let\next\verbatim@
```

2. }: Call \verbatim@@testend to check if this is the right environment name.

```
113              \else\if\noexpand#1\noexpand]%
114                \let\next\verbatim@@testend
```

3. \: This character must not occur in the name of an environment. Thus we stop collecting characters. In principle, the same argument would apply to other characters as well, e.g., {. However, \ is a special case, since it may be the first character of \end. This means that we have to look again for \end{⟨*environment name*⟩}. Note that we prefixed the ! by a \noexpand primitive, to protect ourselves against it being an active character.

```
115              \else\if\noexpand#1\noexpand!%
116                \expandafter\verbatim@addtoline
117                 \expandafter{\the\@temptokena[}%
118                \expandafter\verbatim@addtoline
119                 \expandafter{\@tempc}%
120              \def\next{\verbatim@!}%
```

4. Any other character: collect it and continue. We cannot use \edef to define \@tempc since its replacement text might contain active character tokens.

```
121              \else \expandafter\def\expandafter\@tempc\expandafter
122                {\@tempc#1}\fi\fi\fi
```

As before, the macro ends by calling itself, to process the next character if appropriate.

```
123              \next}%
```

\verbatim@@testend Unlike the previous macros `\verbatim@@testend` is simple: it has only to check if the `\end{…}` matches the corresponding `\begin{…}`.

```
124        \def\verbatim@@testend{%
```

We use `\next` again to define the things that are to be done. Remember that the name of the current environment is held in `\@currenvir`, the characters accumulated by `\verbatim@testend` are in `\@tempc`. So we simply compare these and prepare to execute `\end{⟨current environment⟩}` macro if they match. Before we do this we call `\verbatim@finish` to process the last line. We define `\next` via `\edef` so that `\@currenvir` is replaced by its expansion. Therefore we need `\noexpand` to inhibit the expansion of `\end` at this point.

```
125            \ifx\@tempc\@currenvir
126                \verbatim@finish
127                \edef\next{\noexpand\end{\@currenvir}%
```

Without this trick the `\end` command would not be able to correctly check whether its argument matches the name of the current environment and you'd get an interesting LATEX error message such as:

```
! \begin{verbatim*} ended by \end{verbatim*}.
```

But what do we do with the rest of the characters, those that remain on that line? We call `\verbatim@rescan` to take care of that. Its first argument is the name of the environment just ended, in case we need it again. `\verbatim@rescan` takes the list of characters to be reprocessed as its second argument. (This token list was inserted after the current macro by `\verbatim@@@`.) Since we are still in an `\edef` we protect it by means of `\noexpand`.

```
128                    \noexpand\verbatim@rescan{\@currenvir}}%
```

If the names do not match, we reinsert everything read up to now and prepare to call `\verbatim@` to process the rest of the line.

```
129        \else
130            \expandafter\verbatim@addtoline
131                \expandafter{\the\@temptokena[}%
132                \expandafter\verbatim@addtoline
133                    \expandafter{\@tempc]}%
134            \let\next\verbatim@
135        \fi
```

Finally we call `\next`.

```
136        \next}%
```

\verbatim@rescan In principle `\verbatim@rescan` could be used to analyse the characters remaining after the `\end{...}` command and pretend that these were read "properly", assuming "standard" category codes are in force.[1] But this is not always possible (when there are unmatched curly braces in the rest of the line). Besides, we think that this is not worth the effort: After a `verbatim` or `verbatim*` environment a new line in the output is begun anyway, and an `\end{comment}` can easily be put on a line by itself. So there is no reason why there should be any text here. For the benefit of the user who did put something there (a comment, perhaps) we simply issue a warning and drop them. The method of testing is explained in Appendix D, p. 376 of the TEXbook. We use `^^M` instead of the `!` character used there since this is a character that cannot appear in `#1`. The two `\noexpand` primitives are necessary to avoid expansion of active characters and macros.

One extra subtlety should be noted here: remember that the token list we are currently building will first be processed by the `\lowercase` primitive before TEX carries out the definitions. This means that the 'C' character in the argument to the `\@warning` macro must be protected against being changed to 'c'. That's the reason why we added the `\lccode`\C=`\C` assignment above. We can now finish

---

[1] Remember that they were all read with category codes 11 (letter) and 12 (other) so that control sequences are not recognized as such.

the argument to `\lowercase` as well as the group in which the category codes were changed.

