ConTEXt

Generic Modules

Hans Hagen / PRAGMA

Zwolle, The Netherlands

1996 October 8

version : 1996 October 8
address : PRAGMA
Postbox 125
8000 AC Zwolle
The Netherlands
e-mail : pragma@pi.net

1 Missing

Some support modules are more or less independant. This module, which is not part of plain CONTEXT, provides the missing macros and declarations of registers.

The next calls prevent reloading or, what's even worse, overloading of already defined macros and registers. We do so because some definitions are substitutes. Just to be sure we test on two macros.

```
\ifx \undefined \writestatus
  \let\next=\relax
\else \ifx \undefined \unprotect
  \let\next=\relax
\else
  \let\next=\endinput
\fi \fi
```

```
\next
```

\writestatus We start each module with a message. Normally the output is formatted, but here we keep things simple.

```
\def\writestatus#1#2%
  {\immediate\write16{#1 : #2}}
```

Lets see if it works.

```
\writestatus{loading}{Context Support Macros / Missing}
```

\protect \unprotect Next we present a poor mans alternative for **\protect** and **\unprotect**, two commands that enable us to use the characters **@**, **!** and **?** in macro names.

```
\ifx \undefined \protect
  \let\protect=\relax
\fi
```

```
\def\unprotect%
  {\catcode`@=11
   \catcode`!=11
   \catcode`?=11
   \let\normalprotect=\protect
   \edef\protect%
     {\catcode`@=\the\catcode`@\relax
      \catcode`!=\the\catcode`!\relax
      \catcode`?=\the\catcode`?\relax
      \let\protect=\normalprotect}}
```

We start using this one it at once.

```
\unprotect
```

\scratch... \if... \next... We need some scratch registers. Users are free to use them, but can never be sure of their value once another macro is called. We only allocate things when they are yet undefined. This way we can't mess up other macro packages, but of course previous definitions can mess up our modules.

These macros are a bit complicated by the fact that Plain TEX defines the **\new**–macros as being outer. Furthermore nested **\if**'s can get us into trouble.

```
\def\definecontextobject%
  {\iftrue}
```

```
\def\gobblecontextobject%
  {\setbox0=\hbox
     \bgroup
     \long\def\gobblecontextobject##1\fi{\egroup}%
     \expandafter\gobblecontextobject\string}

\def\ifnocontextobject#1\do%
  {\ifx#1\undefined
     \let\next=\definecontextobject
   \else
     \writestatus{system}{beware of conflicting \string#1}%
     \let\next=\gobblecontextobject
   \fi
   \next}

\ifnocontextobject \scratchcounter       \do \newcount  \scratchcounter  \fi
\ifnocontextobject \scratchdimen         \do \newdimen  \scratchdimen    \fi
\ifnocontextobject \scratchskip          \do \newskip   \scratchskip     \fi
\ifnocontextobject \scratchmuskip        \do \newmuskip \scratchmuskip   \fi
\ifnocontextobject \scratchbox           \do \newbox    \scratchbox      \fi
\ifnocontextobject \scratchread          \do \newread   \scratchread     \fi
\ifnocontextobject \scratchwrite         \do \newwrite  \scratchread     \fi

\ifnocontextobject \nextbox              \do \newbox    \nextbox         \fi

\ifnocontextobject \nextdepth            \do \newdimen  \nextdepth       \fi

\ifnocontextobject \ifCONTEXTtrue        \do \newif\ifCONTEXT            \fi
\ifnocontextobject \ifdonetrue           \do \newif\ifdone               \fi
\ifnocontextobject \ifeightbitcharacters \do \newif\ifeightbitcharacters \fi
```

`\@@...` We use symbolic name for ⟨*catcodes*⟩. Thee can only be used when we are in unprotected state.

```
\ifnocontextobject \@@escape             \do \chardef\@@escape     = 0   \fi
\ifnocontextobject \@@begingroup         \do \chardef\@@begingroup = 1   \fi
\ifnocontextobject \@@endgroup           \do \chardef\@@endgroup   = 2   \fi
\ifnocontextobject \@@letter             \do \chardef\@@letter     = 11  \fi
\ifnocontextobject \@@other              \do \chardef\@@other      = 12  \fi
\ifnocontextobject \@@active             \do \chardef\@@active     = 13  \fi
```

`\everyline` `\EveryLine` `\EveryPar` In CONTEXT we use `\everypar` for special purposes and provide `\EveryPar` as an alternative. The same goes for `\everyline` and `\EveryLine`.

```
\ifnocontextobject \everyline            \do \newtoks\everyline          \fi
\ifnocontextobject \EveryPar             \do \let\EveryPar  = \everypar  \fi
\ifnocontextobject \EveryLine            \do \let\EveryLine = \everyline \fi
```

`\!!...` We reserve ourselves some scratch strings (i.e. macros).

```
\ifnocontextobject \!!stringa            \do \def\!!stringa     {}       \fi
\ifnocontextobject \!!stringb            \do \def\!!stringb     {}       \fi
\ifnocontextobject \!!stringc            \do \def\!!stringc     {}       \fi
\ifnocontextobject \!!stringd            \do \def\!!stringd     {}       \fi
```

\!!... The next set of definitions speed up processing a bit. Furthermore it saves memory.

```
\ifnocontextobject \!!zeropoint          \do \def\!!zeropoint   {0pt}    \fi
\ifnocontextobject \!!tenthousand        \do \def\!!tenthousand {10000}  \fi

\ifnocontextobject \!!width              \do \def\!!width       {width}  \fi
\ifnocontextobject \!!height             \do \def\!!height      {height} \fi
\ifnocontextobject \!!depth              \do \def\!!depth       {depth}  \fi

\ifnocontextobject \!!plus               \do \def\!!plus        {plus}   \fi
\ifnocontextobject \!!minus              \do \def\!!minus       {minus}  \fi
```

\smashbox The system modules offer a range of smashing macros, of which we only copied **\smashbox**.

```
\ifnocontextobject \smashbox \do

  \def\smashbox#1%
    {\wd#1=\!!zeropoint
     \ht#1=\!!zeropoint
     \dp#1=\!!zeropoint}

\fi
```

\dowithnextbox Also without further comment, we introduce a macro that gets the next box and does something usefull with it. Because the **\afterassignment** is executed inside the box, we have to use a **\aftergroup** too.

```
\ifnocontextobject \dowithnextbox \do

  \def\dowithnextbox#1%
    {\def\dodowithnextbox{#1}%
     \afterassignment\dododowithnextbox
     \setbox\nextbox}

  \def\dododowithnextbox%
    {\aftergroup\dodowithnextbox}

\fi
```

That's it. Please forget this junk and take a look at how it should be done.

```
\protect
```

2 Verbatim

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

```
\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.

```
\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.

```
\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`.

```
\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
\setcontrolspaces

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
```

\obeycharacters

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
```

\processinlineverbatim

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
   \def\endofverbatimcommand{#1\egroup}%
   \bgroup
   \aftergroup\endofverbatimcommand
   \futurelet\next\doprocessinlineverbatim}
```

\processdisplayverbatim

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}
```

\ifeightbitcharacters

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.

24
```
\def\uncatcodecharacters%
  {\ifeightbitcharacters
     \scratchcounter=255
   \else
     \scratchcounter=127
```

```
\fi
\loop
  \catcode\scratchcounter=\@@letter
  \advance\scratchcounter by -1
  \ifnum\scratchcounter>-1
\repeat}
```

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`.

25
```
{\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.

26
```
\newif\iflinepar
```

27
```
\def\dodoverbatimline#1%
  {\leavevmode\the\everyline\strut\processverbatimline{#1}%
   \EveryPar{}%
   \lineparfalse
   \obeyedline\par}
```

`\obeyemptylines` 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.

28
```
\bgroup
\catcode`\^^L=\@@active  \gdef\emptypage  {^^L}
\catcode`\^^M=\@@active  \gdef\emptyline  {^^M}
                         \gdef\emptyspace { }
\egroup
```

29
```
\def\doemptyverbatimline%
  {\vskip\ht\strutbox
   \vskip\dp\strutbox
   {\setbox0=\hbox{\the\everyline}}%
   \linepartrue}
```

30
```
\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 columnfloats, 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.

