A New Implementation of the LaTeX verbatim and verbatim* Environments*

Rainer Schöpf†

Abstract

This style option 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 contains a redefinition of LaTeX's `\verb` command to better detect the omission of the closing delimiter.

1 Usage notes

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

- 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.
- 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}`.

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

This style 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.**[1] 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* end of 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 these two concepts turn 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 former case and to be thrown away in the latter.

`\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 style file designers

The `verbatim` environment of LaTeX version 2.09 does not offer a good interface to programmers. In contrast, this style file provides a simple mechanism to implement similar features, the `comment` environment provided 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}
```

---

* This file has version number v1.4a dated 90/04/04. The documentation was last revised on 90/04/04.

†Many thanks to Chris Rowley from The Open University, UK, for looking this over, making a lot of useful suggestions, and discovering bugs. And many thanks to all the beta testers who tried this style file out.

[1] This is the price one has to pay for the removal of the old `verbatim` environment's size limitations.

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 `\begin{verbatim}` inside a definition, e.g.

   ```
   \newenvironment{myverbatim}%
   {\endgraf\noindent␣MYVERBATIM:%
   ␣\endgraf\begin{verbatim}}%
   {\end{verbatim}}
   ```

   If you try this example, TEX will report a "runaway argument" error. More generally, it is not possible to use `\begin{verbatim}`... `\end{verbatim}` or the related environments in the definition of the new environment.
2. You can*not* use the `verbatim` environment inside user defined *commands*; e.g.,

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

   does *not* work; nor does

   ```
   \newcommand[1]{\verbatimfile}%
     {\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.

Per default, `\verbatim@font` switches to the typewriter font and disables the ?‘ and !‘ ligatures.

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 call 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 style file reads and processes characters from the input stream in the following way:

1. Before it starts to read the first character of an input line the macro `\verbatim@startline` is called.
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 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 no-ops whereas `\verbatim@addtoline` discards its argument.

Another possibility is 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}%
  \llap{\theVerbatimLineNo
  \ \hskip\@totalleftmargin}}
```

As a final nontrivial example we describe the definition of an environment called `verbatimwrite`. It writes all text in its body to a file the name of which it 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

We use a mechanism similar to the one implemented for the `\comment...\endcomment` macro in $\mathcal{AMS}$-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.

As always we begin by identifying the latest version of this file on the VDU and in the transcript file.

```
\typeout{Style-Option: `verbatim'
   \fileversion \space <\filedate>  (RmS)}
\typeout{English Documentation
   \@spaces \@spaces \space <\docdate>  (RmS)}
```

3.1 Preliminaries

`\addto@hook` We begin by defining a macro that adds tokens to a hook. The first argument is supposed to be a token register, the second consists of arbitrary TEX text.

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

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

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

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

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

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

```
\begingroup
 \catcode`\ =\active%
\gdef\@vobeyspaces{\catcode`\ \active\let \@xobeysp}%
\endgroup
```

\@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.)

```
\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.

```
\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).

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

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

```
\def\verbatim@addtoline#1{%
  \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.

```
\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 those appearing in the token register \verbatim@line where $ characters were read with category code 12 (other). Note that \ifcat expands the following tokens so that \the\verbatim@line is replaced by the accumulated characters

```
\def\verbatim@finish{\ifcat$\the\verbatim@line$\else
  \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. For the default computer modern typewriter font (cmtt) we have to avoid the ligatures ¡ and ¿ (as produced by !‘ and ?‘). We do this by making the backquote ‘ character active and defining it to insert an explicit kern before the backquote character. While the backquote character is active we cannot use it in a construction like \catcode‘⟨*char*⟩=⟨*number*⟩. Instead we use the ASCII code of this character (96).

```
\begingroup
 \catcode`\`=\active
 \gdef\verbatim@font{\tt \catcode96\active \def`{\kern\z@\char96 }}
\endgroup
```

\@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.

```
\def\@verbatim{\the\every@verbatim
  \trivlist \item[]%
```

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

```
  \leftskip\@totalleftmargin\rightskip\z@
  \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.

```
  \@@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.

```
  \def\par{\leavevmode\null\@@par}%
```

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

```
  \obeylines
```

switch to the font to be used,

```
  \verbatim@font
```

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

```
  \let\do\@makeother \dospecials}
```

\verbatim \verbatim* Now we define the toplevel macros. \verbatim is slightly changed: after setting up things properly it calls \verbatim@start.

```
\def\verbatim{\@verbatim \frenchspacing\@vobeyspaces\verbatim@start}
```

\verbatim* is defined accordingly.

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

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

```
\let\endverbatim=\endtrivlist
\expandafter\let\csname endverbatim*\endcsname =\endtrivlist
```

3.3 The comment environment

\comment \endcomment The \comment macro is similar to \verbatim*. However, we do not need to switch 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.

```
\def\comment{\@bsphack
             \let\do\@makeother\dospecials\catcode`\^^M\active
             \let\verbatim@startline\relax
             \let\verbatim@addtoline\@gobble
             \let\verbatim@processline\relax
             \let\verbatim@finish\relax
             \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.

```
\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 `\lowercase`!

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.

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

This allows the use of this code for applications where other category codes are in effect.

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

```
\begingroup
 \vrb@catcodes
 \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).

```
 \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:

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

Now we start the token list passed to `\lowercase`.

```
 \lowercase{%
```

Since this is done in a group all macro definitions are executed globally.

\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.

```
55    \gdef\verbatim@start#1{%
56      \verbatim@startline
57      \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.

```
58        \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.

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

Finally we call \next.

```
60      \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@:

```
61    \gdef\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.

```
62    \gdef\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.

```
63      \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@@@.

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

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

```
65    \gdef\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.

