CONTEXT

Verbatim

group: CONTEXT Support Macros

version: 1997.01.04

date: 1997 July 25

author: Hans Hagen

Because this module is quite independant of system macros, it can be used as a stand–alone verbatim environment.

```
1  \ifx \undefined \writestatus \input supp-mis.tex \fi
```

Verbatim typesetting, especially of TEX sources, is a non–trivial task. This is a direct results of the fact that characters can have ⟨*catcodes*⟩ other than 11 and such characters needs a special treatment. What for instance is TEX supposed to do when it encounters a `$` or an `#`? This module deals with these matters.

```
2  \writestatus{loading}{Context Support Macros / Verbatim}
```

The verbatim environment has some features, like coloring TEX text, seldom found in other environments. Especially when the output of TEX is viewed on an electronic medium, coloring has a positive influence on the readability of TEX sources, so we found it very acceptable to dedicate half of this module to typesetting TEX specific character sequences in color. In this module we'll also present some macro's for typesetting inline, display and file verbatim. The macro's are capable of handling `<tab>` too.

This module shows a few tricks that are often overseen by novice, like the use of the TEX primitive `\meaning`. First I'll show in what way the users are confronted with verbatim typesetting. Because we want to be able to test for symmetry and because we hate the method of closing down the verbatim mode with some strange active character, we will use the following construction for display verbatim:

```
\starttyping
The Dutch word 'typen' stands for 'typing', therefore in the Dutch version
one will not find the word 'verbatim'.
\stoptyping
```

In CONTEXT files can be typed with `\typefile` and inline verbatim can be accomplished with `\type`. This last command comes in many flavors:

```
We can say \type<<something>> or \type{something}. The first one is a bit
longer but also supports slanted typing, which accomplished by typing
\type<<a <<slanted>> word>>. We can also use commands to enhance the text
\type<<with <</bf boldfaced>> text>>. Just to be complete, we decided
to accept also \LaTeX\ alike verbatim, which means that \type+something+
and \type|something| are valid commands too. Of course we want the grouped
alternatives to process \type{hello {\bf big} world}} with braces.
```

In the core modules, we will build this support on top of this module. There these commands can be tuned with accompanying setup commands. There we can enable commands, slanted typing, control spaces, `<tab>`–handling and (here we are:) coloring. We can also setup surrounding white space and indenting. Here we'll only show some examples.

```
3  \unprotect
```

`\verbatimfont` When we are typesetting verbatim we use a non–proportional (mono spaced) font. Normally this font is available by calling `\tt`. In CONTEXT this command does a complete font–style switch. There we could have stuck with `\tttf`.

```
4  \ifx \undefined \verbatimfont \def\verbatimfont {\tt} \fi
```

\obeyedspace \obeyedtab \obeyedline \obeyedpage

We have followed Knuth in naming macros that make <space>, <newline> and <newpage> active and assigning them \obeysomething, but first we set some default values.

```
\def\obeyedspace {\hbox{ }}
\def\obeyedtab   {\obeyedspace}
\def\obeyedline  {\par}
\def\obeyedpage  {\vfill\eject}
```

\controlspace \setcontrols..

First we define \obeyspaces. When we want visible spaces (control spaces) we only have to adapt the definition of \obeyedspace to:

```
\def\controlspace {\hbox{\char32}}

\bgroup
\catcode`\ =\@@active
\gdef\obeyspaces{\catcode`\ =\@@active\def {\obeyedspace}}
\gdef\setcontrolspaces{\catcode`\ =\@@active\def {\controlspace}}
\egroup
```

\obeytabs \obeylines \obeypages \ignoretabs \ignorelines \ignorepages

Next we take care of <newline> and <newpage> and because we want to be able to typeset listings that contain <tab>, we have to handle those too. Because we have to redefine the <newpage> character locally, we redefine the meaning of this (often already) active character.

```
\catcode`\^^L=\@@active \def^^L{\par}

\bgroup

\catcode`\^^I=\@@active
\catcode`\^^M=\@@active
\catcode`\^^L=\@@active

\gdef\obeytabs    {\catcode`\^^I=\@@active\def^^I{\obeyedtab}}
\gdef\obeylines   {\catcode`\^^M=\@@active\def^^M{\obeyedline}}
\gdef\obeypages   {\catcode`\^^L=\@@active\def^^L{\obeyedpage}}

\gdef\ignoretabs  {\catcode`\^^I=\@@active\def^^I{\obeyedspace}}
\gdef\ignorelines {\catcode`\^^M=\@@active\def^^M{\obeyedspace}}
\gdef\ignorepages {\catcode`\^^L=\@@active\def^^L{\obeyedline}}

\egroup
```

\obeycharact..

We also predefine \obeycharacters, which will enable us to implement character–specific behavior, like colored verbatim.

```
\let\obeycharacters=\relax
```

\settabskips

The macro \settabskip can be used to enable tab handling. Processing tabs is sometimes needed when one processes a plain ASCII listing. Tab handling slows down verbatim typesetting considerably.

```
\bgroup

\catcode`\^^I=\@@active

\gdef\settabskips%
  {\let\processverbatimline=\doprocesstabskipline
   \catcode`\^^I=\@@active
   \let^^I=\doprocesstabskip}

\egroup
```

\processinli..