31
```
\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}
```

32
```
\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}

33  \let\endoftabskipping    = \relax
    \let\processverbatimline = \relax

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

\processfileverbatim

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.

```
35  \newif\ifprocessingverbatim

36  \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\scratchread=#1%
       \def\doreadline%
         {\read\scratchread to \next
          \ifeof\scratchread
            % 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\scratchread
       \let\next=\relax
     \else
       \let\next=\doreadline
     \fi
     \next}%
  \readline
  \closein\scratchread
  \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:

37
```
\newif\ifcontrolspace
\newif\ifverbatimtabs
\newif\ifprettyverbatim
```

38
```
\def\presettyping%
  {\ifcontrolspace
     \let\obeyspace=\setcontrolspace
   \fi
   \ifverbatimtabs
     \let\obeytabs=\settabskips
   \fi
   \ifprettyverbatim
     \let\obeycharacters=\setupprettytextype
   \fi}
```

39
```
\def\type%
  {\bgroup
   \presettyping
   \processinlineverbatim{\egroup}}
```

40
```
\def\starttyping%
  {\bgroup
   \presettyping
```

```
  \processdisplayverbatim{\stoptyping}}

\def\stoptyping%
  {\egroup}

\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.
- 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

[1] TEXEDIT has been operative since 1991.

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 `#`.

43
```
\newif\ifintexcommand
\newif\ifintexparameter
```

`\splittexparameters` The mechanism described here, is meant to be used with color. It is nevertheless possible to use 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.

44
```
\newif\ifsplittexparameters    \splittexparameterstrue
```

`\splittexcontrols` 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.

45
```
\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.

46
```
\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.

47
```
\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
```

48
```
\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.

49
```
\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.

50
```
\def\texbeginofcommand%
  {\texendofparameter
   \ifintexcommand
   \else
     \global\intexcommandtrue
     \global\firstintexcommandtrue
     \texbeginofpretty[texprettytwo]%
   \fi}
```

51
```
\def\texendofcommand%
  {\ifintexcommand
     \texendofpretty
     \global\intexcommandfalse
     \global\firstintexcommandfalse
   \fi}
```

52
```
\def\texbeginofparameter%
  {\texendofcommand
   \ifintexparameter
   \else
     \global\intexparametertrue
     \texbeginofpretty[texprettythree]%
   \fi}
```

53
```
\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.

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

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

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

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

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

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

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

```
      \fi
    \fi}
```

61
```
\def\textypeseven#1%
  {\ifnotfirstintexcommand#1%
     \texbeginofcommand
     \string#1%
   \fi}
```

62
```
\def\dodotextypeeight#1%
  {\texendofparameter
   \ifx\next#1%
     \ifsplittexcontrols
       \ifintexcommand
         \texendofcommand
         \string#1\string#1%
         \texbeginofcommand
       \else
         \string#1\string#1%
       \fi
     \else
       \string#1\string#1%
     \fi
     \let\next=\relax
   \else
     \textypethree#1%
   \fi
   \next}
```

63
```
\def\textypeeight#1%
  {\def\dotextypeeight{\dodotextypeeight#1}%
   \afterassignment\dotextypeeight\let\next=}
```

64
```
\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.

65
```
\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}}
```

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.

66
```
\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`.

67
```
\def\texpresetcatcode%
  {\def\\##1%
     {\expandafter\catcode\expandafter`\csname##1\endcsname\@@active}}
```

68
```
\def\texsettypenine%
  {\def\\##1%
     {\def##1{\textypenine##1}}}
```

69
```
\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.

70
```
\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.

71
```
\def\+{\tabalign}
```

72
```
\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
```

\permitshiftedendofver.. 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.

```
73  \ifx \undefined \convertargument \else

74  \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
   \@EA\convertargument\csname\verbatimname\endcsname\to\endofverbatimcommand
   \verbatimfont
   \setupcopyverbatim
   \let\doverbatimline=\relax
   \copyverbatimline}

75 \let\doifendofverbatim=\doifinstringelse

76 \def\permitshiftedendofverbatim%
     {\let\doifendofverbatim=\doifelse}

77 {\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}}

78 \fi

79 \protect
```

3 Visualization

Although an integral part of CONTEXT, this module is one of the support modules. Its stand alone character permits use in PLAIN TEX or TEX based macropackages.

This module is still in development. Depending on my personal need and those of whoever uses it, the macros will be improved in terms of visualization, efficiency and compatibility.

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

One of the strong points of TEX is abstraction of textual input. When macros are defined well and do what we want them to do, we will seldom need the tools present in What You See Is What You Get systems. For instance, when entering text we don't need rulers, because no manual shifting and/or alignment of text is needed. On the other hand, when we are designing macros or specifying layout elements, some insight in TEX's advanced spacing, kerning, filling, boxing and punishment abilities will be handy. That's why we've implemented a mechanism that shows some of the inner secrets of TEX.

```
\writestatus{loading}{Context Support Macros / Visualization}
```

In this module we are going to redefine some TEX primitives and PLAIN macro's. Their original meaning is saved in macros with corresponding names, preceded by `normal`. These original macros are (1) used to temporary restore the old values when needed and (2) used to prevent recursive calls in the macros that replace them.

```
\unprotect
```

\normalhbox \normalvbox \normalvtop

There are three types of boxes, one horizontal and two vertical in nature. As we will see later on, all three types are to be handled according to their orientation and baseline behavior. Especially `\vtop`'s need our special attention.

```
\let\normalhbox     = \hbox
\let\normalvbox     = \vbox
\let\normalvtop     = \vtop
```

\normalhskip \normalvskip

Next come the flexible skips, which come in two flavors too. Like boxes these are handled with TEX primitives.

```
\let\normalhskip    = \hskip
\let\normalvskip    = \vskip
```

\normalpenalty \normalkern

Both penalties and kerns are taken care of by mode sensitive primitives. This means that when making them visible, we have to take the current mode into account.

```
\let\normalpenalty  = \penalty
\let\normalkern     = \kern
```

\normalhglue \normalvglue

Glues on the other hand are macro's defined in PLAIN TEX. As we will see, their definitions make the implementation of their visible counterparts a bit more TEXnical.

```
\let\normalhglue    = \hglue
\let\normalvglue    = \vglue
```

\normalmkern
\normalmskip

Math mode has its own spacing primitives, preceded by `m`. Due to the relation with the current font and the way math is typeset, their unit `mu` is not compatible with other dimensions. As a result, the visual appearance of these primitives is kept primitive too.

```
\let\normalmkern    = \mkern
\let\normalmskip    = \mskip
```

\hfilneg
\vfilneg

Fills can be made visible quite easy. We only need some additional negation macros. Because PLAIN TEX only offers `\hfilneg` and `\vfilneg`, we define our own alternative double `ll`'ed ones.

```
\def\hfillneg%
  {\normalhskip\!!zeropoint \!!plus-1fill\relax}

\def\vfillneg%
  {\normalvskip\!!zeropoint \!!plus-1fill\relax}
```

\normalhss
\normalhfil
\normalhfill
\normalvss
\normalvfil
\normalvfill

The positive stretch primitives are used independant and in combination with `\leaders`.

```
\let\normalhss      = \hss
\let\normalhfil     = \hfil
\let\normalhfill    = \hfill
\let\normalvss      = \vss
\let\normalvfil     = \vfil
\let\normalvfill    = \vfill
```

\normalhfilneg
\normalhfillneg
\normalvfilneg
\normalvfillneg

Keep in mind that both `\hfillneg` and `\vfillneg` are not part of PLAIN TEX and therefore not documented in standard TEX documentation. They can nevertheless be used at will.

```
\let\normalhfilneg  = \hfilneg
\let\normalhfillneg = \hfillneg
\let\normalvfilneg  = \vfilneg
\let\normalvfillneg = \vfillneg
```