```
\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.

```
  \verbatim@processline
  \verbatim@startline
  \let\next\verbatim@
```

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

```
\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) and use it on the rest of the line, after appending a \@nil token to it. This token can never appear in #1. We use the following definition of \@tempa to store the rest of the line (after the first \end) in token register \toks@ which we shall use again in a moment.

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

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.

```
  \@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. We stored its characters in token register \toks@. Therefore we use the following \edef to insert them again after the \verbatim@test token. A ^^M character is appended to denote the end of the line.

```
  \edef\next{\noexpand\verbatim@test\the\toks@\noexpand^^M}%
```

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

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

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

```
\gdef\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.

```
\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.

   ```
   \if\noexpand#1\noexpand^^M%
   ```

   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.

   ```
   \expandafter\verbatim@addtoline
     \expandafter{\the\@temptokena}%
   \verbatim@processline
   \verbatim@startline
   \let\next\verbatim@
   ```

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

   ```
   \else \if\noexpand#1
     \@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.

   ```
   \else \if\noexpand#1\noexpand[%
     \let\@tempc\@empty
     \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.

   ```
   \else
     \expandafter\verbatim@addtoline
       \expandafter{\the\@temptokena}%
     \def\next{\verbatim@#1}%
   \fi\fi\fi
   ```

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

```
\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.

```
\gdef\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.

```
\if\noexpand#1\noexpand~%
  \expandafter\verbatim@addtoline
    \expandafter{\the\@temptokena[]%
  \expandafter\verbatim@addtoline
    \expandafter{\@tempc}%
  \verbatim@processline
  \verbatim@startline
  \let\next\verbatim@
```

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

```
\else\if\noexpand#1\noexpand]%
  \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.

```
\else\if\noexpand#1\noexpand!%
  \expandafter\verbatim@addtoline
    \expandafter{\the\@temptokena[]%
  \expandafter\verbatim@addtoline
    \expandafter{\@tempc}%
  \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.

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

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

```
  \next}%
```

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

```
\gdef\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.

```
\ifx\@tempc\@currenvir
  \verbatim@finish
  \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`.

```
119                     \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.

```
120       \else
121         \expandafter\verbatim@addtoline
122           \expandafter{\the\@temptokena[}%
123           \expandafter\verbatim@addtoline
124             \expandafter{\@tempc]}%
125         \let\next\verbatim@
126       \fi
```

Finally we call `\next`.

```
127       \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.[2] 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 the argument to `\lowercase` as well as the group in which the category codes were changed.

```
128   \gdef\verbatim@rescan#1#2^^M{\if\noexpand^^M\noexpand#2^^M\else
129         \@warning{Characters dropped after `\string\end{#1}'}\fi}}
130 \endgroup
```

3.5 The `\verbatiminput` command

`\verbatiminput` `\verbatiminput` first starts a group to keep font and category changes local.

```
131 \def\verbatiminput{\begingroup
```

The right sequence of actions is crucial here. First we must check if a star follows. Then we must read the argument (the file name). Finally we must set up everything to read the contents of the file verbatim. Therefore we must not start by calling `\@verbatim` to change font and the category code of characters. Instead we call one

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

of the macros \sverbatim@input or \verbatim@input, depending on whether a star follows.

```
\@ifstar\sverbatim@input\verbatim@input}
```

\sverbatim@input

\sverbatim@input reads the file name argument and sets up everything as in the \verbatim macro. Then it reads in the file, finishes off the trivlist environment started by \@verbatim and closes the group opened in \verbatiminput. This restores everything to its normal settings.

```
\def\sverbatim@input#1{\@verbatim
  \@input{#1}\endtrivlist\endgroup\@doendpe}
```

\verbatim@input

\verbatim@input is nearly the same; it additionally calls \frenchspacing and \@vobeyspaces (as in \verbatim and \verb).

```
\def\verbatim@input#1{\@verbatim
  \frenchspacing \@vobeyspaces
  \@input{#1}\endtrivlist\endgroup\@doendpe}
```

3.6 Redefinition of the \verb command.

The implementation here has the following advantage over that in the original LaTeX: it will not accept that the end of the input line is reached before the verbatim text has ended. Instead, it will end the verbatim text and generate an error message.

\verb

We need special category codes during the definition: the end of line character (^^M) must be an active character. We do this in the same way as above:

```
\begingroup
  \lccode`\~=`\^^M
  \lowercase{%
    \gdef\verb{\begingroup
```

We use here \verbatim@font rather than switching directly to \tt.

```
      \verbatim@font
```

Now we make the end of line character active and define it to restore everything back to normal and to signal an error.

```
      \def~{\endgroup\@latexerr{\string\verb\space command ended by
                                end of line.}\@ehc}%
```

The rest is copied from latex.tex where we have replaced one macro (\@verb) by its expansion.

```
    \let\do\@makeother \dospecials
    \@ifstar\@sverb{\@vobeyspaces \frenchspacing \@sverb}}}
\endgroup
```

\@sverb

\@sverb gains control when we are ready to look for the delimiting character. It reads it and defines this character to be equivalent to the \endgroup primitive. I.e. it will restore everything to normal when it occurs for the second time. But this is not enough: if the first character of \verb's argument is a space and if a line break occurs at this point the space will still disappear. To avoid this we include an empty \hbox{} at the beginning.

```
\def\@sverb#1{%
  \catcode`#1\active
  \lccode`\~`#1%
  \lowercase{\let~\endgroup}%
  \leavevmode\null}
```

Index

The italic numbers denote the lines where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

◇ Rainer Schöpf
Institut für Theoretische Physik
der Universität Heidelberg
Philosophenweg 16
D-6900 Heidelberg
Federal Republic of Germany