Although the inline verbatim commands presented here will be extended and embedded in the core modules of CONTEXT, they can be used separately. Both grouped and character alternatives are provided but << and nested braces are implemented in the core module. This commands takes one argument: the closing command.

```
\processinlineverbatim{\closingcommand}
```

One can define his own verbatim commands, which can be very simple:

```
\def\Verbatim {\processinlineverbatim\relax}
```

or a bit more more complex:

```
\def\GroupedVerbatim%
  {\bgroup
   \dosomeusefullthings
   \processinlineverbatim\egroup}
```

Before entering inline verbatim mode, we take care of the unwanted <tab>, <newline> and <newpage> characters and turn them into <space>. We need the double \bgroup construction to keep the closing command local.

19

```
\def\setupinlineverbatim%
  {\verbatimfont
   \let\obeytabs=\ignoretabs
   \let\obeylines=\ignorelines
   \let\obeypages=\ignorepages
   \setupcopyverbatim}
```

20

```
\def\doprocessinlineverbatim%
  {\ifx\next\bgroup
     \setupinlineverbatim
     \catcode`\{=\@@begingroup
     \catcode`\}=\@@endgroup
     \def\next{\let\next=}%
   \else
     \setupinlineverbatim
     \def\next##1{\catcode`##1=\@@endgroup}%
   \fi
   \next}
```

21

```
\def\processinlineverbatim#1%
  {\bgroup
   \localcatcodestrue % TeX processes paragraph's
   \def\endofverbatimcommand{#1\egroup}%
   \bgroup
   \aftergroup\endofverbatimcommand
   \futurelet\next\doprocessinlineverbatim}
```

\processdisp..

The closing command is executed afterwards as an internal command and therefore should not be given explicitly when typesetting inline verbatim.

We can define a display verbatim environment with the command \processdisplayverbatim in the following way:

```
\processdisplayverbatim{\closingcommand}
```

For instance, we can define a simple command like:

```
\def\BeginVerbatim {\processdisplayverbatim{EndVerbatim}}
```

But we can also do more advance things like:

```
\def\BeginVerbatim {\bigskip \processdisplayverbatim{\EndVerbatim}}
\def\EndVerbatim   {\bigskip}
```

When we compare these examples, we see that the backslash in the closing command is optional. One is free in actually defining a closing command. If one is defined, the command is executed after ending verbatim mode.

22
```
\def\processdisplayverbatim#1%
  {\par
   \bgroup
   \escapechar=-1
   \xdef\verbatimname{\string#1}%
   \egroup
   \def\endofdisplayverbatim{\csname\verbatimname\endcsname}%
   \bgroup
   \parindent\!!zeropoint
   \ifdim\lastskip<\parskip
     \removelastskip
     \vskip\parskip
   \fi
   \parskip\!!zeropoint
   \processingverbatimtrue
   \linepartrue
   \expandafter\let\csname\verbatimname\endcsname=\relax
   \edef\endofverbatimcommand{\csname\verbatimname\endcsname}%
   \edef\endofverbatimcommand{\meaning\endofverbatimcommand}%
   \verbatimfont
   \setupcopyverbatim
   \let\doverbatimline=\relax
   \copyverbatimline}
```

We save the closing sequence in `\endofverbatimcommand` in such a way that it can be compared on a line by line basis. For the conversion we use `\meaning`, which converts the line to non–expandable tokens. We reset `\parskip`, because we don't want inter–paragraph skips to creep into the verbatim source. Furthermore we `\relax` the line–processing macro while getting the rest of the first line. The initialization command `\setupcopyverbatim` does just what we expect it to do: it assigns all characters ⟨*catcode*⟩ 11. Next we switch to french spacing and call for obeyance.

23
```
\def\setupcopyverbatim%
  {\uncatcodecharacters
   \frenchspacing
   \obeyspaces
   \obeytabs
   \obeylines
   \obeycharacters}
```

\ifeightbitc..
\setcatcodes
\uncatcodesp..
\uncatcodech..

As its name says, `\uncatcodecharacters` resets the ⟨*catcode*⟩ of characters. When we use an upper bound of 127 or 255, depending in `\ifeightbitcharacters`. By counting down, we only have to use one counter. The macro `\setcatcodes` can be uses to set alternative values. The macro `\resetspecialcharacters` resets characters with special meanings. This macro is not used in the verbatim macros, but is best defined in this module.