Visualization is not always wanted. Instead of turning this option off in those (unpredictable) situations, we just redefine a few PLAIN macros.

```
\def\rlap#1{\normalhbox to \!!zeropoint{#1\normalhss}}
\def\llap#1{\normalhbox to \!!zeropoint{\normalhss#1}}
```

```
\def~{\normalpenalty\!!tenthousand\ }
```

\makeruledbox

Ruled boxes can be typeset is many ways. Here we present just one alternative. This implementation may be a little complicated, but it supports all three kind of boxes. The next command expects a ⟨*box*⟩ specification, like:

```
\makeruledbox0
```

\baselinerule
\baselinefill
\baselinesmash

We can make the baseline of a box visible, both dashed and as a rule. Normally the line is drawn on top of the baseline, but a smashed alternative is offered too. If we want them all, we just say:

```
\baselineruletrue
\baselinefilltrue
\baselinesmashtrue
```

At the cost of some overhead these alternatives are implemented using `\if`'s:

```
\newif\ifbaselinerule  \baselineruletrue
\newif\ifbaselinefill  \baselinefillfalse
\newif\ifbaselinesmash \baselinesmashfalse
```

\toprule
\bottomrule
\leftrule
\rightrule

Rules can be turned on and off, but by default we have:

```
\topruletrue
\bottomruletrue
\leftruletrue
\rightruletrue
```

As we see below:

16
```
\newif\iftoprule       \topruletrue
\newif\ifbottomrule    \bottomruletrue
\newif\ifleftrule      \leftruletrue
\newif\ifrightrule     \rightruletrue
```

\boxrulewidth

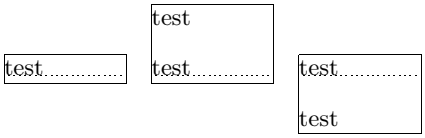
The width in the surrounding rules can be specified by assigning an apropriate value to the dimension used. This module defaults the width to:

```
\boxrulewidth=.2pt
```

Although we are already low on ⟨*dimensions*⟩ it's best to spend one here, mainly because it enables easy manipulation, like multiplication by a given factor.

17
```
\newdimen\boxrulewidth \boxrulewidth=.2pt
```

The core macro **\makeruledbox** looks a bit hefty. The manipulation at the end is needed because we want to preserve both the mode and the baseline. This means that **\vtop**'s and **\vbox**'es behave the way we expect them to do.



The **\cleaders** part of the macro is responsible for the visual baseline. The **\normalhfill** belongs to this primitive too. By storing and restoring the height and depth of box **#1**, we preserve the mode.

18
```
\def\makeruledbox#1%
  {\edef\ruledheight {\the\ht#1}%
   \edef\ruleddepth  {\the\dp#1}%
   \edef\ruledwidth  {\the\wd#1}%
   \setbox\scratchbox=\normalvbox
     {\dontcomplain
      \offinterlineskip
      \hrule
        \!!height\boxrulewidth
        \iftoprule\else\!!width\!!zeropoint\fi
      \normalvskip-\boxrulewidth
      \normalhbox to \ruledwidth
         {\vrule
            \!!height\ruledheight
            \!!depth\ruleddepth
            \!!width\ifleftrule\else0\fi\boxrulewidth
          \ifdim\ruledheight>\!!zeropoint \else \baselinerulefalse \fi
          \ifdim\ruleddepth>\!!zeropoint \else \baselinerulefalse \fi
          \ifbaselinerule
            \ifdim\ruledwidth<20\boxrulewidth
```

```
      \baselinefilltrue
    \fi
    \cleaders
      \ifbaselinefill
        \hrule
          \ifbaselinesmash
            \!!height\boxrulewidth
          \else
            \!!height.5\boxrulewidth
            \!!depth.5\boxrulewidth
          \fi
      \else
        \normalhbox
          {\normalhskip2.5\boxrulewidth
           \vrule
           \ifbaselinesmash
             \!!height\boxrulewidth
           \else
             \!!height.5\boxrulewidth
             \!!depth.5\boxrulewidth
           \fi
           \!!width5\boxrulewidth
           \normalhskip2.5\boxrulewidth}%
      \fi
  \fi
  \normalhfill
  \vrule
    \!!width\ifrightrule\else0\fi\boxrulewidth}%
\normalvskip-\boxrulewidth
\hrule
  \!!height\boxrulewidth
  \ifbottomrule\else\!!width\!!zeropoint\fi}%
\wd#1=\!!zeropoint
\setbox#1=\ifhbox#1\normalhbox\else\normalvbox\fi
  {\normalhbox{\box#1\lower\ruleddepth\box\scratchbox}}%
\ht#1=\ruledheight
\wd#1=\ruledwidth
\dp#1=\ruleddepth}
```

Just in case one didn't notice: the rules are in fact layed over the box. This way the contents of a box cannot visually interfere with the rules around (upon) it. A more advanced version of ruled boxes can be found in one of the core modules of CONTEXT. There we take offsets, color, rounded corners, backgrounds and alignment into account too.

\ruledhbox
\ruledvbox
\ruledvtop

These macro's can be used instead of **\hbox**, **\vbox** and **\vtop**. They just do what their names state. Using an auxiliary macro would save us a few words of memory, but it would make their appearance even more obscure.

one two three four five

```
\hbox
  {\strut
   one
   two
   \hbox{three}
   four
   five}
```

19
```
\def\ruledhbox%
  {\normalhbox\bgroup
   \dowithnextbox{\makeruledbox\nextbox\box\nextbox\egroup}%
   \normalhbox}
```

first line
second line
third line
fourth line
fifth line

```
\vbox
  {\strut
   first line  \par
   second line \par
   third line  \par
   fourth line \par
   fifth line
   \strut }
```

20
```
\def\ruledvbox%
  {\normalvbox\bgroup
   \dowithnextbox{\makeruledbox\nextbox\box\nextbox\egroup}%
   \normalvbox}
```

first line
second line
third line
fourth line
fifth line

```
\vtop
  {\strut
   first line  \par
   second line \par
   third line  \par
   fourth line \par
   fifth line
   \strut }
```

21
```
\def\ruledvtop%
  {\normalvtop\bgroup
   \dowithnextbox{\makeruledbox\nextbox\box\nextbox\egroup}%
   \normalvtop}
```

\ruledbox
\setruledbox

Of the next two macros the first can be used to precede a box of ones own choice. One can for instance prefix boxes with \ruledbox and afterwards — when the macro satisfy the needs — let it to \relax.

```
\ruledbox\hbox{What rules do you mean?}
```

The macro \setruledbox can be used to directly rule a box.

```
\setruledbox12=\hbox{Who's talking about rules here?}
```

At the cost of some extra macros we can implement a variant that does not need the =, but we stick to:

22
```
\def\ruledbox%
  {\dowithnextbox{\makeruledbox\nextbox\box\nextbox}}
```

23
```
\def\setruledbox#1=%
  {\dowithnextbox{\makeruledbox\nextbox\setbox#1=\nextbox}}
```

\investigateskip
\investigatecount
\investigatemuskip

Before we meet the visualizing macro's, we first implement ourselves some handy utility ones. Just for the sake of efficiency and readability, we introduce some status variables, that tell us a bit more about the registers we use:

```
\ifflexible
\ifzero
\ifnegative
\ifpositive
```

These status variables are set when we call for one of the investigation macros, e.g.

```
\investigateskip\scratchskip
```

We use some dirty trick to check stretchability of ⟨*skips*⟩. Users of these macros are invited to study their exact behavior first. The positive and negative states both include zero and are in fact non-negative (≥ 0) and non-positive (≤ 0) .