```
137    \def\verbatim@rescan#1#2~{\if\noexpand~\noexpand#2~\else
138        \@warning{Characters dropped after `\string\end{#1}'}\fi}}
```

3.5 The `\verbatiminput` command

`\verbatim@in@stream` We begin by allocating an input stream (out of the 16 available input streams).

```
139 \newread\verbatim@in@stream
```

`\verbatim@readfile` The macro `\verbatim@readfile` encloses the main loop by calls to the macros `\verbatim@startline` and `\verbatim@finish`, respectively. This makes sure that the user can initialize and finish the command when the file is empty or doesn't exist. The `verbatim` environment has a similar behaviour when called with an empty text.

```
140 \def\verbatim@readfile#1{%
141    \verbatim@startline
```

When the file is not found we issue a warning.

```
142    \openin\verbatim@in@stream #1\relax
143    \ifeof\verbatim@in@stream
144        \typeout{No file #1.}%
145    \else
```

At this point we pass the name of the file to `\@addtofilelist` so that its appears appears in the output of a `\listfiles` command. In addition, we use `\ProvidesFile` to make a log entry in the transcript file and to distinguish files read in via `\verbatiminput` from others.

```
146        \@addtofilelist{#1}%
147        \ProvidesFile{#1}[(verbatim)]%
```

While reading from the file it is useful to switch off the recognition of the end-of-line character. This saves us stripping off spaces from the contents of the line.

```
148        \expandafter\endlinechar\expandafter\m@ne
149        \expandafter\verbatim@read@file
150        \expandafter\endlinechar\the\endlinechar\relax
151        \closein\verbatim@in@stream
152    \fi
153    \verbatim@finish
154 }
```

`\verbatim@read@file` All the work is done in `\verbatim@read@file`. It reads the input file line by line and recursively calls itself until the end of the file.

```
155 \def\verbatim@read@file{%
156    \read\verbatim@in@stream to\next
157    \ifeof\verbatim@in@stream
158    \else
```

For each line we call `\verbatim@addtoline` with the contents of the line. `\verbatim@processline` is called next.

```
159        \expandafter\verbatim@addtoline\expandafter{\next}%
160        \verbatim@processline
```

After processing the line we call `\verbatim@startline` to initialize all before we read the next line.

```
161        \verbatim@startline
```

Without `\expandafter` each call of `\verbatim@read@file` uses space in TeX's input stack.[2]

---

[2] A standard TeX would report an overflow error if you try to read a file with more than ca. 200 lines. The same error occurs if the first line of code in §390 of *"TeX: The Program"* is missing.

```
162     \expandafter\verbatim@read@file
163   \fi
164 }
```

\verbatiminput \verbatiminput first starts a group to keep font and category changes local. Then it calls the macro \verbatim@input with additional arguments, depending on whether an asterisk follows.

```
165 \def\verbatiminput{\begingroup
166   \@ifstar{\verbatim@input\relax}%
167           {\verbatim@input{\frenchspacing\@vobeyspaces}}}
```

\verbatim@input \verbatim@input first checks whether the file exists, using the standard macro \IfFileExists which leaves the name of the file found in \@filef@und. Then everything is set up as in the \verbatim macro.

```
168 \def\verbatim@input#1#2{%
169   \IfFileExists {#2}{\@verbatim #1\relax
```

Then it reads in the file, finishes off the `trivlist` environment started by \@verbatim and closes the group. This restores everything to its normal settings.

```
170     \verbatim@readfile{\@filef@und}\endtrivlist\endgroup\@doendpe}%
```

If the file is not found a more or less helpful message is printed. The final \endgroup is needed to close the group started in \verbatiminput above.