24
```
\def\doprocesscatcodes#1%
  {\ifeightbitcharacters
     \scratchcounter=255
   \else
     \scratchcounter=127
   \fi
   \loop
     \savecatcode
     #1\relax
     \advance\scratchcounter by -1
     \ifnum\scratchcounter>-1
   \repeat
   \let\savecatcode=\relax
   \let\restorecatcodes=\dorestorecatcodes}
```

25
```
\def\uncatcodespecials%
  {\doprocesscatcodes
     {\ifnum\catcode\scratchcounter=\@@letter\relax\else
        \catcode\scratchcounter=\@@other
      \fi}%
   \catcode`\   =\@@space
   \catcode`\^^L=\@@ignore
   \catcode`\^^M=\@@endofline
   \catcode`\^^?=\@@ignore}
```

26
```
\def\setcatcodes#1%
  {\doprocesscatcodes
     {\catcode\scratchcounter=#1}}
```

27
```
\def\uncatcodecharacters%
  {\setcatcodes\@@letter}
```

\iflocalcatc..
\restorecatc..
\beginrestor..
\endrestorec..

We're not finished dealing ⟨*catcodes*⟩ yet. In CONTEXT we use only one auxiliary file, which deals with tables of contents, registers, two pass tracking, references etc. This file, as well as files concerning graphics, is processed when needed, which can be in the mid of typesetting verbatim. However, when reading in data in verbatim mode, we should temporary restore the normal ⟨*catcodes*⟩, and that's exactly what the next macros do. Saving the catcodes can be disabled by saying `\localcatcodestrue`.

The previous macros call for `\savecatcode`, which is implemented as:

28
```
\newif\iflocalcatcodes
```

29
```
\def\savecatcode%
  {\iflocalcatcodes \else
     \expandafter\edef\csname @@cc@@\the\scratchcounter\endcsname%
       {\the\catcode\scratchcounter}%
   \fi}
```

It's counterpart is:

```
\def\restorecatcode%
  {\expandafter\catcode\expandafter\scratchcounter\expandafter=
     \csname @@cc@@\the\scratchcounter\endcsname}
```

When we want to restore ⟨*catcodes*⟩ we call for `\restorecatcodes`, which default to `\relax`

```
\let\restorecatcodes=\relax
```

or when we've saves things calls for:

```
\def\dorestorecatcodes%
  {\iflocalcatcodes \else
     \doprocesscatcodes\restorecatcode
   \fi}
```

We also provide an alternative, that forces grouping when needed. An application of this macros can be found in buffering data.

```
\def\beginrestorecatcodes%
  {\ifx\restorecatcodes\relax
     \let\endrestorecatcodes=\relax
   \else
     \bgroup
     \let\beginrestorecatcodes=\bgroup
     \let\endrestorecatcodes=\egroup
   \fi}
```

The main copying routine of display verbatim does an ordinary string–compare on the saved closing command and the current line. The space after `#1` in the definition of `\next` is essential! As a result of using `\obeylines`, we have to use %'s after each line but none after the first `#1`.

```
{\obeylines%
 \gdef\copyverbatimline#1
   {\ifx\doverbatimline\relax% gobble rest of the first line
      \let\doverbatimline=\dodoverbatimline%
      \def\next{\copyverbatimline}%
    \else%
      \def\next{#1 }%
      \ifx\next\emptyspace%
        \def\next%
          {\doemptyverbatimline{#1}%
           \copyverbatimline}%
      \else%
        \edef\next{\meaning\next}%
        \ifx\next\endofverbatimcommand%
          \def\next%
            {\egroup\endofdisplayverbatim}%
        \else%
          \def\next%
            {\doverbatimline{#1}%
             \copyverbatimline}%
        \fi%
      \fi%
   \fi%
   \next}}
```

The actual typesetting of a line is done by a separate macro, which enables us to implement <tab> handling. The \do and \dodo macros take care of the preceding \parskip, while skipping the rest of the first line. The \relax is used as an signal.

\iflinepar A careful reader will see that \linepar is reset. This boolean can be used to determine if the current line is the first line in a pseudo paragraph and this boolean is set after each empty line.

```
35 \newif\iflinepar
```

```
36 \def\dodoverbatimline#1%
     {\leavevmode\the\everyline\strut\processverbatimline{#1}%
      \EveryPar{}%
      \lineparfalse
      \obeyedline\par}
```

\obeyemptyli.. Empty lines in verbatim can lead to white space on top of a new page. Because this is not what we want, we turn them into vertical skips. This default behavior can be overruled by:

```
\obeyemptylines
```

Although it would cost us only a few lines of code, we decided not to take care of multiple empty lines. When a (display) verbatim text contains more successive empty lines, this probably suits some purpose.

```
37 \bgroup
\catcode`\^^L=\@@active  \gdef\emptypage  {^^L}
\catcode`\^^M=\@@active  \gdef\emptyline  {^^M}
                         \gdef\emptyspace { }
\egroup
```

```
38 \def\doemptyverbatimline%
     {\vskip\ht\strutbox
      \vskip\dp\strutbox
      {\setbox0=\hbox{\the\everyline}}%
      \linepartrue}
```

```
39 \def\obeyemptylines%
     {\def\doemptyverbatimline{\doverbatimline}}
```

TEX does not offer \everyline, which is a direct result of its advanced multi–pass paragraph typesetting mechanism. Because in verbatim mode paragraphs and lines are more or less equal, we can easily implement our own simple \everyline support.