24
```
\newif\ifflexible
\newif\ifzero
\newif\ifnegative
\newif\ifpositive
```

25
```
\def\investigateskip#1%
  {\relax
   \scratchdimen=#1\relax
   \edef\!!stringa{\the\scratchdimen}%
   \edef\!!stringb{\the#1}%
   \ifx\!!stringa\!!stringb \flexiblefalse \else \flexibletrue \fi
   \ifdim#1=\!!zeropoint\relax
     \zerotrue      \else
     \zerofalse     \fi
   \ifdim#1<\!!zeropoint\relax
     \positivefalse \else
     \positivetrue  \fi
   \ifdim#1>\!!zeropoint\relax
     \negativefalse \else
     \negativetrue  \fi}
```

26
```
\def\investigatecount#1%
  {\relax
   \flexiblefalse
   \ifnum#1=0
     \zerotrue      \else
     \zerofalse     \fi
   \ifnum#1<0
     \positivefalse \else
     \positivetrue  \fi
   \ifnum#1>0
     \negativefalse \else
     \negativetrue  \fi}
```

27
```
\def\investigatemuskip#1%
  {\relax
   \edef\!!stringa{\the\scratchmuskip}%
   \edef\!!stringb{0mu}%
   \def\!!stringc##1##2\\{##1}%
   \expandafter\edef\expandafter\!!stringc\expandafter
```

```
    {\expandafter\!!stringc\!!stringa\\}%
  \edef\!!stringd{-}%
  \flexiblefalse
  \ifx\!!stringa\!!stringb
    \zerotrue
    \negativefalse
    \positivefalse
  \else
    \zerofalse
    \ifx\!!stringc\!!stringd
      \positivefalse
      \negativetrue
    \else
      \positivetrue
      \negativefalse
    \fi
  \fi}
```

\dontinterfere Indentation, left and/or right skips, redefinition of **\par** and assignments to **\everypar** can lead to unwanted results. We can therefore turn all those things off with **\dontinterfere**.

28
```
\def\dontinterfere%
  {\everypar  = {}%
   \let\par   = \endgraf
   \parindent = \!!zeropoint
   \parskip   = \!!zeropoint
   \leftskip  = \!!zeropoint
   \rightskip = \!!zeropoint
   \relax}
```

\dontcomplain In this module we do a lot of box manipulations. Because we don't want to be confronted with to many over- and underfull messages we introduce **\dontcomplain**.

29
```
\def\dontcomplain%
  {\hbadness  = \!!tenthousand
   \hfuzz     = \maxdimen
   \vbadness  = \!!tenthousand
   \vfuzz     = \maxdimen}
```

Now the neccessary utility macros are defined, we can make a start with the visualizing ones. The implementation of these macros is a compromise between readability, efficiency of coding and processing speed. Sometimes we do in steps what could have been done in combination, sometimes we use a few boxes more or less then actually needed, and more than once one can find the same piece of rule drawing code twice.

\ifcenteredvcue \normalvcue Depending on the context, one can force visual vertical cues being centered along **\hsize** or being put at the current position. Although centering often looks better, we've chosen the second alternative as default. The main reason for doing so is that often when we don't set the **\hsize** ourselves, TEX takes the value of the surrounding box. As a result the visual cues can migrate outside the current context.

This behavior is accomplished by a small but effective auxiliary macro, which behavior can be influenced by the boolean **\centeredvcue**. By saying

```
\centeredvcuetrue
```

one turns centering on. As said, we turn it off.

```
\newif\ifcenteredvcue  \centeredvcuefalse
```

```
\def\normalvcue#1%
  {\normalhbox \ifcenteredvcue to \hsize \fi {\normalhss#1\normalhss}}
```

We could have used the more robust version

```
\def\normalvcue%
  {\normalhbox \ifcenteredvcue to \hsize \fi
   \bgroup\bgroup\normalhss
   \aftergroup\normalhss\aftergroup\egroup
   \let\next=}
```

or the probably best one:

```
\def\normalvcue%
  {\hbox \ifcenteredvcue to \hsize
     \bgroup\bgroup\normalhss
     \aftergroup\normalhss\aftergroup\egroup
   \else
     \bgroup
   \fi
   \let\next=}
```

Because we don't have to preserve ⟨*catcodes*⟩ and only use small arguments, we stick to the first alternative.

\testrulewidth

We build our visual cues out of rules. At the cost of a much bigger DVI file, this is to be prefered over using characters (1) because we cannot be sure of their availability and (2) because their dimensions are fixed.

As with ruled boxes, we use a ⟨*dimension*⟩ to specify the width of the ruled elements. This dimension defaults to:

```
\testrulewidth=\boxrulewidth
```

Because we prefer whole numbers for specifying the dimensions, we often use even multiples of `\testrulewidth`.

\visiblestretch

A second variable is introduced because of the stretch components of ⟨*skips*⟩. At the cost of some accuracy we can make this stretch visible.

```
\visiblestretchtrue
```

```
\newdimen\testrulewidth  \testrulewidth=\boxrulewidth
\newif\ifvisiblestretch  \visiblestretchfalse
```

\ruledhss
\ruledhfil
\ruledhfilneg
\ruledhfill
\ruledhfillneg

We start with the easiest part, the fills. The scheme we follow is *visual filling – going back – normal filling*. Visualizing is implemented using `\cleaders`. Because the ⟨*box*⟩ that follows this command is constructed only once, the `\copy` is not really a prerequisite. We prefer using a `\normalhbox` here instead of a `\hbox`.

```
\def\setvisiblehfilbox#1\to#2#3#4%
  {\setbox#1=\normalhbox
     {\vrule
        \!!width#2\testrulewidth
        \!!height#3\testrulewidth
        \!!depth#4\testrulewidth}%
   \smashbox#1}
```

```
\def\doruledhfiller#1#2#3#4%
  {#1#2%
   \bgroup
     \dontinterfere
     \dontcomplain
     \setvisiblehfilbox0\to{4}{#3}{#4}%
     \setvisiblehfilbox2\to422%
     \copy0\copy2
     \bgroup
       \setvisiblehfilbox0\to422%
       \cleaders
         \normalhbox to 12\testrulewidth
           {\normalhss\copy0\normalhss}%
         #1%
     \egroup
     \setbox0=\normalhbox
       {\normalhskip-4\testrulewidth\copy0\copy2}%
     \smashbox0
     \box0
   \egroup}
```

The horizontal fillers differ in their boundary visualization. Watch the small dots. Fillers can be combined within reasonable margins.

\hss test

\hfil test

\hfill test

\hfil\hfil . test . \hfil

The negative counterparts are visualizes, but seldom become visible, apart from their boundaries.

\hfilneg test

\hfillneg test

Although leaders are used for visualizing, they are visualized themselves correctly as the next example shows.

All five substitutions use the same auxiliary macro. Watch the positive first – negative next approach.

```
\def\ruledhss%
  {\doruledhfiller\normalhss\normalhfilneg{0}{0}}

\def\ruledhfil%
  {\doruledhfiller\normalhfil\normalhfilneg{10}{-6}}

\def\ruledhfill%
  {\doruledhfiller\normalhfill\normalhfillneg{18}{-14}}

\def\ruledhfilneg%
```

```
  {\doruledhfiller\normalhfilneg\normalhfil{-6}{10}}
```

39
```
\def\ruledhfillneg%
  {\doruledhfiller\normalhfillneg\normalhfill{-14}{18}}
```

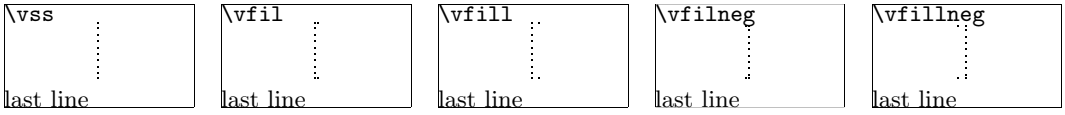
\ruledvss \ruledvfil \ruledvfilneg \ruledvfill \ruledvfillneg

The vertical mode commands adopt the same visualization scheme, but are implemented in a slightly different way.

```
\def\setvisiblevfilbox#1\to#2#3#4%
  {\setbox#1=\normalvcue
     {\vrule
        \!!width#2\testrulewidth
        \!!height#3\testrulewidth
        \!!depth#4\testrulewidth}%
   \smashbox#1}%
```

41
```
\def\doruledvfiller#1#2#3%
  {#1#2%
   \bgroup
     \dontinterfere
     \dontcomplain
     \offinterlineskip
     \setvisiblevfilbox0\to422%
     \setbox2=\normalhbox
       {\normalhskip -#3\testrulewidth\copy0}%
     \smashbox2
     \copy2
     \bgroup
       \setbox2=\normalhbox
         {\normalhskip -2\testrulewidth\copy0}%
       \smashbox2
       \copy2
       \cleaders
         \normalvbox to 12\testrulewidth
           {\normalvss\copy2\normalvss}%
         #1%
       \setbox2=\normalvbox
         {\vskip-2\testrulewidth\copy2}%
       \smashbox2
       \box2
     \egroup
     \setbox2=\normalvbox
       {\vskip-2\testrulewidth\copy2}%
     \smashbox2
     \box2
   \egroup}
```

Because they act the same as their horizontal counterparts we only show a few examples.