```
171     {\typeout {No file #2.}\endgroup}}
172 ⟨/package⟩
```

3.6   Getting verbatim text into arguments.

One way of achieving this is to define a macro (command) whose expansion is the required verbatim text. This command can then be used anywhere that the verbatim text is required. It can be used in arguments, even moving ones, but it is fragile (at least, the version here is).

Here is some code which claims to provide this. It is a much revised version of something I (Chris) did about 2 years ago. Maybe it needs further revision.

It is only intended as an extension to \verb, not to the `verbatim` environment. It should therefore, perhaps, treat line-ends similarly to whatever is best for \verb.

\newverbtext This is the command to produce a new macro whose expansion is verbatim text. This command itself cannot be used in arguments, of course! It is used as follows:

```
\newverbtext{\myverb}"^%{ &~_\}}@ #"
```

The rules for delimiting the verbatim text are the same as those for \verb.

```
173 ⟨*verbtext⟩
174 \def \newverbtext {%
175   \@ifstar {\@tempswatrue \@verbtext }{\@tempswafalse \@verbtext *}%
176 }
```

I think that a temporary switch is safe here: if not, then suitable \lets can be used.

```
177 \def \@verbtext *#1#2{%
178   \begingroup
179     \let\do\@makeother \dospecials
180     \let\do\do@noligs \verbatim@nolig@list
181     \@vobeyspaces
182     \catcode`#2\active
183     \catcode`~\active
184     \lccode`\~`#2%
185     \lowercase
```

We use a temporary macro here and a trick so that the definition of the command itself can be done inside the group and be a local definition (there may be better ways to achieve this).

```
186        {\def \@tempa ##1~%
187            {\whitespaces
```

If these `\noexpand`s were `\noexpand\protect\noexpand`, would this make these things robust?

```
188            \edef #1{\noexpand \@verbtextmcheck
189                \bgroup
190                \if@tempswa
191                  \noexpand \visiblespaces
192                \fi
193                \noexpand \@verbtextsetup
194                ##1%
195                \egroup}%
196            }%
197        \expandafter\endgroup\@tempa
198      }
199 }
```

This sets up the correct type of group for the mode: it must not be expanded at define time!

```
200 \def \@verbtextmcheck {%
201    \relax\ifmmode
202            \hbox
203        \else
204            \leavevmode
205            \null
206        \fi
207 }
```

This contains other things which should not be expanded during the definition.

```
208 \def \@verbtextsetup {%
209    \frenchspacing
210    \verbatim@font
211    \verbtextstyle
212 }
```

The command `\verbtextstyle` is a document-level hook which can be used to override the predefined typographic treatment of commands defined with `\newverbtext` commands.

`\visiblespaces` and `\whitespaces` are examples of possible values of this hook.

```
213 \let \verbtextstyle \relax
214 \def \visiblespaces {\chardef \  32\relax}
215 \def \whitespaces {\let \ \@@space}
216 \let \@@space \ %
217 ⟨/verbtext⟩
```

4 Testing the implementation

For testing the implementation and for demonstration we provide an extra file. It can be extracted by using the conditional 'testdriver'. It uses a small input file called 'verbtest.tst' that is distributed separately. Again, we use individual '+' guards.

```
218 ⟨*testdriver⟩
219 \documentclass{article}
220
221 \usepackage{verbatim}
222
```