\EveryPar \EveryLine In this module we've reserved \everypar for the things to be done with paragraphs and \everyline for line specific actions. In CONTEXT however, we use \everypar for placing side- and column-floats, inhibiting indentation and some other purposes. In verbatim mode, every line becomes a paragraph, which means that \everypar is executed frequently. To be sure, the user specific use of both \everyline and \everypar is implemented by means of \EveryLine and \EveryPar.

We still have to take care of the <tab>. A <tab> takes eight spaces and a <space> normally has a width of 0.5 em. Because we can be halfway a tabulation, we must keep track of the position. This takes time, especially when we print complete files, therefore we \relax this mechanism by default.

```
40 \def\doprocesstabskip%
     {\obeyedspace % \hskip.5em  or  \hbox to .5em{}
      \ifdone
        \advance\scratchcounter by 1
```

```
    \let\next=\doprocesstabskip
    \donefalse
  \else\ifnum\scratchcounter>7\relax
    \let\next=\relax
  \else
    \advance\scratchcounter 1\relax
    \let\next=\doprocesstabskip
  \fi\fi
  \next}
```

41
```
\def\dodoprocesstabskipline#1#2\endoftabskipping%
  {\ifnum\scratchcounter>7\relax
     \scratchcounter=1\relax
     \donetrue
   \else
     \advance\scratchcounter 1\relax
     \donefalse
   \fi
   \ifx#1\relax
     \let\next=\relax
   \else
     \def\next{#1\dodoprocesstabskipline#2\endoftabskipping}%
   \fi
   \next}
```

42
```
\let\endoftabskipping    = \relax
\let\processverbatimline = \relax
```

43
```
\def\doprocesstabskipline#1%
  {\bgroup
   \scratchcounter=1\relax
   \dodoprocesstabskipline#1\relax\endoftabskipping
   \egroup}
```

\processfile..

The verbatim typesetting of files is done on a bit different basis. This time we don't check for a closing command, but look for `<eof>` and when we've met, we make sure it does not turn into an empty line.

```
\processfileverbatim{filename}
```

Typesetting a file in most cases results in more than one page. Because we don't want problems with files that are read in during the construction of the page, we set `\ifprocessingverbatim`, so the output routine can adapt its behavior. Originally we used `\scratchread`, but because we want to support nesting, we decided to use a separate input file.

44
```
\newif\ifprocessingverbatim
```

45
```
\newread\verbatiminput
```

46
```
\def\processfileverbatim#1%
  {\par
   \bgroup
   \parindent\!!zeropoint
   \ifdim\lastskip<\parskip
     \removelastskip
     \vskip\parskip
```

```
\fi
\parskip\!!zeropoint
\processingverbatimtrue
\linepartrue
\uncatcodecharacters
\verbatimfont
\frenchspacing
\obeyspaces
\obeytabs
\obeylines
\obeypages
\obeycharacters
\openin\verbatiminput=#1%
\def\doreadline%
  {\read\verbatiminput to \next
   \ifeof\verbatiminput
     % we don't want <eof> to be treated as <crlf>
   \else\ifx\next\emptyline
     \expandafter\doemptyverbatimline\expandafter{\next}%
   \else\ifx\next\emptypage
     \expandafter\doemptyverbatimline\expandafter{\next}%
   \else
     \expandafter\dodoverbatimline\expandafter{\next}%
   \fi\fi\fi
   \readline}%
\def\readline%
  {\ifeof\verbatiminput
     \let\next=\relax
   \else
     \let\next=\doreadline
   \fi
   \next}%
\readline
\closein\verbatiminput
\egroup
\ignorespaces}
```

These macro's can be used to construct the commands we mentioned in the beginning of this documentation. We leave this to the fantasy of the reader and only show some PLAIN TEX alternatives for display verbatim and listings. We define three commands for typesetting inline text, display text and files verbatim. The inline alternative also accepts user supplied delimiters.

```
\type{text}

\starttyping
... verbatim text ...
\stoptyping

\typefile{filename}
```

We can turn on the options by:

```
\controlspacetrue
\verbatimtabstrue
```

```
\prettyverbatimtrue
```

Here is the implementation:

```
47 \newif\ifcontrolspace
   \newif\ifverbatimtabs
   \newif\ifprettyverbatim

48 \def\presettyping%
     {\ifcontrolspace
        \let\obeyspace=\setcontrolspace
      \fi
      \ifverbatimtabs
        \let\obeytabs=\settabskips
      \fi
      \ifprettyverbatim
        \let\obeycharacters=\setupprettytextype
      \fi}

49 \def\type%
     {\bgroup
      \presettyping
      \processinlineverbatim{\egroup}}

50 \def\starttyping%
     {\bgroup
      \presettyping
      \processdisplayverbatim{\stoptyping}}

51 \def\stoptyping%
     {\egroup}

52 \def\typefile#1%
     {\bgroup
      \presettyping
      \processfileverbatim{#1}%
      \egroup}
```

One can use the different `\obeysomething` commands to influence the behavior of these macro's. We use for instance `\obeycharacters` for making `/` an active character when we want to include typesetting commands.