Keep in mind that **\vfillneg** is not part of PLAIN TEX, but are mimmicked by a macro.

42
```
\def\ruledvss%
```

```
  {\doruledvfiller\normalvss\normalvfilneg{2}}

\def\ruledvfil%
  {\doruledvfiller\normalvfil\normalvfilneg{-4}}

\def\ruledvfill%
  {\doruledvfiller\normalvfill\normalvfillneg{-12}}

\def\ruledvfilneg%
  {\doruledvfiller\normalvfilneg\normalvfil{8}}

\def\ruledvfillneg%
  {\doruledvfiller\normalvfillneg\normalvfill{16}}
```

\ruledhskip Skips differ from kerns in two important aspects:

- line and pagebreaks are allowed at a skip
- skips can have a positive and/or negative stretchcomponent

Stated a bit different: kerns are fixed skips at which no line or pagebreak can occur. Because skips have a more open character, they are visualized in a open way.


```
one
\hskip +30pt plus 5pt
two
\hskip +30pt
\hskip -10pt plus 5pt
three
\hskip   0pt
four
\hskip +30pt
five
```

When skips have a stretch component, this is visualized by means of a dashed line. Positive skips are on top of the baseline, negative ones are below it. This way we can show the combined results. An alternative visualization of stretch could be drawing the mid line over a length of the stretch, in positive or negative direction.

```
\def\doruledhskip%
  {\relax
   \dontinterfere
   \dontcomplain
   \investigateskip\scratchskip
   \ifzero
     \setbox0=\normalhbox
       {\normalhskip-\testrulewidth
        \vrule
          \!!width4\testrulewidth
          \!!height16\testrulewidth
          \!!depth16\testrulewidth}%
   \else
     \setbox0=\normalhbox to \ifnegative-\fi\scratchskip
       {\vrule
          \!!width2\testrulewidth
          \ifnegative\!!depth\else\!!height\fi16\testrulewidth
        \cleaders
          \hrule
```

```
            \ifnegative
              \!!depth2\testrulewidth
              \!!height\!!zeropoint
            \else
              \!!height2\testrulewidth
              \!!depth\!!zeropoint
            \fi
          \normalhfill
        \ifflexible
          \normalhskip\ifnegative\else-\fi\scratchskip
          \normalhskip2\testrulewidth
          \cleaders
            \normalhbox
              {\normalhskip 2\testrulewidth
               \vrule
                 \!!width2\testrulewidth
                 \!!height\ifnegative-7\else9\fi\testrulewidth
                 \!!depth\ifnegative9\else-7\fi\testrulewidth
               \normalhskip 2\testrulewidth}%
            \normalhfill
        \fi
        \vrule
          \!!width2\testrulewidth
          \ifnegative\!!depth\else\!!height\fi16\testrulewidth}%
      \setbox0=\normalhbox
        {\ifnegative\else\normalhskip-\scratchskip\fi
         \box0}%
    \fi
    \smashbox0%
    \ifvisiblestretch \else
      \flexiblefalse
    \fi
    \ifflexible
      % breaks ok but small displacements can occur
      \skip2=\scratchskip
      \advance\skip2 by -1\scratchskip
      \divide\skip2 by 2
      \advance\scratchskip by -\skip2
      \normalhskip\scratchskip
      \normalpenalty\!!tenthousand
      \box0
      \normalhskip\skip2
    \else
      \normalhskip\scratchskip
      \box0
    \fi
    \egroup}

48  \def\ruledhskip%
      {\bgroup
       \afterassignment\doruledhskip
       \scratchskip=}
```

The visual skip is located at a feasible point. Normally this does not interfere with the normaltypesetting process. The next examples show (1) the default behavior, (2) the (not entirely correct) distributed stretch and (3) the way the text is typeset without cues.

test test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test

test test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test

test test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test testtest test

\ruledvskip We are less fortunate when implementing the vertical skips. This is a direct result of interference between the boxes that visualize the skip and skip removal at a pagebreak. Normally skips disappear at the top of a page, but not of course when visualized in a `\vbox`. A quite perfect simulation could have been built if we would have had available two more primitives: `\hnop` and `\vnop`. These new primitives could stand for boxes that are visible but are not taken into account in any way. They are there for us, but not for TEX.

first line

second line

third line
fourth line

fifth line
sixth line

```
first line
\vskip +30pt plus 5pt
second line
\vskip +30pt
\vskip -10pt plus 5pt
third line
\par
fourth line
\vskip +30pt
fifth line
\vskip   0pt
sixth line
```

We have to postpone `\prevdepth`. Although this precaution probably is not completely waterproof, it works quite well.

49

```
\def\dodoruledvskip%
  {\nextdepth=\prevdepth
   \dontinterfere
   \dontcomplain
   \offinterlineskip
   \investigateskip\scratchskip
   \ifzero
     \setbox0=\normalvcue
       {\vrule
          \!!width32\testrulewidth
          \!!height2\testrulewidth
          \!!depth2\testrulewidth}%
   \else
     \setbox0=\normalvbox to \ifnegative-\fi\scratchskip
       {\hrule
          \!!width16\testrulewidth
```

```
      \!!height2\testrulewidth
    \ifflexible
      \cleaders
        \normalhbox to 16\testrulewidth
          {\normalhss
           \normalvbox
             {\normalvskip 2\testrulewidth
              \hrule
              \!!width2\testrulewidth
              \!!height2\testrulewidth
              \normalvskip 2\testrulewidth}%
           \normalhss}%
        \normalvfill
    \else
      \normalvfill
    \fi
    \hrule
      \!!width16\testrulewidth
      \!!height2\testrulewidth}%
  \setbox2=\normalvbox to \ht0
    {\hrule
       \!!width2\testrulewidth
       \!!height\ht0}%
  \ifnegative
    \ht0=\!!zeropoint
    \setbox0=\normalhbox
      {\normalhskip2\testrulewidth % will be improved
       \normalhskip-\wd0\box0}%
  \fi
  \smashbox0%
  \smashbox2%
  \setbox0=\normalvcue
    {\box2\box0}%
  \setbox0=\normalvbox
    {\ifnegative\normalvskip\scratchskip\fi\box0}%
  \smashbox0%
\fi
\ifvisiblestretch
  \ifflexible
    \skip2=\scratchskip
    \advance\skip2 by -1\scratchskip
    \divide\skip2 by 2
    \advance\scratchskip by -\skip2
    \normalvskip\skip2
  \fi
\fi
\normalpenalty\!!tenthousand
\box0
\prevdepth=\nextdepth % not \dp0=\nextdepth
\normalvskip\scratchskip}
```

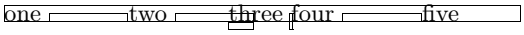
We try to avoid interfering at the top of a page. Of course we only do so when we are in the main vertical list.

```
50 \def\doruledvskip%
```

```
  {\par
   \ifdim\pagegoal=\maxdimen
     \ifinner
       \dodoruledvskip
     \fi
   \else
     \dodoruledvskip
   \fi
   \egroup}
```

51
```
\def\ruledvskip%
  {\bgroup
   \afterassignment\doruledvskip
   \scratchskip=}
```

\ruledkern The macros that implement the kerns are a bit more complicated than needed, because they also serve the visualization of glue, our PLAIN defined kerns with stretch or shrink. We've implemented both horizontal and vertical kerns as ruled boxes.




```
one
\kern +30pt
two
\kern +30pt
\kern -10pt
three
\kern   0pt
four
\kern +30pt
five
```

Positive and negative kerns are placed on top or below the baseline, so we are able to track their added result. We didn't mention spacings of 0 pt yet. Zero values are visualized a bit different, because we want to see them anyhow.