```
223 \newenvironment{myverbatim}%
224    {\endgraf\noindent MYVERBATIM:\endgraf\verbatim}%
225    {\endverbatim}
226
227 \makeatletter
228
229 \newenvironment{verbatimlisting}[1]%
230 {\def\verbatim@startline{\input{#1}%
231    \def\verbatim@startline{\verbatim@line{}}%
232    \verbatim@startline}%
233  \verbatim}{\endverbatim}
234
235 \newwrite\verbatim@out
236
237 \newenvironment{verbatimwrite}[1]%
238 {\@bsphack
239  \immediate\openout \verbatim@out #1
240  \let\do\@makeother\dospecials\catcode`\^^M\active
241  \def\verbatim@processline{%
242     \immediate\write\verbatim@out{\the\verbatim@line}}%
243  \verbatim@start}%
244 {\immediate\closeout\verbatim@out\@esphack}
245
246 \makeatother
247
248 \addtolength{\textwidth}{30pt}
249
250 \begin{document}
251
252 \typeout{}
253 \typeout{===> Expect ``characters dropped''
254          warning messages in this test! <====}
255 \typeout{}
256
257 Text Text Text Text Text Text Text Text Text Text Text
258 Text Text Text Text Text Text Text Text Text Text Text
259 Text Text Text Text Text Text Text Text Text Text Text
260   \begin{verbatim}
261     test
262     \end{verbatim*}
263     test
264     \end{verbatim
265     test of ligatures: <`!`?`>
266     \endverbatim
267     test
268     \end   verbatim
269     test
270     test of end of line:
271     \end
272     {verbatim}
273   \end{verbatim} Further text to be typeset: $\alpha$.
274 Text Text Text Text Text Text Text Text Text Text Text
275 Text Text Text Text Text Text Text Text Text Text Text
276 Text Text Text Text Text Text Text Text Text Text Text
277   \begin{verbatim*}
278     test
279     test
280   \end {verbatim*}
281 Text Text Text Text Text Text Text Text Text Text Text
282 Text Text Text Text Text Text Text Text Text Text Text
283 Text Text Text Text Text Text Text Text Text Text Text
284   \begin{verbatim*}  bla bla
```

```
285     test
286     test
287   \end {verbatim*}
288 Text Text Text Text Text Text Text Text Text Text Text
289 Text Text Text Text Text Text Text Text Text Text Text
290 Text Text Text Text Text Text Text Text Text Text Text
291 Text Text Text Text Text Text Text Text Text Text Text
292
293 First of Chris Rowley's fiendish tests:
294 \begin{verbatim}
295 the double end test<text>
296 \end\end{verbatim}  or even \end \end{verbatim}
297 %
298 %not \end\ended??
299 %\end{verbatim}
300
301 Another of Chris' devils:
302 \begin{verbatim}
303 the single brace test<text>
304 \end{not the end\end{verbatim}
305 %
306 %not \end}ed??
307 %\end{verbatim}
308 Back to my own tests:
309   \begin{myverbatim}
310     test
311     test
312   \end {myverbatim} rest of line
313 Text Text Text Text Text Text Text Text Text Text Text
314 Text Text Text Text Text Text Text Text Text Text Text
315 Text Text Text Text Text Text Text Text Text Text Text
316
317 Test of empty verbatim:
318 \begin{verbatim}
319 \end{verbatim}
320 Text Text Text Text Text Text Text Text Text Text Text
321 Text Text Text Text Text Text Text Text Text Text Text
322 Text Text Text Text Text Text Text Text Text Text Text
323   \begin {verbatimlisting}{verbtest.tex}
324     Additonal verbatim text
325       ...
326   \end{verbatimlisting}
327 And here for listing a file:
328   \verbatiminput{verbtest.tex}
329 And again, with explicit spaces:
330   \verbatiminput*{verbtest.tex}
331 Text Text Text Text Text Text Text Text Text Text Text
332 Text Text Text Text Text Text Text Text Text Text Text
333 Text Text Text Text Text Text Text Text Text Text Text
334   \begin{comment}
335     test
336     \end{verbatim*}
337     test
338     \end {comment
339     test
340     \endverbatim
341     test
342     \end   verbatim
343     test
344   \end {comment} Further text to be typeset: $\alpha$.
345 Text Text Text Text Text Text Text Text Text Text Text
346 Text Text Text Text Text Text Text Text Text Text Text
```

```
347 Text Text Text Text Text Text Text Text Text Text Text
348   \begin{comment}  bla bla
349     test
350     test
351   \end {comment}
352 Text Text Text Text Text Text Text Text Text Text Text
353 Text Text Text Text Text Text Text Text Text Text Text
354 Text Text Text Text Text Text Text Text Text Text Text
355 Text Text Text Text Text Text Text Text Text Text Text
356
357 \begin{verbatimwrite}{verbtest.txt}
358 asfa<fa<df
359 sdfsdfasd
360 asdfa<fsa
361 \end{verbatimwrite}
362
363 \end{document}
364 ⟨/testdriver⟩
```