We'll spend the remainder of this article on coloring the verbatim text. At PRAGMA we use the integrated environment TEXEDIT for editing and processing TEX documents.[1] This program also supports real time spell checking and TEX based file management. Although definitely not exclusive, the programs cooperate nicely with CONTEXT. Because TEX can be considered a tool for experts, we've tried to put as less a burden on non–technical users as possible. This is accomplished in the following ways:

- We've added some trivial symmetry checking to TEXEDIT. Sources are checked for the use of brackets, braces, begin–end and start–stop like constructions, with or without arguments.
- Although TEX is very tolerant to unformatted input, we stimulate users to make the ASCII source as clean as possible. Many sources I've seen in distribution sets look so awful, that I sometimes wonder how people get them working. In our opinion, a good–looking source leads to less errors.

[1] TEXEDIT has been operative since 1991.

- We use parameter driven setups and make the commands as tolerant as possible. We don't accept commands that don't look nice in ASCII.
- Finally —I could have added some more— we use color.

When in spell–checking–mode, the words spelled correctly are shown in *green*, the unknown or wrongly spelled words are in *red* and upto four categories of words, for instance passive verbs and nouns, become *blue* (or cyan) or *yellow*. Short and nearly always correct words are in white (on a black screen). This makes checking–on–the–fly very easy and convenient, especially because we place the accents automatically.

In TEX–mode we show TEX–specific tokens and sequences of tokens in appropriate colors and again we use four colors. We use those colors in a way that supports parameter driven setups, table typesetting and easy visual checking of symmetry. Furthermore the text becomes more readable.

color	**characters that are influenced**
red	`{ } $`
green	`\this \!!that \??these \@@those`
yellow	`‘ ’ ~ ^ _ & / + -	%`
blue	`( ) # [ ] " < > =`

Macro–definition and style files often look quite green, because they contain many calls to macros. Pure text files on the other hand are mostly white (on the screen) and color clearly shows their structure.

When I prepared the interactive PDF manuals of CONTEXT, TEXEDIT and PPCHTEX (1995), I decided to include the original source text of the manuals as an appendix. At every chapter or (sub)section the reader can go to the corresponding line in the source, just to see how things were done in TEX. Of course, the reader can jump from the to corresponding typeset text too.

Confronted with those long (boring) sources, I decided that a colored output, in accordance with TEXEDIT would be nice. It would not only visually add some quality to the manual, but also make the sources more readable.

Apart from a lot of ⟨*catcode*⟩–magic, programming the color macros was surprisingly easy. Although the macro's are hooked into the standard CONTEXT verbatim mechanism, they are set up in a way that embedding them in another verbatim environment is possible.

We can turn on coloring by reassigning `\obeycharacters`:

```
\let\obeycharacters=\setupprettytextype
```

During pretty typesetting we can be in two states: *command* and *parameter*. The first condition becomes true if we encounter a backslash, the second state is entered when we meet a `#`.

```
53  \newif\ifintexcommand
    \newif\ifintexparameter
```

The mechanism described here, is meant to be used with color. It is nevertheless possible to use
\splittexpar.. different fonts instead of distinctive colors. When using color, it's better to end parameter mode after the `#`. When on the other hand we use a slanted typeface for the hashmark, then a slanted number looks better.

```
54  \newif\ifsplittexparameters   \splittexparameterstrue
```

\splittexcon..

With \splittexcontrols we can influence the way control characters are processed in macro names. By default, the ^^ part is uncolored. When this boolean is set to false, they get the same color as the other characters.

55 \newif\ifsplittexcontrols \splittexcontrolstrue

The next boolean is used for internal purposes only and keeps track of the length of the name. Because two–character sequences starting with a backslash are always seen as a command.

56 \newif\iffirstintexcommand

We use a maximum of four colors because more colors will distract too much. In the following table we show the logical names of the colors, their color and *rgb* values.

identifier	color	r	g	b	bw
texprettyone	red	0.9	0.0	0.0	0.30
texprettytwo	green	0.0	0.8	0.0	0.45
texprettythree	yellow	0.0	0.0	0.9	0.60
texprettyfour	blue	0.8	0.8	0.6	0.75

This following poor mans implementation of color is based on PostScript. One can of course use grayscales too. In the core modules these macros are redefined to using the color mechanism present in CONTEXT.

57
```
\def\setcolorverbatim%
  {\splittexparameterstrue
   \def\texprettyone   {.9 .0 .0 }       % red
   \def\texprettytwo   {.0 .8 .0 }       % green
   \def\texprettythree {.0 .0 .9 }       % blue
   \def\texprettyfour  {.8 .8 .6 }       % yellow
   \def\texbeginofpretty[##1]%
     {\special{ps:: \csname##1\endcsname setrgbcolor}}
   \def\texendofpretty%
     {\special{ps:: 0 0 0 setrgbcolor}}} % black
```

58
```
\def\setgrayverbatim%
  {\splittexparameterstrue
   \def\texprettyone   {.30 }            % gray
   \def\texprettytwo   {.45 }            % gray
   \def\texprettythree {.60 }            % gray
   \def\texprettyfour  {.75 }            % gray
   \def\texbeginofpretty[##1]%
     {\special{ps:: \csname##1\endcsname setgray}}
   \def\texendofpretty%
     {\special{ps:: 0 setgray}}}         % black
```

One can redefine these two commands after loading this module. When available, one can also use appropriate font–switch macro's. We default to color.