52
```
\def\doruledhkern%
  {\dontinterfere
   \dontcomplain
   \baselinerulefalse
   \investigateskip\scratchskip
   \boxrulewidth=2\testrulewidth
   \ifzero
     \setbox0=\ruledhbox to 8\testrulewidth
       {\vrule
          \!!width\!!zeropoint
          \!!height16\testrulewidth
          \!!depth16\testrulewidth}%
     \setbox0=\normalhbox
       {\normalhskip-4\testrulewidth\box0}%
   \else
     \setbox0=\ruledhbox to \ifnegative-\fi\scratchskip
       {\vrule
          \!!width\!!zeropoint
          \ifnegative\!!depth\else\!!height\fi16\testrulewidth
        \ifflexible
          \normalhskip2\testrulewidth
          \cleaders
```

```
            \normalhbox
              {\normalhskip 2\testrulewidth
               \vrule
                 \!!width2\testrulewidth
                 \!!height\ifnegative-7\else9\fi\testrulewidth
                 \!!depth\ifnegative9\else-7\fi\testrulewidth
               \normalhskip 2\testrulewidth}%
            \normalhfill
        \else
          \normalhfill
        \fi}%
      \testrulewidth=2\testrulewidth
      \setbox0=\ruledhbox{\box0}%  \make...
    \fi
    \smashbox0%
    \normalpenalty\!!tenthousand
    \normalhbox to \!!zeropoint
      {\ifnegative\normalhskip1\scratchskip\fi
       \box0}%
    \afterwards\scratchskip
    \egroup}
```

53
```
\def\ruledhkern#1%
  {\bgroup
   \let\afterwards=#1\relax
   \afterassignment\doruledhkern
   \scratchskip=}
```

After having seen the horizontal ones, the vertical kerns will not surprise us. In this example we use \par to switch to vertical mode.

first line

second line

third line
fourth line

fifth line
sixth line

```
first line
\par \kern +30pt
second line
\par \kern +30pt
\par \kern -10pt
third line
\par
fourth line
\par \kern +30pt
fifth line
\par \kern   0pt
sixth line
```

Like before, we have to postpone \prevdepth. If we leave out this trick, we got ourselves some wrong spacing.

54
```
\def\dodoruledvkern%
  {\nextdepth=\prevdepth
   \dontinterfere
   \dontcomplain
   \baselinerulefalse
   \offinterlineskip
   \investigateskip\scratchskip
   \boxrulewidth=2\testrulewidth
```

```
    \ifzero
      \setbox0=\ruledhbox to 32\testrulewidth
        {\vrule
           \!!width\!!zeropoint
           \!!height4\testrulewidth
           \!!depth4\testrulewidth}%
    \else
      \setbox0=\ruledvbox to \ifnegative-\fi\scratchskip
        {\hsize16\testrulewidth
         \ifflexible
           \cleaders
             \normalhbox to 16\testrulewidth
               {\normalhss
                \normalvbox
                  {\normalvskip 2\testrulewidth
                   \hrule
                     \!!width2\testrulewidth
                     \!!height2\testrulewidth
                   \normalvskip 2\testrulewidth}%
                \normalhss}%
             \normalvfill
         \else
           \vrule
             \!!width\!!zeropoint
             \!!height\ifnegative-\fi\scratchskip
           \normalhfill
         \fi}
    \fi
    \testrulewidth=2\testrulewidth
    \setbox0=\ruledvbox{\box0}%  \make...
    \smashbox0%
    \setbox0=\normalvbox
      {\ifnegative\normalvskip\scratchskip\fi
       \normalvcue
        {\ifnegative\normalhskip-16\testrulewidth\fi\box0}}%
    \smashbox0%
    \normalpenalty\!!tenthousand
    \box0
    \prevdepth=\nextdepth} % not \dp0=\nextdepth

55  \def\doruledvkern%
      {\ifdim\pagegoal=\maxdimen
         \ifinner
           \dodoruledvkern
         \fi
       \else
         \dodoruledvkern
       \fi
       \afterwards\scratchskip
       \egroup}

56  \def\ruledvkern#1%
      {\bgroup
       \let\afterwards=#1\relax
       \afterassignment\doruledvkern
```

```
  \scratchskip=}
```

57
```
\def\ruledkern%
  {\ifvmode
     \let\next=\ruledvkern
   \else
     \let\next=\ruledhkern
   \fi
   \next\normalkern}
```

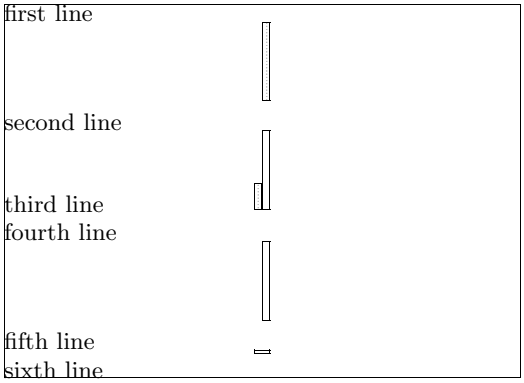
\ruledhglue
\ruledvglue

The non-primitive glue commands are treated as kerns with stretch. This stretch is presented as a dashed line. I have to admit that until now, I've never used these glue commands.


```
one
\hglue +30pt plus 5pt
two
\hglue +30pt
\hglue -10pt plus 5pt
three
\hglue   0pt
four
\hglue +30pt
five
```

58
```
\def\doruledhglue%
  {\leavevmode
   \scratchcounter=\spacefactor
   \vrule\!!width\!!zeropoint
   \normalpenalty\!!tenthousand
   \ruledhkern\normalhskip\scratchskip
   \spacefactor=\scratchcounter
   \egroup}
```

59
```
\def\ruledhglue%
  {\bgroup
   \afterassignment\doruledhglue\scratchskip=}
```




```
first line
\vglue +30pt plus 5pt
second line
\vglue +30pt
\vglue -10pt plus 5pt
third line
\par
fourth line
\vglue +30pt
fifth line
\vglue   0pt
sixth line
```

60
```
\def\doruledvglue%
  {\par
   \nextdepth=\prevdepth
   \hrule\!!height\!!zeropoint
   \normalpenalty\!!tenthousand
```

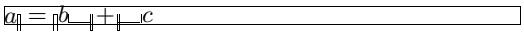
```
  \ruledvkern\normalvskip\scratchskip
  \prevdepth=\nextdepth
  \egroup}
```

61
```
\def\ruledvglue%
  {\bgroup
   \afterassignment\doruledvglue\scratchskip=}
```

\ruledmkern
\ruledmskip

Mathematical kerns and skips are specified in mu. This font related unit is incompatible with those of ⟨*dimensions*⟩ and ⟨*skips*⟩. Because in math mode spacing is often a very subtle matter, we've used a very simple, not overloaded way to show them.