59 \setcolorverbatim

Here come the commands that are responsible for entering and leaving the two states. As we can see, they've got much in common.

```
60  \def\texbeginofcommand%
      {\texendofparameter
       \ifintexcommand
       \else
         \global\intexcommandtrue
         \global\firstintexcommandtrue
         \texbeginofpretty[texprettytwo]%
       \fi}

61  \def\texendofcommand%
      {\ifintexcommand
         \texendofpretty
         \global\intexcommandfalse
         \global\firstintexcommandfalse
       \fi}

62  \def\texbeginofparameter%
      {\texendofcommand
       \ifintexparameter
       \else
         \global\intexparametertrue
         \texbeginofpretty[texprettythree]%
       \fi}

63  \def\texendofparameter%
      {\ifintexparameter
         \texendofpretty
         \global\intexparameterfalse
       \fi}
```

We've got nine types of characters. The first type concerns the grouping characters that become red and type seven takes care of the backslash. Type eight is the most recently added one and handles the control characters starting with `^^`. In the definition part at the end of this module we can see how characters are organized by type.

```
64  \def\ifnotfirstintexcommand#1%
      {\iffirstintexcommand
         \string#1%
         \texendofcommand
       \else}

65  \def\textypeone#1%
      {\ifnotfirstintexcommand#1%
         \texendofcommand
         \texendofparameter
         \texbeginofpretty[texprettyone]\string#1\texendofpretty
       \fi}

66  \def\textypetwo#1%
      {\ifnotfirstintexcommand#1%
         \texendofcommand
         \texendofparameter
         \texbeginofpretty[texprettythree]\string#1\texendofpretty
       \fi}
```

```
67  \def\textypethree#1%
      {\ifnotfirstintexcommand#1%
         \texendofcommand
         \texendofparameter
         \texbeginofpretty[texprettyfour]\string#1\texendofpretty
       \fi}

68  \def\textypefour#1%
      {\ifnotfirstintexcommand#1%
         \texendofcommand
         \texendofparameter
         \string#1%
       \fi}

69  \def\textypefive#1%
      {\ifnotfirstintexcommand#1%
         \texbeginofparameter
         \string#1%
       \fi}

70  \def\textypesix#1%
      {\ifnotfirstintexcommand#1%
         \ifintexparameter
           \ifsplittexparameters
             \texendofparameter
             \string#1%
           \else
             \string#1%
             \texendofparameter
           \fi
         \else
           \texendofcommand
           \string#1%
         \fi
       \fi}

71  \def\textypeseven#1%
      {\ifnotfirstintexcommand#1%
         \texbeginofcommand
         \string#1%
       \fi}

72  \def\textypeeight#1#2%
      {\texendofparameter
       \ifx#1#2%
         \ifsplittexcontrols
           \ifintexcommand
             \texendofcommand
             \string#1\string#1%
             \texbeginofcommand
           \else
             \string#1\string#2%
           \fi
         \else
```

```
      \string#1\string#1%
    \fi
  \else
    \ifintexcommand
      \firstintexcommandfalse
      \string#1#2%
    \else
      \textypethree#1#2%
    \fi
  \fi}
```

73
```
\def\textypenine#1%
  {\texendofparameter
   \global\firstintexcommandfalse
   \string#1}
```

We have to take care of the control characters we mentioned before. We obey their old values but only after ending our two states.

74
```
\def\texsetcontrols%
  {\global\let\oldobeyedspace = \obeyedspace
   \global\let\oldobeyedline  = \obeyedline
   \global\let\oldobeyedpage  = \obeyedpage
   \def\obeyedspace%
     {\texendofcommand
      \texendofparameter
      \oldobeyedspace}%
   \def\obeyedline%
     {\texendofcommand
      \texendofparameter
      \oldobeyedline}%
   \def\obeyedpage%
     {\texendofcommand
      \texendofparameter
      \oldobeyedpage}%
   \let\obeytabs=\ignoretabs}
```

Next comes the tough part. We have to change the ⟨*catcode*⟩ of each character. These macro's are tuned for speed and simplicity. When viewed in color they look quite simple.

75
```
\def\setupprettytextype%
  {\texsetcontrols
   \texsetspecialpretty
   \texsetalphabetpretty
   \texsetextrapretty}
```

When handling the lowercase characters, we cannot use lowercased macro names. This means that we have to redefine some well known macros, like `\bgroup`.

76
```
\def\texpresetcatcode%
  {\def\\##1%
     {\expandafter\catcode\expandafter`\csname##1\endcsname\@@active}}
```

77
```
\def\texsettypenine%
  {\def\\##1%
```

```
    {\def##1{\textypenine##1}}}
```

78

```
\bgroup
  \bgroup
    \gdef\texpresetalphapretty%
      {\texpresetcatcode
       \\A\\B\\C\\D\\E\\F\\G\\H\\I\\J\\K\\L\\M%
       \\N\\O\\P\\Q\\R\\S\\T\\U\\V\\W\\X\\Y\\Z}
    \texpresetalphapretty
    \gdef\texsetalphapretty%
      {\texpresetalphapretty
       \texsettypenine
       \\A\\B\\C\\D\\E\\F\\G\\H\\I\\J\\K\\L\\M%
       \\N\\O\\P\\Q\\R\\S\\T\\U\\V\\W\\X\\Y\\Z}
  \egroup
  \global\let\TEXPRESETCATCODE = \texpresetcatcode
  \global\let\TEXSETTYPENINE   = \texsettypenine
  \global\let\BGROUP           = \bgroup
  \global\let\EGROUP           = \egroup
  \global\let\GDEF             = \gdef
  \BGROUP
    \GDEF\TEXPRESETALPHAPRETTY%
      {\TEXPRESETCATCODE
       \\a\\b\\c\\d\\e\\f\\g\\h\\i\\j\\k\\l\\m%
       \\n\\o\\p\\q\\r\\s\\t\\u\\v\\w\\x\\y\\z}
    \TEXPRESETALPHAPRETTY
    \GDEF\TEXSETALPHAPRETTY%
      {\TEXPRESETALPHAPRETTY
       \TEXSETTYPENINE
       \\a\\b\\c\\d\\e\\f\\g\\h\\i\\j\\k\\l\\m%
       \\n\\o\\p\\q\\r\\s\\t\\u\\v\\w\\x\\y\\z}
  \EGROUP
  \gdef\texsetalphabetpretty%
    {\texsetalphapretty
     \TEXSETALPHAPRETTY}
\egroup
```

Macro names normally only may contain characters, but in unprotected state we can also use the characters `@`, `!` and `?`. Of course they are only colored (green) when they are part of a name.

79

```
\bgroup
  \gdef\texpresetextrapretty%
    {\texpresetcatcode
     \\?\\!\\@}
  \texpresetextrapretty
  \gdef\texsetextrapretty%
    {\texpresetextrapretty
     \texsettypenine
     \\?\\!\\@}
\egroup
```

Here comes the main specification routine. In this macro we also have to change the escape character to `!` and use `X`, `Y` and `Z` for grouping and ignoring, which makes the result a bit less readable. Plain TEX defines `\+` as an outer macro, so we have to redefine this one too.

```
80  \def\+{\tabalign}

81  \bgroup
      \gdef\texpresetspecialpretty%
        {\def\\##1{\catcode`##1\@@active}%
         \\\[\\\]\\\=\\\<\\\>\\\#\\\(\\\)\\\"%
         \\\$\\\{\\\}%
         \\\-\\\+\\\|\\\%\\\/\\\_\\\^\\\&\\\~\\\'\\\`%
         \\\.\\\,\\\:\\\;%
         \\\*%
         \\\1\\\2\\\3\\\4\\\5\\\6\\\7\\\8\\\9%
         \\\\}
      \catcode`\X=\the\catcode`\{
      \catcode`\Y=\the\catcode`\}
      \catcode`\Z=\the\catcode`\%
      \gdef\texsetsometypes%
        {\def\!##1##2{\def##1{##2{##1}}}}%
      XZ
       \catcode`\!=\@@escape
       !texpresetspecialpretty
       !gdef!texsetspecialpretty
         XZ
          !texpresetspecialpretty
          !texsetsometypes
          !! $ !textypeone   !! { !textypeone   !! } !textypeone
          !! [ !textypetwo   !! ] !textypetwo   !! ( !textypetwo   !! ) !textypetwo
          !! = !textypetwo   !! < !textypetwo   !! > !textypetwo   !! " !textypetwo
          !! - !textypethree !! + !textypethree !! / !textypethree
          !! | !textypethree !! % !textypethree !! ’ !textypethree !! ‘ !textypethree
          !! _ !textypethree !! ^ !textypethree !! & !textypethree !! ~ !textypethree
          !! . !textypefour  !! , !textypefour  !! : !textypefour  !! ; !textypefour
          !! * !textypefour
          !! # !textypefive
          !! 1 !textypesix   !! 2 !textypesix   !! 3 !textypesix
          !! 4 !textypesix   !! 5 !textypesix   !! 6 !textypesix
          !! 7 !textypesix   !! 8 !textypesix   !! 9 !textypesix
          !! \ !textypeseven
          !! ^ !textypeeight
         YZ
      YZ
    \egroup
```