62
```
\def\dodoruledmkern#1%
  {\dontinterfere
   \dontcomplain
   \setbox0=\normalhbox
     {$\normalmkern\ifnegative-\fi\scratchmuskip$}%
   \setbox0=\normalhbox to \wd0
     {\vrule
        \!!height16\testrulewidth
        \!!depth16\testrulewidth
        \!!width\testrulewidth
      \leaders
        \hrule
          \!!height\ifpositive16\else-14\fi\testrulewidth
          \!!depth\ifpositive-14\else16\fi\testrulewidth
        \normalhfill
      \ifflexible
        \normalhskip-\wd0
        \leaders
          \hrule
            \!!height\testrulewidth
            \!!depth\testrulewidth
          \normalhfill
      \fi
      \vrule
        \!!height16\testrulewidth
        \!!depth16\testrulewidth
        \!!width\testrulewidth}%
   \smashbox0%
   \ifnegative
      #1\scratchmuskip
     \box0
   \else
     \box0
      #1\scratchmuskip
   \fi
   \egroup}
```

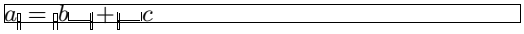


```
$a \mkern3mu = \mkern3mu
 b \quad
 \mkern-2mu + \mkern-2mu
 \quad c$
```

63
```
\def\doruledmkern%
  {\investigatemuskip\scratchmuskip
```

```
  \flexiblefalse
  \dodoruledmkern\normalmkern}
```

64
```
\def\ruledmkern%
  {\bgroup
   \afterassignment\doruledmkern\scratchmuskip=}
```

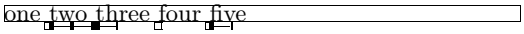


```
$a \mskip3mu = \mskip3mu
 b \quad
 \mskip-2mu + \mskip-2mu
 \quad c$
```

65
```
\def\doruledmskip%
  {\investigatemuskip\scratchmuskip
   \flexibletrue
   \dodoruledmkern\normalmskip}
```

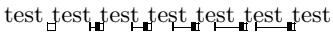
66
```
\def\ruledmskip%
  {\bgroup
   \afterassignment\doruledmskip\scratchmuskip=}
```

\penalty After presenting fills, skip, kerns and glue we've come to see penalties. In the first implementation — most of the time needed to develop this set of macros went into testing different types of visualization — penalties were mere small blocks with one black half, depending on the sign. This most recent version also gives an indication of the amount of penalty. Penalties can go from less than −10000 to over +10000, and their behavior is somewhat non-linear, with some values having special meanings. We therefore decided not to use its value for a lineair indicator.

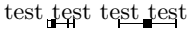


```
one
\penalty +100
two
\penalty +100
\penalty -100
three
\penalty    0
four
\penalty +100
five
```

The small sticks at the side of the penalty indicate it size. The next example shows the positive and negative penalties of 0, 1, 10, 100, 1000 and 10000.



This way stacked penalties of different severance can be shown in combination.



67
```
\def\setruledpenaltybox#1#2#3#4#5#6%
  {\setbox#1=\normalhbox
     {\ifnum#2=0 \else
        \ifnum#2>0
          \def\sign{+}%
        \else
          \def\sign{-}%
```

```
      \fi
      \dimen0=\ifnum\sign#2>9999
                28\else
              \ifnum\sign#2>999
                22\else
              \ifnum\sign#2>99
                16\else
              \ifnum\sign#2>9
                10\else
                4
              \fi\fi\fi\fi \testrulewidth
      \ifnum#2<0
        \normalhskip-\dimen0
        \normalhskip-2\testrulewidth
        \vrule
          \!!width2\testrulewidth
          \!!height#3\testrulewidth
          \!!depth#4\testrulewidth
      \fi
      \vrule
        \!!width\dimen0
        \!!height#5\testrulewidth
        \!!depth#6\testrulewidth
      \ifnum#2>0
        \vrule
          \!!width2\testrulewidth
          \!!height#3\testrulewidth
          \!!depth#4\testrulewidth
       \fi
    \fi}%
  \smashbox#1}
```

68

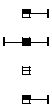
```
\def\doruledhpenalty%
  {\dontinterfere
   \dontcomplain
   \investigatecount\scratchcounter
   \testrulewidth=2\testrulewidth
   \boxrulewidth=\testrulewidth
   \setbox0=\ruledhbox to 8\testrulewidth
     {\ifnegative\else\normalhss\fi
      \vrule
        \!!depth8\testrulewidth
        \!!width\ifzero0\else4\fi\testrulewidth
      \ifpositive\else\normalhss\fi}%
   \setruledpenaltybox{2}{\scratchcounter}{0}{8}{-3.5}{4.5}%
   \normalpenalty\!!tenthousand
   \setbox0=\normalhbox
     {\normalhskip-4\testrulewidth
      \ifnegative
        \box2\box0
      \else
        \box0\box2
      \fi}%
   \smashbox0%
   \box0
```

```
  \normalpenalty\scratchcounter
  \egroup}
```

69
```
\def\ruledhpenalty%
  {\bgroup
   \afterassignment\doruledhpenalty
   \scratchcounter=}
```

The size of a vertical penalty is also shown on the horizontal axis. This way there is less interference with the often preceding or following skips and kerns.

first line
second line
third line
fourth line
fifth line



```
first line
\par \penalty +100
second line
\par \penalty +100
\par \penalty -100
third line
\par \penalty    0
fourth line
\par \penalty +100
fifth line
```

70
```
\def\doruledvpenalty%
  {\ifdim\pagegoal=\maxdimen
   \else
     \nextdepth=\prevdepth
     \dontinterfere
     \dontcomplain
     \investigatecount\scratchcounter
     \testrulewidth=2\testrulewidth
     \boxrulewidth=\testrulewidth
     \setbox0=\ruledhbox
       {\vrule
          \!!height4\testrulewidth
          \!!depth4\testrulewidth
          \!!width\!!zeropoint
        \vrule
          \!!height\ifnegative.5\else4\fi\testrulewidth
          \!!depth\ifpositive.5\else4\fi\testrulewidth
          \!!width8\testrulewidth}%
     \setruledpenaltybox{2}{\scratchcounter}{4}{4}{.5}{.5}%
     \setbox0=\normalhbox
       {\normalhskip-4\testrulewidth
        \ifnegative
          \box2\box0
        \else
          \box0\box2
        \fi
        \normalhss}%
     \smashbox0%
     \normalpenalty\!!tenthousand
     \nointerlineskip
     \dp0=\nextdepth  % not \prevdepth=\nextdepth
     \normalvbox
       {\normalvcue{\box0}}%
```

```
  \fi
  \normalpenalty\scratchcounter
  \egroup}
```

71
```
\def\ruledvpenalty%
  {\bgroup
   \afterassignment\doruledvpenalty
   \scratchcounter=}
```

72
```
\def\ruledpenalty%
  {\ifvmode
     \let\next=\ruledvpenalty
   \else
     \let\next=\ruledhpenalty
   \fi
   \next}
```

\showfils \dontshowfils \showboxes \dontshowboxes \showskips \dontshowskips \showpenalties \dontshowpenalties

For those who want to manipulate the visual cues in detail, we have grouped them.

73
```
\def\showfils%
  {\let\hss      = \ruledhss
   \let\hfil     = \ruledhfil
   \let\hfill    = \ruledhfill
   \let\hfilneg  = \ruledhfilneg
   \let\hfillneg = \ruledhfillneg
   \let\vss      = \ruledvss
   \let\vfil     = \ruledvfil
   \let\vfill    = \ruledvfill
   \let\vfilneg  = \ruledvfilneg
   \let\vfillneg = \ruledvfillneg}
```

74
```
\def\dontshowfils%
  {\let\hss      = \normalhss
   \let\hfil     = \normalhfil
   \let\hfill    = \normalhfill
   \let\hfilneg  = \normalhfilneg
   \let\hfillneg = \normalhfillneg
   \let\vss      = \normalvss
   \let\vfil     = \normalvfil
   \let\vfill    = \normalvfill
   \let\vfilneg  = \normalvfilneg
   \let\vfillneg = \normalvfillneg}
```

75
```
\def\showboxes%
  {\baselineruletrue
   \let\hbox     = \ruledhbox
   \let\vbox     = \ruledvbox
   \let\vtop     = \ruledvtop}
```

76
```
\def\dontshowboxes%
  {\let\hbox     = \normalhbox
   \let\vbox     = \normalvbox
   \let\vtop     = \normalvtop}
```

77
```
\def\showskips%
  {\let\hskip    = \ruledhskip
   \let\vskip    = \ruledvskip
```

```
  \let\kern     = \ruledkern
  \let\mskip    = \ruledmskip
  \let\mkern    = \ruledmkern
  \let\hglue    = \ruledhglue
  \let\vglue    = \ruledvglue}
```

78
```
\def\dontshowskips%
  {\let\hskip    = \normalhskip
   \let\vskip    = \normalvskip
   \let\kern     = \normalkern
   \let\mskip    = \normalmskip
   \let\mkern    = \normalmkern
   \let\hglue    = \normalhglue
   \let\vglue    = \normalvglue}
```

79
```
\def\showpenalties%
  {\let\penalty  = \ruledpenalty}
```

80
```
\def\dontshowpenalties%
  {\let\penalty  = \normalpenalty}
```

\showingcomposition
\showcomposition
\dontshowcomposition

All these nice options come together in two macros. The first one turns the options on, the second turnes them off. Both macros only do their job when we are actually showing the composition.

```
\showingcompositiontrue
\showcomposition
```

Because the output routine can do tricky things, like multiple column typesetting and manipulation of the pagebody, shifting things around and so on, the macro **\dontshowcomposition** best can be called when we enter this routine. Too much visual cues just don't make sense. In CONTEXT this has been taken care of.