This text was published in the MAPS of the dutch TEX users group NTG. In that article, the verbatim part of the text was set with the following commands for the examples:

```
\def\starttypen% We simplify the \ConTeXt\ macro.
  {\bgroup
   \everypar{} % We disable some troublesome mechanisms.
   \advance\leftskip by 1em
   \processdisplayverbatim{\stoptypen}}

\def\stoptypen%
  {\egroup}
```

The implementation itself was typeset with:

```
\def\startdefinition%
  {\bgroup
   \everypar{} % Again we disable some troublesome mechanisms.
   \let\obeycharacters=\setupprettytextype
   \EveryPar{\showparagraphcounter}%
   \EveryLine{\showlinecounter}%
   \verbatimcorps
   \processdisplayverbatim{\stopdefinition}}

\def\stopdefinition%
  {\egroup}
```

And because we have both `\EveryPar` and `\EveryLine` available, we can implement a dual numbering mechanism:

```
\newcount\paragraphcounter
\newcount\linecounter

\def\showparagraphcounter%
  {\llap
     {\bgroup
      \counterfont
      \hbox to 4em
        {\global\advance\paragraphcounter by 1
         \hss \the\paragraphcounter \hskip2em}%
      \egroup
      \hskip1em}}

\def\showlinecounter%
  {\llap
     {\bgroup
      \counterfont
      \hbox to 2em
        {\global\advance\linecounter by 1
         \hss \the\linecounter}%
      \egroup
      \hskip1em}}
```

One may have noticed that the `\EveryPar` is only executed once, because we consider each piece of verbatim as one paragraph. When one wants to take the empty lines into account, the following assignments are appropriate:

```
\EveryLine
  {\iflinepar
     \showparagraphcounter
   \fi
   \showlinecounter}
```

In this case, nothing has to be assigned to `\EveryPar`, maybe except of just another extra numbering scheme. The macros used to typeset this documentation are a bit more complicated, because we have to take take 'long' margin lists into account. When such a list exceeds the previous pargraph we postpone placement of the paragraph number till there's room. This way so it does not clash with the margin words.

Normally such commands have to be embedded in a decent setup structure, where options can be set at will.

Now let's summarize the most important commands.

```
\processinlineverbatim{\closingcommand}
\processdisplayverbatim{\closingcommand}
\processfileverbatim{filename}
```

We can satisfy our own specific needs with the following interfacing macro's:

```
\obeyspaces  \obeytabs  \obeylines  \obeypages  \obeycharacters
```

Some needs are fulfilled already with:

```
\setcontrolspace  \settabskips  \setupprettytextype
```

lines can be enhanced with ornaments using:

```
\everypar  \everyline  \iflinepar
```

and color support is implemented by:

```
\texbeginofpretty[#1] ... \texendofpretty
```

We can influence the verbatim environment with the following macro and booleans:

```
\obeyemptylines  \splittexparameters...  \splittexcontrols...
```

The color support macro can be redefined by the user. The parameter `#1` can be one of the four 'fixed' identifiers *texprettyone*, *texprettytwo*, *texprettythree* and *texprettyfour*. We have implemented a more or less general PostScript color support mechanism, using `specials`. One can toggle between color and grayscale with:

```
\setgrayverbatim  \setcolorverbatim
```

We did not mention one drawback of the mechanism described here. The closing command must start at the first position of the line. In CONTEXT we will not have this drawback, because we can test if the end command is a substring of the current line. The testing is done by two of the support macros, which of course are not available in a stand alone application of this module.

\permitshift..

82
```
\ifx \undefined \doifinstringelse \else
```

83
```
\def\processdisplayverbatim#1%
  {\par
   \bgroup
   \escapechar=-1
   \xdef\verbatimname{\string#1}%
   \egroup
   \def\endofdisplayverbatim{\csname\verbatimname\endcsname}%
   \bgroup
   \parindent\!!zeropoint
   \ifdim\lastskip<\parskip
     \removelastskip
     \vskip\parskip
   \fi
   \parskip\!!zeropoint
   \processingverbatimtrue
```

```
   \expandafter\let\csname\verbatimname\endcsname=\relax
   \expandafter\convertargument\csname\verbatimname\endcsname
     \to\endofverbatimcommand
   \verbatimfont
   \setupcopyverbatim
   \let\doverbatimline=\relax
   \copyverbatimline}

\let\doifendofverbatim=\doifelse

\def\permitshiftedendofverbatim%
  {\let\doifendofverbatim=\doifinstringelse}

{\obeylines%
 \gdef\copyverbatimline#1
   {\ifx\doverbatimline\relax% gobble rest of the first line
      \let\doverbatimline=\dodoverbatimline%
      \def\next{\copyverbatimline}%
    \else%
      \convertargument#1 \to\next%
      \ifx\next\emptyspace%
        \def\next%
          {\doemptyverbatimline{#1}%
           \copyverbatimline}%
      \else%
        \doifendofverbatim{\endofverbatimcommand}{\next}%
          {\def\next%
             {\egroup\endofdisplayverbatim}}%
          {\def\next%
             {\doverbatimline{#1}%
              \copyverbatimline}}%
      \fi%
   \fi%
   \next}}

\fi

\protect
```