81
```
\newif\ifshowingcomposition
```

82
```
\def\showcomposition%
  {\ifshowingcomposition
     \showfils
     \showboxes
     \showskips
     \showpenalties
   \fi}
```

83
```
\def\dontshowcomposition%
  {\ifshowingcomposition
     \dontshowfils
     \dontshowboxes
     \dontshowskips
     \dontshowpenalties
   \fi}
```

\showmakeup
\defaulttestrulewidth

Just to make things even more easy, we have defined:

```
\showmakeup
```

For the sake of those who don't (yet) use CONTEXT we preset **\defaulttestrulewidth** to the already set value. Otherwise we default to a corps related value.

```
\def\defaulttestrulewidth{.2pt}
```

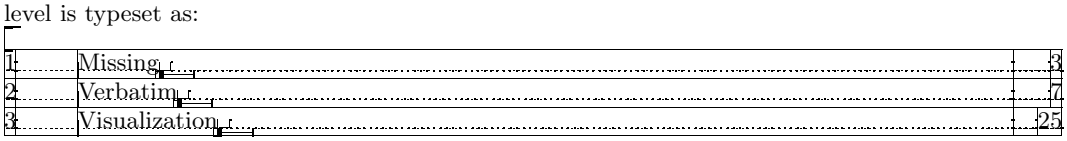
Beware, it's a macro not a ⟨*dimension*⟩.

84
```
\ifx\korpsgrootte\undefined
  \edef\defaulttestrulewidth{\the\testrulewidth}
\else
  \def\defaulttestrulewidth{.02\korpsgrootte}  % still dutch
\fi
```

85
```
\def\showmakeup%
  {\testrulewidth=\defaulttestrulewidth
   \showingcompositiontrue
   \showcomposition}
```

86
```
\protect
```

Lets end with some more advanced examples. When visualized, the table of contents of the outer level is typeset as:



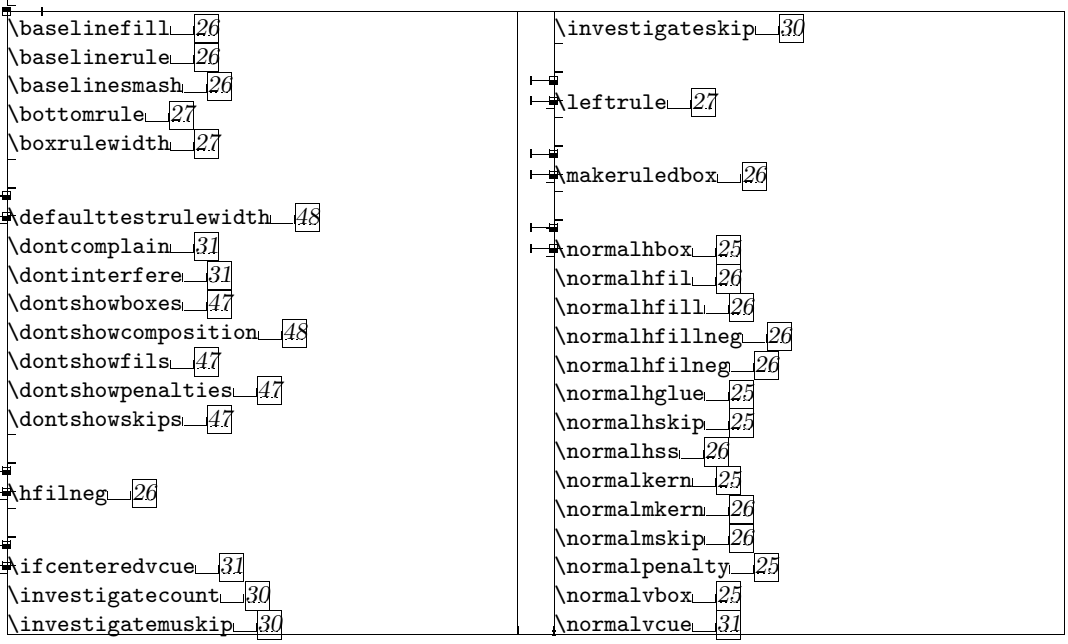
Definitions and enumerations come in many flavors. The next one for instance is defined as:

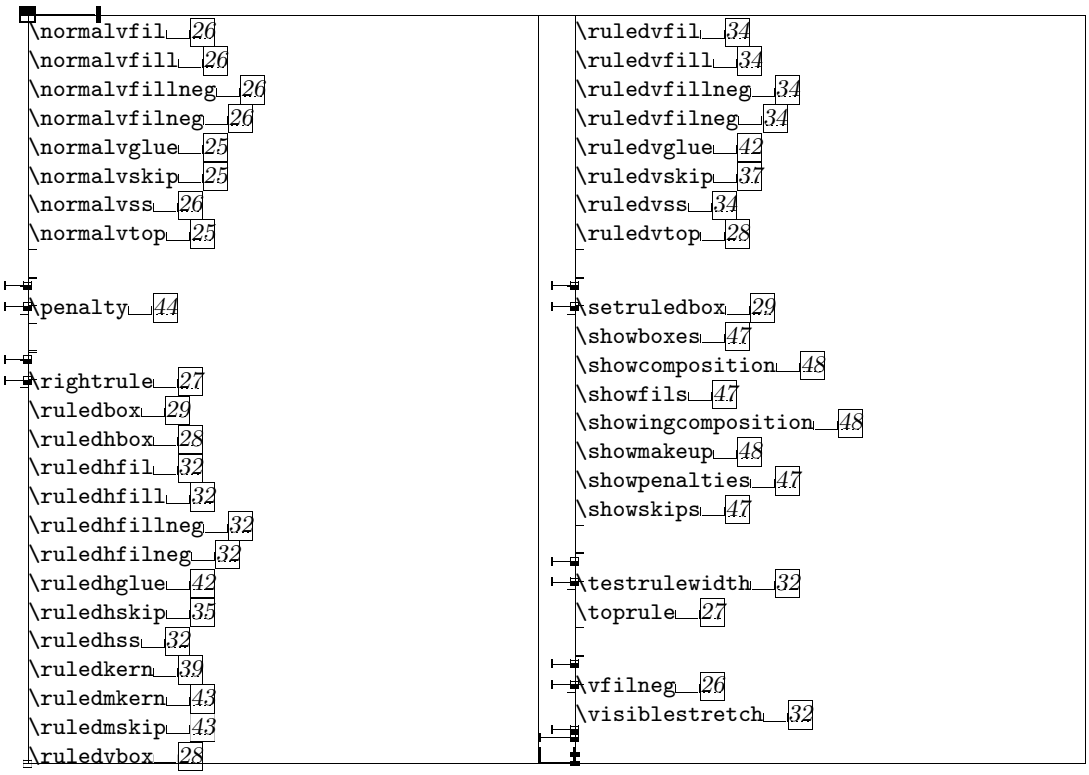
```
\definedescription[test][place=left,hang=3,width=6em]
```

When applied to some text, this would look like:

visual debugger I would be very pleased if TEX had two more primitives: **\vnop** and **\hnop**. Both should act and show up as normal boxes, but stay invisible for TEX when it's doing calculations. The **\vnop** for instance should not interact with the internal mechanism responsible for the disappearing skips, kerns and penalties at a pagebreak. As long as we don't have these two boxtypes, visual debugging will never be perfect.

The index to this section looks like:





Although not impressive examples or typesetting, both show us how and where things happen. When somehow the last lines in this two column index don't allign, then this is due to some still unknown interference.